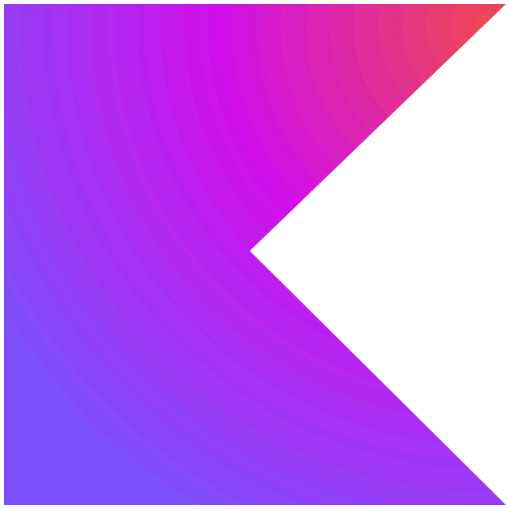


Kotlin Language Documentation 2.4.20

Table of Contents

Kotlin Docs **74**

Get started with Kotlin **74**

Install Kotlin 74

Choose your Kotlin use case 74

Get support 75

Welcome to our tour of Kotlin! **75**

Hello world **75**

Variables 76

String templates 76

Practice 77

Basic types **77**

Practice 79

Collections **80**

List 80

Set 81

Map 83

Practice 85

Control flow **86**

Conditional expressions 86

Conditional expressions practice 89

Ranges 90

Loops 91

Loops practice 92

Functions **94**

Named arguments 95

Default parameter values 96

Functions without return 96

Single-expression functions 97
Early returns in functions 97
Functions practice 98
Lambda expressions 100
Lambda expressions practice 104

Classes 105
Properties 105
Create instance 105
Access properties 106
Member functions 106
Data classes 107
Practice 109

Null safety 111
Nullable types 112
Check for null values 112
Use safe calls 113
Use Elvis operator 113
Practice 114
What's next? 115

Extension functions 115
Extension functions 115
Extension-oriented design 116
Practice 117

Scope functions 118
Scope functions 118
Use case overview 125
Practice 126

Lambda expressions with receiver 128
Lambda expressions with receiver 128

Practice....131

Classes and interfaces....133

Class inheritance....133

Interfaces....136

Delegation....138

Practice....141

Objects....146

Object declarations....146

Practice....149

Open and special classes....153

Open classes....153

Special classes....155

Practice....159

Properties....163

Backing fields....163

Extension properties....165

Delegated properties....166

Practice....171

Null safety....175

Smart casts and safe casts....175

Null values and collections....177

Early returns and the Elvis operator....180

Practice....181

Libraries and APIs....186

The standard library....187

Kotlin libraries....188

Opt in to APIs....189

Practice....190

What's next?....192

What's new in Kotlin 2.4.20 192
Update to Kotlin 2.4.20 192
StableNew features 193
New features 193
Standard library 193
Kotlin/Native 197
Kotlin/Wasm 200
Kotlin/JS 202
Gradle 205
Build tools API 205
Kotlin compiler 206
Breaking changes and deprecations 207
Documentation updates 207
What's new in Kotlin 2.4.0 208
Update to Kotlin 2.4.0 209
StableNew features 209
New features 209
Language 210
Standard library 216
Kotlin/JVM 219
Kotlin/Native 219
Kotlin/Wasm 222
Kotlin/JS 223
Gradle 227
Maven 227
Build tools API 229
Kotlin compiler 230
Kotlin compiler plugins 232
Compose compiler 233
Breaking changes and deprecations 234

Documentation updates 234

Compatibility guide for Kotlin 2.4.x 235

Basic terms 235

Language 236

Standard library 245

Tools 246

Basic syntax overview 252

Package definition and imports 252

Program entry point 252

Print to the standard output 253

Read from the standard input 253

Functions 253

Variables 254

Creating classes and instances 255

Comments 256

String templates 256

Conditional expressions 257

for loop 257

while loop 257

when expression 258

Ranges 258

Collections 259

Nullable values and null checks 259

Type checks and automatic casts 261

Keywords and operators 262

Hard keywords 262

Soft keywords 264

Modifier keywords 264

Special identifiers 265

Operators and special symbols 266

Packages and imports 267

Package headers 267

Imports 267

Default imports 269

Visibility and imports 269

Annotations 270

Declaration 270

Usage 270

Constructors 271

Instantiation 271

Lambdas 272

Annotation use-site targets 272

Java annotations 274

Repeatable annotations 276

Visibility modifiers 277

Packages 277

Class members 277

Modules 279

Coding conventions 279

Configure style in IDE 279

Source code organization 279

Naming rules 281

Formatting 283

Documentation comments 292

Avoid redundant constructs 293

Idiomatic use of language features 294

Coding conventions for libraries 298

Idioms 299

Create DTOs (POJOs/POCOs) 299

Default values for function parameters 299
Filter a list 299
Check the presence of an element in a collection 299
String interpolation 300
Read standard input safely 300
Instance checks 300
Read-only list 300
Read-only map 300
Access a map entry 300
Traverse a map or a list of pairs 300
Iterate over a range 301
Lazy property 301
Extension functions 301
Create a singleton 301
Use inline value classes for type-safe values 301
Instantiate an abstract class 302
If-not-null shorthand 302
If-not-null-else shorthand 302
Execute an expression if null 302
Get first item of a possibly empty collection 302
Execute if not null 303
Map nullable value if not null 303
Return on when statement 303
try-catch expression 303
if expression 303
Builder-style usage of methods that return Unit 304
Single-expression functions 304
Call multiple methods on an object instance (with) 304
Configure properties of an object (apply) 304
Java 7's try-with-resources 305

Generic function that requires the generic type information 305
Swap two variables 305
Mark code as incomplete (TODO) 305
What's next? 305

Types overview 306

Numbers 306

Choose a number type 306
Integer types 307
Floating-point types 308
Arithmetic operations 309
Type conversion 309
Data overflow 312
Bitwise operations 313
Floating-point number comparison 313
Boxing and caching numbers on the JVM 314

Unsigned integer types 314

Unsigned arrays and ranges 315
Unsigned integers literals 315
Use cases 316

Booleans 316

Declare a Boolean variable 317
Produce Boolean values 317
Boolean operations 317
Operator precedence 318
Boolean in conditions 319

Characters 320

Syntax 320
Unicode support 321
Escape sequences 321

Operations 322

Strings 323

Declare strings 324

String templates 325

Basic string operations 326

String conversion 330

Arrays 331

When to use arrays 331

Create arrays 332

Work with arrays 335

Convert to collections 338

What's next? 339

Type checks and casts 340

Checks with is and !is operators 340

Type casts 341

Smart casts 342

as and as? cast operators 347

Type aliases 350

Declare type aliases 351

Conditions and loops 353

If expression 353

When expressions and statements 354

For loops 360

While loops 363

Break and continue in loops 364

Returns and jumps 364

Break and continue labels 364

Return to labels 365

Exception and error handling 366

Throw exceptions 366
Handle exceptions using try-catch blocks 369
Create custom exceptions 373
The Nothing type 375
Exception classes 376
Stack trace 379
Exception interoperability with Java, Swift, and Objective-C 380

Functions 380
Function usage 380
Function scope 388
Generic functions 390
Tail recursive functions 390

Higher-order functions and lambdas 391
Higher-order functions 391
Function types 392
Lambda expressions and anonymous functions 394

Type-safe builders 397
How it works 401
Scope control: @DslMarker 403

Using builders with builder type inference 407
Writing your own builders 408
How builder inference works 409

Context parameters 412
Context parameters resolution 413
Restrictions 415

Inline functions 415
noinline 416
Non-local jump expressions 416
Reified type parameters 418

Inline properties 419
Restrictions for public API inline functions 419

Operator overloading 419

Unary operations 420
Binary operations 421
Infix calls for named functions 424

ExperimentalUnused return value checker 425

Configure the unused return value checker 425
Mark functions to check ignored results 426
Suppress reports for ignored results 427
Check for unused results in higher-order functions 428
Interoperability with Java annotations 429

Classes 429

Creating instances 430
Constructors and initializer blocks 431
Inheritance 437
Abstract classes 437
Companion objects 438

Data classes 439

Properties declared in the class body 440
Copying 440
Data classes and destructuring declarations 441
Standard data classes 441

Extensions 441

Receivers 442
Extension functions 442
Extension properties 446
Companion object extensions 447
Declaring extensions as members 448

Extensions and visibility modifiers 450
Scope of extensions 451

Interfaces 452
Implementing interfaces 452
Properties in interfaces 452
Interfaces Inheritance 453
Resolving overriding conflicts 453
JVM default method generation for interface functions 454

Delegation 455
Overriding a member of an interface implemented by delegation 455

Inheritance 456
Open keyword 457
Overriding methods 458
Overriding properties 458
Derived class initialization order 459
Calling the superclass implementation 459
Overriding rules 460

Object declarations and expressions 461
Object declarations 461
Object expressions 467
Behavior difference between object declarations and expressions 471

This expressions 471
Qualified this 472
Implicit this 475

Sealed classes and interfaces 475
Declare a sealed class or interface 476
Inheritance 478
Use sealed classes with when expression 479
Use case scenarios 480

What's next 483

Enum classes 483

Anonymous classes 483

Implementing interfaces in enum classes 484

Working with enum constants 484

Inline value classes 485

Members 486

Inheritance 487

Representation 487

Inline classes vs type aliases 488

Inline classes and delegation 489

Nested and inner classes 489

Inner classes 490

Anonymous inner classes 490

Functional (SAM) interfaces 491

SAM conversions 491

Migration from an interface with constructor function to a functional interface 492

Functional interfaces vs. type aliases 492

Properties 493

Declaring properties 493

Custom getters and setters 495

Backing fields 497

Compile-time constants 500

Late-initialized properties and variables 501

Overriding properties 502

Delegated properties 502

Delegated properties 502

Standard delegates 504

Delegating to another property 504

Storing properties in a map 505
Local delegated properties 506
Property delegate requirements 506
Translation rules for delegated properties 508
Providing a delegate 509

Null safety 511
Nullable types and non-nullable types 512
Check for null with the if conditional 513
Safe call operator 514
Elvis operator 515
Not-null assertion operator 515
Nullable receiver 516
Let function 517
Safe casts 517
Collections of a nullable type 518
What's next? 518

Equality 518
Structural equality 518
Referential equality 520
Floating-point numbers equality 520
Array equality 520

Generics: in, out, where 521
Variance 521
Type projections 524
Generic functions 526
Generic constraints 526
Definitely non-nullable types 527
Type erasure 527
Underscore operator for type arguments 529

Asynchronous programming techniques 530

Threading 530

Callbacks 531

Futures, promises, and others 532

Reactive extensions 532

Coroutines 533

Coroutines 533

Coroutine concepts 534

What's next 535

Reflection 535

JVM dependency 535

Class references 536

Callable references 537

Destructuring declarations 540

Example: returning two values from a function 541

Example: destructuring declarations and maps 541

Underscore for unused variables 542

Destructuring in lambdas 542

ExperimentalName-based destructuring 543

Backend development with Kotlin 544

Get started 545

Explore frameworks 545

Deploy your applications 546

Join the community 546

Next steps 546

Adding Kotlin to a Java project – tutorial 546

Project configuration 547

Project structure 549

Convert Java files to Kotlin 549

Explore compiler plugins 550
Next step 550

Test Java code using Kotlin and JUnit – tutorial....551
Configure the project 551
Add your first Kotlin test....555
Run tests 557
What's next....559

Test Kotlin projects with Maven 559
Create tests with JUnit 559
Run tests....561
Get detailed failure messages....562
Explore other testing frameworks 564
What's next....565

Use annotation processors in Kotlin projects 565
Use kapt with Java annotation processors 565
Use KSP in Gradle projects....567
What's next....569

Code quality tools in Kotlin projects 569
Code formatting with ktlint 569
Code analysis with detekt 571
Code quality with SonarSource 573
Code coverage with Kover....575
Other tools....577
What's next 578

Get started with Spring Boot and Kotlin 578
Next step....578

Create a Spring Boot project with Kotlin 579
Before you start 579
Create a Spring Boot project 579

Explore the project Gradle build file....582
Explore the generated Spring Boot application....584
Create a controller....584
Run the application....585
Next step....587

Add a data class to Spring Boot project....587

Update your application....587
Run the application....590
Next step....590

Add database support for Spring Boot project....590

Add database support....590
Update the MessageController class....592
Update the MessageService class....592
Configure the database....593
Add messages to database via HTTP request....594
Retrieve messages by id....596
Run the application....600
Next step....601

Use Spring Data CrudRepository for database access....601

Update your application....601
Run the application....604
What's next....605

Create a task manager app with Spring Boot and Claude....605

Set up the environment....605
Create a project....605
Create a development plan....607
Review the commits....609
Run the app....609
Test the app....610

Make refinements 611
What's next? 612

Overview 612

Kotlin/JS 612
BetaKotlin/Wasm 613
Choose your web approach 613
Compatibility mode for web targets 614

Kotlin/JavaScript 614

Use cases for Kotlin/JS 614
Get started with Kotlin/JS 615
Sample projects for Kotlin/JS 616
Kotlin/JS frameworks 616
Join the Kotlin/JS community 616
What's next 616

Get started with Kotlin/JS 617

Create your application in IntelliJ IDEA 617
Create your application using Gradle 624
What's next? 627

Set up a Kotlin/JS project 628

Execution environments 629
Support for ES2015 features 629
Configure output granularity 630
ExperimentalGeneration of TypeScript declaration files (d.ts) 630
Dependencies 631
run task 633
test task 634
webpack bundling 637
CSS 638
Node.js 640

Yarn....642
Distribution target directory....645
Module name....645
package.json customization....646
Run Kotlin/JS....646
Run the Node.js target....647
Run the browser target....648
Development server and continuous compilation....649
Debug Kotlin/JS code....652
Debug in browser....652
Debug in the IDE....654
Debug in Node.js....657
What's next?....657
If you run into any problems....657
Run tests in Kotlin/JS....657
Add test dependencies....658
Configure browsers....658
Add a test....659
Run tests....660
ExperimentalAdvanced configuration....663
ExperimentalConfiguration for plugin authors....664
Leave feedback....665
Kotlin/JS frameworks....665
Kobweb....666
Kilua....666
Summon....666
Kinetica....666
Kotlin React....667
KVision....667

fritz2 667
Doodle 667

Kotlin/JS compiler features 667
Lazy initialization of top-level properties 668
Incremental compilation for development binaries 668
Minification of member names in production 668
Dead code elimination 669

ExperimentalJS plain objects compiler plugin 669
Plain JS objects 670
Enable the plugin 670
Declare a plain object type 670
Use plain objects 671
What's next 672

BetaKotlin/Wasm 672
Kotlin/Wasm and Compose Multiplatform 672
Kotlin/Wasm and WASI 673
Kotlin/Wasm performance 673
Browser API support 674
Leave feedback 675
Learn more 675

BetaGet started with Kotlin/Wasm and Compose Multiplatform 675
Create a project 675
Run the application 676
Generate artifacts 677
Publish the application 679
What's next? 680

BetaGet started with Kotlin/Wasm and WASI 681
Before you start 681
Run the application 681

Test the application 683
What's next? 685

BetaDebug Kotlin/Wasm code 685
Before you start 685
Debug in IntelliJ IDEA 685
Debug in your browser 688
Leave feedback 692
What's next? 692

BetaInteroperability with JavaScript 692
Use JavaScript code in Kotlin 692
Use Kotlin code in JavaScript 697
Type correspondence 698
Exception handling 700
Kotlin/Wasm and Kotlin/JS interoperability differences 700
Web-related browser APIs 702

BetaSupported versions and configuration 702
Browser versions 703
Wasm proposals support 704
Use default import 704
Kotlin/Wasm incremental compilation 705
Diagnostics in fully qualified class names 705
Array out-of-bounds access and traps 706
Experimental annotations 706
Reloads during debugging 706

Kotlin/Native 707
Why Kotlin/Native? 707
Target platforms 707
Interoperability 708
Sharing code between platforms 708

Memory manager 708

Get started with Kotlin/Native 708

In IDE 709

Using Gradle 713

Using the command-line compiler 716

What's next? 717

Kotlin/Native libraries 717

Library compilation 717

klib utility 718

Creating and using a library 719

Library search sequence 720

Library format 720

Using relative paths in klibs 721

What's next? 721

Platform libraries 722

POSIX bindings 722

Popular native libraries 722

What's next 723

Kotlin/Native memory management 723

Garbage collector 723

Memory consumption 725

Unit tests in the background 728

What's next 728

Integration with Swift/Objective-C ARC 728

Threads 729

Garbage collection and lifecycle 730

Support for background state and App Extensions 734

What's next 734

Migrate to the new memory manager 734

Update Kotlin 735
Update dependencies 735
Update your code 735
What's next 736

Kotlin/Native binary options 736
How to enable 736
Binary options 737
What's next 740

Debugging Kotlin/Native 740
Generate binaries with debug information 741
Set breakpoints 742
Use stepping 743
Inspect variables 743
Debug iOS applications 744
Known issues 745
What's next? 745

Kotlin/Native supported targets and hosts 745
Target tiers 745
Hosts 749
What's next? 749

Tips for improving compilation time 750
General recommendations 750
Gradle configuration 750
Windows configuration 753
LLVM configuration 753

Tips for customizing LLVM backend 753
Examine the build log 753
Customize LLVM optimization passes 755

License files for the Kotlin/Native binaries 755

Kotlin/Native FAQ 757
How do I run my program? 757
What is Kotlin/Native memory management model? 757
How do I create a shared library? 757
How do I create a static library or an object file? 757
How do I run Kotlin/Native behind a corporate proxy? 757
How do I specify a custom Objective-C prefix/name for my Kotlin framework? 758
How do I rename the iOS framework? 758
How do I enable bitcode for my Kotlin framework? 758
How do I get better crash reports for iOS? 759
How do I reference objects safely from different coroutines? 759
How can I compile my project with unreleased versions of Kotlin/Native? 759

Kotlin for Android 759

Introduction to library authors' guidelines 760
What's next 761

Minimizing mental complexity overview 761
Next step 762

Simplicity 762
Use explicit API mode 762
Reuse existing concepts 762
Define and build on top of core API 762
Next step 763

Readability 763
Prefer Explicit Composability 763
Use DSLs 764
Use extension functions and properties 765
Avoid using the boolean type as an argument 765
Use numeric types appropriately 766
Next step 766

Consistency **766**

Preserve parameter order, naming and usage 767

Use Object-Oriented design for data and state 767

Choose the appropriate error handling mechanism 767

Maintain conventions and quality 768

Next step 768

Predictability **768**

Do the right thing by default 768

Allow opportunities for extension 769

Prevent unwanted and invalid extensions 769

Avoid exposing mutable state 770

Validate inputs and state 770

Next step 771

Debuggability **771**

Provide a toString method for stateful types 771

Adopt and document a policy for handling exceptions 774

ExperimentalSupport coroutine stack trace recovery for custom exceptions 774

Next step 775

Testability **775**

Avoid global state and stateful top-level functions 775

What's next 776

Backward compatibility guidelines for library authors **776**

Compatibility types 776

Choose compatible language and API versions 777

Use the Binary compatibility validator 777

Specify return types explicitly 778

Avoid adding arguments to existing API functions 778

Avoid widening or narrowing return types 779

Avoid using data classes in your API 780

Avoid changing annotation targets 781
Considerations for using the PublishedApi annotation 782
Evolve APIs pragmatically....782
Use the RequiresOptIn mechanism....782
What's next 783

Best practices for library authors to create informative documentation 783

Provide comprehensive documentation 784
Create personas for your users 784
Document by example whenever possible 784
Thoroughly document your API 785
Document lambda parameters....785
Use explicit links in documentation 786
Be self-contained where possible....786
Use simple English....786
What's next....786

Building a Kotlin library for multiplatform....786

Maximize your reach....787
Design APIs for use from common code 787
Ensure consistent behavior across platforms 787
Test on all platforms 788
Consider non-Kotlin users 788
Promote your library 788

Get started with Kotlin custom scripting – tutorial 788

Project structure 789
Before you start 789
Create a project....789
Add scripting modules....790
Create a script definition....792
Create a scripting host....794
Run scripts....796

What's next? 797

IDEs for Kotlin development 797

IntelliJ IDEA 797

Android Studio 797

Visual Studio Code 798

Other IDEs support 798

Compatibility with the Kotlin language versions 798

What's next? 798

Kotlin Language Server 798

Kotlin in Visual Studio Code 798

Supported features 799

Feedback 799

What's next? 799

Introduction 799

Community 800

Get started with Dokka 800

Gradle 802

Supported versions 802

Apply Dokka 803

Enable build cache and configuration cache 803

Generate documentation 803

Build javadoc.jar 806

Configuration examples 807

Dokka Gradle configuration options 813

General configuration 813

Source set configuration 817

Source link configuration 821

Package options 823

External documentation links configuration 824

Complete configuration 825

Dokka Gradle troubleshooting 829

Memory issues 829

Migrate to Dokka Gradle plugin v2 830

Before you start 831

Migrate your project 832

Finalize your migration 843

What's next 844

Maven 844

Apply Dokka 844

Generate documentation 845

Build javadoc.jar 845

Configuration example 846

Configuration options 847

CLI 855

Get started 855

Generate documentation 855

Command line options 858

JSON configuration 860

HTML 871

Generate HTML documentation 871

Configuration 872

Customization 874

Javadoc 877

Generate Javadoc documentation 878

Dokka plugins 879

Apply Dokka plugins 879

Configure Dokka plugins 881

Notable plugins 882

Module documentation 883

File format 883

Pass files to Dokka 884

Run code snippets 884

IDE: scratches 884

Browser: Kotlin Playground 886

Command line: ki shell 889

Document Kotlin code: KDoc 891

KDoc syntax 891

Inline markup 893

What's next? 894

Kotlin and OSGi 894

Maven 894

Gradle 894

FAQ 895

Kotlin and continuous integration with TeamCity 895

Gradle and Maven 896

IntelliJ IDEA Build System 896

Other CI servers 898

Comparison to Java 898

Some Java issues addressed in Kotlin 898

What Java has that Kotlin does not 898

What Kotlin has that Java does not 899

What's next? 900

Calling Java from Kotlin 900

Getters and setters 900

Java synthetic property references 901

Methods returning void 902

Escaping for Java identifiers that are keywords in Kotlin 902

Null-safety and platform types 902
Mapped types 910
Java generics in Kotlin 912
Java arrays 913
Java varargs 913
Operators 914
Checked exceptions 914
Object methods 914
Inheritance from Java classes 916
Accessing static members 916
Java reflection 916
SAM conversions 917
Using JNI with Kotlin 917
Using Lombok-generated declarations in Kotlin 917

Calling Kotlin from Java 918

Properties 918
Package-level functions 918
Instance fields 919
Static fields 920
Static methods 921
Default methods in interfaces 922
Visibility 923
KClass 924
Handling signature clashes with @JvmName 924
Overloads generation 925
Checked exceptions 926
Null-safety 927
Variant generics 927
ExperimentalInline value classes 929

Using Java records in Kotlin 930

Using Java records from Kotlin code 930
Declare records in Kotlin 930
ExperimentalAnnotate record components in Kotlin 931
Make annotations work with record components 932
Further discussion 932

Strings in Java and Kotlin 932
Concatenate strings 932
Build a string 932
Create a string from collection items 933
Set default value if the string is blank 934
Replace characters at the beginning and end of a string 934
Replace occurrences 934
Split a string 935
Take a substring 935
Use multiline strings 936
What's next? 937

Collections in Java and Kotlin 937
Operations that are the same in Java and Kotlin 938
Operations that differ a bit 940
Operations that don't exist in Java's standard library 941
Mutability 941
Covariance 942
Ranges and progressions 943
Comparison by several criteria 944
Sequences 945
Removal of elements from a list 945
Traverse a map 946
Get the first and the last items of a possibly empty collection 946
Create a set from a list 947
Group elements 947

Filter elements....948
Collection transformation operations....950
What's next?....951

Nullability in Java and Kotlin....951
Support for nullable types....951
Platform types....953
Support for definitely non-nullable types....953
Checking the result of a function call....954
Default values instead of null....955
Functions returning a value or null....955
Aggregate operations....955
Casting types safely....956
What's next?....957

Standard input....957
Read from the standard input with Java Scanner....958
Read from the standard input with readln()....959

Browser and DOM API....960
Interaction with the DOM....960
What's next?....961

Use JavaScript code from Kotlin....961
Inline JavaScript....961
external modifier....962
Equality....965

Dynamic type....966

Use dependencies from npm....967

Use Kotlin code from JavaScript....968
Isolating declarations in a separate JavaScript object in plain mode....968
Package structure....968
Kotlin types in JavaScript....974

JavaScript modules 976

Browser targets 976

JavaScript libraries and Node.js files 976

@JsModule annotation 977

Kotlin/JS reflection 979

Class references 979

KType and typeOf() 980

KClass and createInstance() 980

Example 980

Typesafe HTML DSL 980

Interoperability with Swift/Objective-C 982

Importing Swift/Objective-C libraries to Kotlin 982

Using Kotlin in Swift/Objective-C 982

Mappings 984

Casting between mapped types 994

Subclassing 994

C features 995

Unsupported 995

Kotlin/Native as an Apple framework – tutorial 995

Create a Kotlin library 996

Generated framework headers 999

Garbage collection and reference counting 1002

Use code from Objective-C 1002

Use code from Swift 1003

Connect the framework to your iOS project 1003

What's next 1003

Interoperability with Swift using Swift export 1004

Enable Swift export 1004

Current limitations 1006

Mappings....1007
Cross-language inheritance....1014
Evolution of Swift export....1014

Interoperability with C....1015

Setting up your project....1015
Bindings....1015
What's next....1023

Mapping primitive data types from C – tutorial....1023

Types in C language....1024
Create a C library....1024
Create a Kotlin/Native project....1025
Inspect generated Kotlin APIs for a C library....1028
Update Kotlin code....1028
Next step....1029

Mapping struct and union types from C – tutorial....1029

Mapping struct and union C types....1029
Inspect generated Kotlin APIs for a C library....1030
Use struct and union types from Kotlin....1032
Update Kotlin code....1034
Next step....1035

Mapping function pointers from C – tutorial....1036

Mapping function pointer types from C....1036
Inspect generated Kotlin APIs for a C library....1036
Pass Kotlin function as a C function pointer....1037
Use the C function pointer from Kotlin....1037
Update Kotlin code....1038
Next step....1038

Mapping strings from C – tutorial....1039

Working with C strings....1039

Inspect generated Kotlin APIs for a C library 1040
Pass Kotlin strings to C 1041
Read C strings in Kotlin 1041
Receive C string bytes from Kotlin 1041
Update Kotlin code 1042
What's next 1043

Kotlin/Native as a dynamic library – tutorial 1043

Create a Kotlin library 1043
Generated header file 1046
Use generated headers from C 1050
Compile and run the project 1050
What's next 1051

Create an app using C interop and libcurl – tutorial 1051

Before you start 1052
Create a definition file 1053
Add interoperability to the build process 1054
Write the application code 1055
Compile and run the application 1055
What's next 1056

Definition file 1057

Create and configure a definition file 1057
Properties 1058
Generate bindings using command line 1062
What's next 1062

C, Objective-C, and Swift library import 1062

BetaStability of C and Objective-C library import 1062
Swift library import 1066

Kotlin AI skills 1068

Supported workflows 1069

Get support 1069

Kotlin for AI-powered app development 1069

Koog 1070

Model Context Protocol (MCP) Kotlin SDK 1071

Explore other AI-powered application scenarios 1071

What's next 1072

Create a Kotlin app that answers questions with Spring AI — tutorial 1072

Before you start 1072

Create the project 1073

Update the project configuration 1076

Create a controller to load and search documents 1077

Implement an AI chat endpoint 1079

What's next 1082

Kotlin for data analysis 1082

Kotlin DataFrame 1082

Kandy 1083

What's next 1084

Retrieve data from files 1084

Before you start 1085

Retrieve data 1085

Display data 1085

Inspect data structure 1086

Refine data 1087

Export data 1088

What's next 1089

Retrieve data from web sources and APIs 1089

Before you start 1089

Fetch data from an API 1090

Clean and refine data 1091

Analyze data in Kotlin Notebook 1091
What's next 1093

Connect and retrieve data from databases 1093
Before you start 1094
Connect to a database 1094
Inspect database schema 1095
Load data 1095
Process data 1096
Analyze data 1096
What's next 1097

Data visualization with Kandy 1097
Before you start 1097
Create a DataFrame 1098
Create a line chart 1099
Create a points chart 1099
Create a bar chart 1100
What's next 1101

Data visualization with Lets-Plot for Kotlin 1102
Before you start 1102
Prepare the data 1103
Create a scatter plot 1104
Create a box plot 1105
Create a 2D density plot 1106
What's next 1108

Kotlin and Java libraries for data analysis 1108
Kotlin libraries 1108
Java libraries 1110

Collections overview 1111
Collection types 1112

Constructing collections 1117
Construct from elements 1117
Create with collection builder functions 1117
Empty collections 1117
Initializer functions for lists 1118
Concrete type constructors 1118
Copy 1118
Invoke functions on other collections 1119
Iterators 1119
List iterators 1120
Mutable iterators 1121
Ordering 1122
Natural order 1123
Custom orders 1124
Check sorted order 1124
Reverse order 1125
Random order 1126
Grouping 1126
Collection transformation operations 1127
Map 1127
Zip 1128
Associate 1129
Flatten 1130
String representation 1130
Filtering collections 1131
Filter by predicate 1131
Partition 1132
Test predicates 1132
Retrieve collection parts 1133

Slice....1133
Take and drop....1133
Chunked....1134
Windowed....1135
Retrieve single elements....1135
Retrieve by position....1136
Retrieve by condition....1136
Retrieve with selector....1137
Random element....1137
Check element existence....1138
Collection operations overview....1138
Extension and member functions....1138
Common operations....1139
Write operations....1140
Aggregate operations....1140
Fold and reduce....1141
Collection write operations....1143
Adding elements....1143
Removing elements....1143
Updating elements....1144
List-specific operations....1145
Retrieve elements by index....1145
Retrieve list parts....1145
Find element positions....1145
List write operations....1147
Set-specific operations....1149
Map-specific operations....1150
Retrieve keys and values....1150
Filter....1151

Plus and minus operators 1152
Map write operations 1152
Plus and minus operators 1155
Sequences 1155
Construct 1155
Sequence operations 1157
Sequence processing example 1157
Ranges and progressions 1159
Ranges 1159
Progressions 1160
Read standard input 1161
Handle standard input safely 1162
Opt-in requirements 1163
Opt in to API 1163
Require opt-in to use API 1168
Require opt-in to extend API 1170
Opt-in requirements for pre-stable APIs 1170
Scope functions 1171
Function selection 1171
Distinctions 1172
Functions 1176
takeIf and takeUnless 1179
Time measurement 1180
Calculate duration 1180
Measure time 1184
Time sources 1186
UUIDs 1187
Generate UUIDs 1187

Parse UUIDs 1188
Convert UUIDs to strings 1189
Compare UUIDs 1190
Work with binary representations 1190
Serialize UUIDs 1192
Use UUIDs with Java APIs 1192

Coroutines guide 1193
Table of contents 1193
Additional references 1193

Coroutines basics 1194
Suspending functions 1194
Add the kotlinx.coroutines library to your project 1195
Create your first coroutines 1196
Coroutine scope and structured concurrency 1198
Coroutine builder functions 1200
Coroutine dispatchers 1202
Comparing coroutines and JVM threads 1204
What's next 1205

Cancellation and timeouts 1205
Cancel coroutines 1206
Make coroutines react to cancellation 1208
Handle values safely when canceling coroutines 1212
Timeout 1214

Flows 1215
Cold flows 1216
Hot flows 1229

Flow operators 1248
Intermediate operators 1249
Terminal operators 1261

Coroutines and channels - tutorial....1265

Before you start....1265

Blocking requests....1267

Callbacks....1270

Suspending functions....1274

Coroutines....1276

Concurrency....1278

Structured concurrency....1282

Showing progress....1286

Channels....1289

Testing coroutines....1293

What's next....1297

Composing suspending functions....1297

Sequential by default....1297

Concurrent using async....1298

Lazily started async....1299

Async-style functions....1300

Structured concurrency with async....1302

Coroutine context and dispatchers....1303

Dispatchers and threads....1303

Unconfined vs confined dispatcher....1304

Debugging coroutines and threads....1305

Jumping between threads....1307

Job in the context....1308

Children of a coroutine....1308

Parental responsibilities....1309

Naming coroutines for debugging....1310

Combining context elements....1311

Coroutine scope....1311

Channels **1314**
Channel basics 1314
Closing and iteration over channels 1314
Building channel producers 1315
Pipelines 1315
Prime numbers with pipeline 1316
Fan-out 1318
Fan-in 1319
Buffered channels 1320
Channels are fair 1320
Ticker channels 1321

Coroutine exceptions handling **1322**
Exception propagation 1323
CoroutineExceptionHandler 1323
Cancellation and exceptions 1324
Exceptions aggregation 1326
Supervision 1328

Shared mutable state and concurrency **1331**
The problem 1331
Volatiles are of no help 1332
Thread-safe data structures 1333
Thread confinement fine-grained 1334
Thread confinement coarse-grained 1335
Mutual exclusion 1336

Select expression (experimental) **1337**
Selecting from channels 1337
Selecting on close 1339
Selecting to send 1341
Selecting deferred values 1342

Switch over a channel of deferred values....1343

Debug coroutines....1346

Enable debug mode....1346

Stack trace recovery....1347

ExperimentalThe debug agent....1351

What's next....1359

Debug coroutines using IntelliJ IDEA – tutorial....1359

Create coroutines....1359

Debug coroutines....1360

Debug Kotlin Flow using IntelliJ IDEA – tutorial....1363

Create a Kotlin flow....1363

Debug the coroutine....1365

Add a concurrently running coroutine....1368

Debug a Kotlin flow with two coroutines....1368

Serialization....1369

Kotlin serialization libraries....1369

Supported serialization formats....1370

Supported serialization types....1370

What's next....1370

Get started with Kotlin serialization....1371

Add plugins and dependencies....1371

Serialize objects to JSON....1374

Deserialize objects from JSON....1375

What's next....1376

Serialize built-in types....1376

Basic types....1376

Standard library types....1379

What's next....1382

Serialize classes....1383

The @Serializable annotation....1383
Optional properties....1385
Customize class serialization....1389
What's next....1391
JSON serialization overview....1391
What's next....1392
Customize the Json instance....1392
Customize JSON structure....1393
Customize JSON deserialization....1399
Customize name mapping between JSON and Kotlin....1403
What's next....1405
JSON elements....1405
Parse strings to JSON elements....1406
Access the contents of JSON elements....1406
Create JSON elements....1407
Decode Json elements....1411
What's next....1411
Transform JSON structure....1412
Modify JSON structure....1412
Select the appropriate polymorphic class based on the JSON content....1415
ExperimentalAdd custom behavior to the default serializer....1417
ExperimentalImplement custom serialization logic in JSON....1418
What's next....1421
ExperimentalSerialize JSON with I/O sources....1421
Serialize JSON to JVM output streams....1422
Deserialize JSON from JVM input streams....1422
JSON serialization with kotlinx-io and Okio....1423
What's next....1426
Serialize polymorphic classes....1426

Serialize closed polymorphic classes 1427
Define custom serial names for subclasses 1429
Base class properties with backing fields 1430
Serialize objects in polymorphic class hierarchies 1430
Serialize open polymorphic classes 1431
What's next 1445

Create and use serializers 1445
Obtain serializers 1446
Create custom serializers 1447
Apply serializers 1463
What's next 1469

Kotlin Metadata JVM library 1469
Add the library to your project 1469
Read and parse metadata 1470
Modify metadata 1476
Create metadata from scratch 1477
What's next? 1480

Overview 1480
How Lincheck works 1481
Explore Lincheck 1481
Learn more 1482

Getting started with Lincheck 1482
Create a project 1482
Add dependencies 1482
Write your first test 1483
Write a test for a data structure 1484
What's next 1485

Testing arbitrary code 1486
Example: test ConcurrentHashMap functions 1487

What's next 1488
See also 1488

How to test data structures 1488
The testing process 1489
Example: test an implementation of a Treiber stack structure 1490
What's next 1492
See also 1493

Testing strategies 1493
Model checking 1493
Stress testing 1493
Choose a strategy 1494
What's next 1494
See also 1494

Configuring a testing strategy 1495
How to enable options 1495
Scenario minimization 1495
Scenario generation 1495
Stalled execution detection 1497
Thread switching in loops 1497
Verification 1498
Progress guarantees 1498
Library analysis 1498
What's next 1499
See also 1499

Argument generation constraints 1500
Generator types 1500
What's next 1502
See also 1502

Operation execution options 1502

Single-thread operation groups....1502
Single-use operations....1503
Blocking operations....1503
Cancelable operations....1504
See also....1505

Progress guarantees....1506
Example: test ConcurrentHashMap for obstruction-freedom....1506
Example: test ConcurrentSkipListMap for obstruction-freedom....1507
See also....1508

Results verification and validation....1508
Verification....1508
Validation....1513
What's next....1513

Model checking....1513
Deterministic exploration....1514
Bounded exploration....1516
Known limitations and workarounds....1516
See also....1521

Lincheck in Kotlin Multiplatform projects....1521
Set up a project....1521
Create shared classes....1523
Write shared tests....1524
Write JVM-specific tests....1524
Run the tests....1525
Support for platforms other than the JVM....1530
See also....1530

API references....1530

Gradle....1532
What's next?....1532

Get started with Gradle and Kotlin/JVM 1533

Create a project 1533

Explore the build script 1535

Run the application 1536

What's next? 1538

Configure a Gradle project 1538

Apply the plugin 1538

Targeting the JVM 1541

Targeting multiple platforms 1549

Targeting Android 1549

Targeting the web 1549

Triggering configuration actions with the KotlinBasePlugin interface 1551

Configure dependencies 1552

Declare repositories 1560

ExperimentalRegister generated sources 1561

What's next? 1561

Gradle best practices 1561

Organize 1562

Optimize 1562

Compiler options in the Kotlin Gradle plugin 1564

How to define options 1564

Target the JVM 1573

Target JavaScript 1574

All Kotlin compilation tasks 1574

All compiler options 1575

What's next? 1579

Compilation and caches in the Kotlin Gradle plugin 1580

Incremental compilation 1580

Gradle build cache support 1581

Gradle configuration cache support 1581
The Kotlin daemon and how to use it with Gradle 1581
Rolling back to the previous compiler 1584
Trying the latest language version 1584
Build reports 1585
What's next? 1587

ExperimentalBinary compatibility validation in the Kotlin Gradle plugin 1587
How to enable 1587
Check for binary compatibility issues 1588
Update reference ABI dump 1588
Configure filters 1588
Prevent inferred changes for unsupported targets 1590
Include publications from the maven-publish plugin 1590

Support for Gradle plugin variants 1591
Troubleshooting 1592
What's next? 1594

Maven 1594
What's next? 1594

Configure a Maven project 1595
Automatic configuration 1595
Manual configuration 1597
Configure Java modules (JPMS) 1602
What's next? 1602

Set repositories and dependencies in a Maven project 1602
Declare repositories 1603
Set dependencies 1603
Manage dependencies with a BOM 1605
What's next? 1606

Configure Kotlin compiler for your Maven project 1606

Specify compiler options 1606
Choose execution strategy....1607
Enable incremental compilation....1608
What's next?....1608

Package your Maven application 1608

Create JAR files 1608
Create self-contained JAR files 1609

BetaBuild tools API 1609

Integration with Gradle....1610
Integration with Maven....1611

Kotlin daemon 1611

Kotlin daemon configuration....1611

K2 compiler migration guide 1613

Performance improvements 1614
Language feature improvements....1614
How to enable the Kotlin K2 compiler....1623
Support in IDEs....1624
Try the Kotlin K2 compiler in the Kotlin Playground....1624
How to roll back to the previous compiler....1624
Changes....1624
Compatibility with Kotlin releases....1639
Compatibility with Kotlin libraries 1639
Compiler plugins support 1639
Share your feedback on the new K2 compiler....1640

Kotlin command-line compiler....1640

Install the compiler....1641
Create and run an application 1641
Compile a library....1642
Run the REPL....1642

Run scripts 1642
What's next? 1643

Create a console app – tutorial 1643
Create a project 1643
Create an application 1645
Run the application 1646
What's next? 1648

Kotlin compiler options 1648
Compiler options 1648
Schema for compiler options 1649
Common options 1649
Kotlin/JVM compiler options 1655
Kotlin/JS compiler options 1657
Kotlin/Native compiler options 1659

Compiler execution strategy 1662
Configure in Gradle 1662
Configure in Maven 1664

Compiler plugins 1664

All-open compiler plugin 1665
Gradle 1666
Maven 1667
Spring support 1667
Command-line compiler 1668

kapt compiler plugin 1669
Set up the plugin 1669
Configure annotation processors 1672
Optimize kapt builds 1674
Analyze performance 1675
Generate Kotlin sources 1676

Compiler options....1677
What's next....1682
Lombok compiler plugin....1682
Supported annotations....1682
Gradle....1683
Maven....1684
Using with kapt....1684
Command-line compiler....1685
No-arg compiler plugin....1685
In your Kotlin file....1686
Gradle....1686
Maven....1687
JPA support....1687
Command-line compiler....1688
ExperimentalPower-assert compiler plugin....1688
Apply the plugin....1688
Use the Power-assert plugin....1691
Add support for Power-assert to your library....1701
What's next....1703
SAM-with-receiver compiler plugin....1703
Gradle....1704
Maven....1705
Command-line compiler....1705
Custom compiler plugins....1705
The Kotlin compiler and compiler plugins....1706
Kotlin compiler plugin template....1707
Frontend plugin API....1707
Backend plugin API....1708
Test your plugin....1709

Get help 1709

Compose compiler migration guide 1709

Migrating a Jetpack Compose project 1710

What's next 1711

Compose compiler options DSL 1711

Purpose and use of feature flags 1711

Kotlin Symbol Processing API 1712

Overview 1712

How KSP looks at source files 1713

SymbolProcessorProvider: the entry point 1714

Resources 1714

Supported libraries 1715

Getting started with KSP 1716

Add a KSP-based processor to your project 1716

Create your own processor 1717

What's next? 1726

Migrate from kapt to KSP 1726

Add the KSP plugin to your project 1727

Update your processor 1727

Remove the kapt plugin 1728

What's next? 1728

KSP examples 1729

Get all member functions 1729

Check whether a class or function is local 1729

Find the actual class or interface declaration that the type alias points to 1729

Collect suppressed names in a file annotation 1729

How KSP models Kotlin code 1729

Type and resolution 1730

Java annotation processing to KSP reference 1731

Program elements 1731

Types 1732

Misc 1733

Details 1734

Incremental processing 1741

Dirty files 1741

Implementation 1742

Example processor 1743

Reporting bugs 1744

Multiple-round processing 1744

Deferring symbols to the next round 1745

Validating symbols 1745

Accessing files and symbols 1745

Processor instantiation 1746

Error and exception handling 1746

KSP with Kotlin Multiplatform 1746

Add KSP to a multiplatform project 1746

Find KSP configuration names 1749

Compilation and processing 1749

Running KSP from the command line 1749

Classpath 1750

Main class 1750

Options 1751

KSP FAQ 1752

Why KSP? 1752

Why is KSP faster than kapt? 1752

Is KSP Kotlin-specific? 1753

How to upgrade KSP? 1753

Can I use a newer KSP implementation with an older Kotlin compiler? 1753
How often do you update KSP? 1753
Besides Kotlin, are there other version requirements for libraries? 1753
What is KSP's future roadmap? 1754

Kotlin roadmap 1754
Key priorities 1754
Kotlin roadmap by subsystem 1754
What's changed since February 2026 1756

Kotlin language features and proposals 1757

Kotlin evolution principles 1767
Principles of pragmatic evolution 1767
Incompatible changes 1767
Decision making 1768
Language feature delivery 1769
Libraries 1770
Compiler options 1770
Compatibility tools 1771

Stability of Kotlin components 1772
Stability levels explained 1772
GitHub badges for Kotlin components 1773
Stability of subcomponents 1773
Current stability of Kotlin components 1773
Language features and design proposals 1776

Kotlin release process 1776
Upcoming Kotlin releases 1777
Update to a new Kotlin version 1777
IDE support 1778
Standard library security support 1778
Release history 1779

What's new in Kotlin 2.3.20 1788

Update to Kotlin 2.3.20 1788

StableNew features 1788

New features 1789

Language 1793

Standard library 1796

Kotlin compiler plugins 1797

Kotlin/JVM 1798

Kotlin/Native 1799

Kotlin/Wasm 1801

Kotlin/JS 1802

Gradle 1804

Maven 1805

Build tools API 1806

Breaking changes and deprecations 1808

Documentation updates 1809

What's new in Kotlin 2.3.0 1809

IDE support 1810

Language 1810

Kotlin/JVM: Support for Java 25 1815

Kotlin/Native 1815

Kotlin/Wasm 1817

Kotlin/JS 1818

Gradle 1822

Standard library 1822

Compose compiler: Stack traces for minified Android applications 1825

Breaking changes and deprecations 1826

Documentation updates 1826

How to update to Kotlin 2.3.0 1827

Compatibility guide for Kotlin 2.3.x 1827

Basic terms....1827
Language....1828
Standard library....1831
Tools....1834
Build tool removal....1848

What's new in Kotlin 2.2.20....1848

IDE support....1849
Language....1849
ExperimentalKotlin/JVM: Support invokedynamic with when expressions....1856
Kotlin Multiplatform....1856
Kotlin/Native....1861
Kotlin/Wasm....1865
Kotlin/JS....1868
Gradle....1870
Maven: Support for the Kotlin daemon in the kotlin-maven-plugin....1872
New common schema for Kotlin compiler options....1872
Standard library....1872
Compose compiler....1875
Breaking changes and deprecations....1875
Documentation updates....1875
How to update to Kotlin 2.2.20....1876

What's new in Kotlin 2.2.0....1876

IDE support....1877
Language....1877
ExperimentalKotlin compiler: Unified management of compiler warnings....1883
Kotlin/JVM....1885
Kotlin/Native....1888
Kotlin/Wasm....1890
Kotlin/JS....1891
Gradle....1894

ExperimentalNew experimental build tools API 1897
Standard library 1898
Compose compiler 1899
Breaking changes and deprecations 1900
Documentation updates 1902
How to update to Kotlin 2.2.0 1903

Compatibility guide for Kotlin 2.2.x 1903

Basic terms 1903
Language 1904
Standard library 1909
Tools 1910

What's new in Kotlin 2.1.20 1918

IDE support 1919
Kotlin K2 compiler 1919
ExperimentalKotlin Multiplatform: new DSL to replace Gradle's Application plugin 1920
Kotlin/Native 1921
Kotlin/Wasm 1921
Gradle 1924
Standard library 1925
Compose compiler 1928
Breaking changes and deprecations 1929
Documentation updates 1929
How to update to Kotlin 2.1.20 1930

What's new in Kotlin 2.1.0 1930

IDE support 1931
Language 1931
Kotlin K2 compiler 1936
Kotlin/JVM 1939
Kotlin Multiplatform 1940
Kotlin/Native 1945

Kotlin/Wasm 1946
Kotlin/JS 1954
Gradle improvements 1955
Compose compiler updates 1958
Standard library 1959
Documentation updates 1962
Compatibility guide for Kotlin 2.1.0 1963
Install Kotlin 2.1.0 1963

Compatibility guide for Kotlin 2.1.x 1964
Basic terms 1964
Language 1964
Standard library 1971
Tools 1973

What's new in Kotlin 2.0.20 1975
IDE support 1976
Language 1976
Kotlin Multiplatform 1978
Kotlin/Native 1981
Kotlin/Wasm 1982
Kotlin/JS 1983
Gradle 1984
Compose compiler 1986
Standard library 1988
Documentation updates 1991
Install Kotlin 2.0.20 1992

What's new in Kotlin 2.0.0 1992
IDE support 1992
Kotlin K2 compiler 1993
Kotlin/JVM 2004
Kotlin/Native 2005

Kotlin/Wasm 2006
Kotlin/JS 2008
Gradle improvements 2013
Standard library 2023
Install Kotlin 2.0.0 2025

Compatibility guide for Kotlin 2.0.x 2025
Basic terms 2026
Language 2026
Tools 2039

What's new in Kotlin 1.9.20 2042
IDE support 2042
New Kotlin K2 compiler updates 2043
Kotlin/JVM 2044
Kotlin/Native 2044
Kotlin Multiplatform 2049
Kotlin/Wasm 2057
Gradle 2059
Standard library 2060
Documentation updates 2062
Install Kotlin 1.9.20 2063

What's new in Kotlin 1.9.0 2063
IDE support 2064
New Kotlin K2 compiler updates 2064
Language 2066
Kotlin/JVM 2068
Kotlin/Native 2068
Kotlin Multiplatform 2072
Kotlin/Wasm 2073
Kotlin/JS 2075
Gradle 2077

Standard library 2080
Documentation updates....2087
Install Kotlin 1.9.0 2087
Compatibility guide for Kotlin 1.9.0 2087

Compatibility guide for Kotlin 1.9.x....2088

Basic terms 2088
Language 2088
Standard library 2099
Tools....2101

What's new in Kotlin 1.8.20 2102

IDE support 2102
New Kotlin K2 compiler updates....2103
Language....2104
New Kotlin/Wasm target 2109
Kotlin/JVM 2111
Kotlin/Native....2112
Kotlin Multiplatform 2115
Kotlin/JavaScript 2119
Gradle 2120
Standard library....2123
Serialization updates 2126
Documentation updates....2127
Install Kotlin 1.8.20 2127

What's new in Kotlin 1.8.0 2128

IDE support....2128
Kotlin/JVM....2128
Kotlin/Native....2129
Kotlin Multiplatform: A new Android source set layout 2131
Kotlin/JS....2134
Gradle 2135

Standard library....2139
Documentation updates....2143
Install Kotlin 1.8.0....2143
Compatibility guide for Kotlin 1.8.0....2143
Compatibility guide for Kotlin 1.8.x....2143
Basic terms....2144
Language....2144
Standard library....2157
Tools....2158
What's new in Kotlin 1.7.20....2160
Support for Kotlin K2 compiler plugins....2161
Language....2162
Kotlin/JVM....2166
Kotlin/Native....2168
Kotlin/JS....2170
Gradle....2170
Standard library....2171
Documentation updates....2173
Install Kotlin 1.7.20....2174
What's new in Kotlin 1.7.0....2174
New Kotlin K2 compiler for the JVM in Alpha....2175
Language....2176
Kotlin/JVM....2178
Kotlin/Native....2179
Kotlin/JS....2181
Standard library....2182
Gradle....2187
Migrating to Kotlin 1.7.0....2193
Compatibility guide for Kotlin 1.7.20....2193

Basic terms 2194
Language 2194

Compatibility guide for Kotlin 1.7.0 2195
Basic terms 2195
Language 2196
Standard library 2201
Tools 2203

What's new in Kotlin 1.6.20 2206
Language 2206
Kotlin/JVM 2208
Kotlin/Native 2210
Kotlin Multiplatform 2215
Kotlin/JS 2218
Security 2219
Gradle 2222

What's new in Kotlin 1.6.0 2223
Language 2224
Supporting previous API versions for a longer period 2226
Kotlin/JVM 2227
Kotlin/Native 2228
Kotlin/JS 2231
Kotlin Gradle plugin 2232
Standard library 2232
Tools 2236
Coroutines 1.6.0-RC 2236
Migrating to Kotlin 1.6.0 2237

Compatibility guide for Kotlin 1.6.x 2237
Basic terms 2237
Language 2238

Standard library 2244
Tools 2248

What's new in Kotlin 1.5.30 2250

Language features 2250
Kotlin/JVM 2255
Kotlin/Native 2256
Kotlin Multiplatform 2259
Kotlin/JS 2262
Gradle 2262
Standard library 2265
Serialization 1.3.0-RC 2268

What's new in Kotlin 1.5.20 2269

Kotlin/JVM 2269
Kotlin/Native 2270
Kotlin/JS 2272
Gradle 2272
Standard library 2273

What's new in Kotlin 1.5.0 2273

Language features 2274
Kotlin/JVM 2275
Kotlin/Native 2278
Kotlin/JS 2279
Kotlin Multiplatform 2279
Standard library 2279
kotlin-test library 2284
kotlinx libraries 2287
Migrating to Kotlin 1.5.0 2288

Compatibility guide for Kotlin 1.5.x 2288

Basic terms 2288

Language and stdlib 2289
Tools 2298

What's new in Kotlin 1.4.30 2298

Language features 2299
Kotlin/JVM 2302
Kotlin/Native 2302
Kotlin/JS 2303
Gradle project improvements 2303
Standard library 2303
Serialization updates 2305

What's new in Kotlin 1.4.20 2306

Kotlin/JVM 2306
Kotlin/JS 2307
Kotlin/Native 2309
Kotlin Multiplatform 2311
Standard library 2311
Kotlin Android Extensions 2312

What's new in Kotlin 1.4.0 2313

Language features and improvements 2313
New tools in the IDE 2318
New compiler 2320
Kotlin/JVM 2323
Kotlin/JS 2325
Kotlin/Native 2326
Kotlin Multiplatform 2329
Gradle project improvements 2332
Standard library 2334
Stable JSON serialization 2341
Scripting and REPL 2341
Migrating to Kotlin 1.4.0 2342

Compatibility guide for Kotlin 1.4.x 2343

Basic terms 2343

Language and stdlib 2343

Tools 2357

What's new in Kotlin 1.3 2358

Coroutines release 2358

Kotlin/Native 2358

Multiplatform projects 2358

Contracts 2359

Capturing when subject in a variable 2360

@JvmStatic and @JvmField in companions of interfaces 2361

Nested declarations in annotation classes 2361

Parameterless main 2362

Functions with big arity 2362

Progressive mode 2362

Inline classes 2363

Unsigned integers 2363

@JvmDefault 2364

Standard library 2364

Tooling 2366

Compatibility guide for Kotlin 1.3.x 2367

Basic terms 2367

Incompatible changes 2368

What's new in Kotlin 1.2 2377

Table of contents 2377

Multiplatform projects (experimental) 2378

Other language features 2379

Standard library 2382

JVM backend 2384

JavaScript backend 2385
Tools 2385

What's new in Kotlin 1.1 2385
Table of contents 2386
JavaScript 2386
Coroutines (experimental) 2386
Other language features 2387
Standard library 2392
JVM Backend 2396
JavaScript backend 2397

Participate in the Kotlin Early Access Preview 2398
How the EAP can help you be more productive with Kotlin 2399
Build details 2399

Configure your build for EAP 2399
Configure in Gradle 2400
Configure in Maven 2401
If you run into any problems 2402

Learning materials overview 2402

Kotlin hands-on 2403
Creating a web app with Spring Boot and Kotlin 2403
Building web applications with React and Kotlin/JS 2403
Building web applications with Spring Boot and Kotlin 2403
Creating HTTP APIs with Ktor 2403
Creating a WebSocket chat with Ktor 2404
Creating an interactive website with Ktor 2404
Introduction to Kotlin coroutines and channels 2404
Introduction to Kotlin/Native 2404
Kotlin Multiplatform: networking and data storage 2404
Targeting iOS and Android with Kotlin Multiplatform 2404

Kotlin tips **2404**

null + null in Kotlin 2404

Deduplicating collection items 2405

The suspend and inline mystery 2405

Unshadowing declarations with their fully qualified name 2405

Return and throw with the Elvis operator 2405

Destructuring declarations 2405

Operator functions with nullable values 2405

Timing code 2406

Improving loops 2406

Strings 2406

Doing more with the Elvis operator 2406

Kotlin collections 2406

What's next? 2406

Kotlin Koans **2406**

Kotlin books **2407**

Advent of Code puzzles in idiomatic Kotlin **2409**

Get ready for Advent of Code 2410

Advent of Code 2022 2410

Advent of Code 2021 2412

Advent of Code 2020 2413

What's next? 2415

Learning Kotlin with JetBrains Academy plugin **2415**

Teaching Kotlin with JetBrains Academy plugin **2415**

FAQ **2416**

What is Kotlin? 2416

What is the current version of Kotlin? 2416

Is Kotlin free? 2416

Is Kotlin an object-oriented language or a functional one? 2416

What advantages does Kotlin give me over the Java programming language? 2416
Is Kotlin compatible with the Java programming language? 2416
What can I use Kotlin for? 2416
Can I use Kotlin for Android development? 2417
Can I use Kotlin for backend development? 2417
Can I use Kotlin for web development? 2417
Can I use Kotlin for desktop development? 2417
Can I use Kotlin for native development? 2417
What IDEs support Kotlin? 2417
What build tools support Kotlin? 2417
What does Kotlin compile down to? 2418
Which versions of JVM does Kotlin target? 2418
Is Kotlin hard? 2418
What companies are using Kotlin? 2418
Who develops Kotlin? 2418
Where can I learn more about Kotlin? 2418
Are there any books on Kotlin? 2418
Are any online courses available for Kotlin? 2418
Does Kotlin have a community? 2419
Are there Kotlin events? 2419
Is there a Kotlin conference? 2419
Is Kotlin on social media? 2419
Any other online Kotlin resources? 2419
Where can I get an HD Kotlin logo? 2419

Google Summer of Code with Kotlin 2419

GSoC 2026: project ideas 2420
Past GSoC projects with Kotlin 2420

Google Summer of Code with Kotlin 2026 2420

Kotlin contributor guidelines for Google Summer of Code (GSoC) 2420
Project ideas 2421

Google Summer of Code with Kotlin 2025 2423

Kotlin contributor guidelines for Google Summer of Code (GSoC) 2423

Project ideas 2424

Google Summer of Code with Kotlin 2024 2431

Kotlin contributor guidelines for Google Summer of Code (GSoC) 2431

Project ideas 2432

Google Summer of Code with Kotlin 2023 2435

Project ideas 2435

Contribution 2439

Participate in Early Access Preview 2439

Contribute to the compiler and standard library 2440

Contribute to the Kotlin IDE plugin 2440

Contribute to other Kotlin libraries and tools 2440

Contribute to the documentation 2440

Translate documentation to other languages 2440

Hold events and presentations 2440

KUG guidelines 2441

How to run a KUG? 2441

Support for KUGs from JetBrains 2441

Support from JetBrains for other tech communities 2442

Kotlin Night guidelines 2442

Event guidelines 2442

Event requirements 2442

JetBrains support 2443

Code of conduct and guidelines for Kotlin Slack 2443

How to behave 2444

How not to behave 2444

How to report issues 2444

Basic usage guidelines 2444

Moderators....2445
Copyright....2447
Kotlin brand assets....2447
Kotlin logo....2447
Kotlin ecosystem logos....2448
Kotlin mascot....2449
Kotlin User Group brand assets....2450
Kotlin Night brand assets....2452
Kotlin documentation as PDF....2456
Security....2456

Kotlin Docs

Get started with Kotlin

Kotlin is a modern language that's concise, multiplatform, and interoperable with Java and other languages.

New to Kotlin? Take our tour to learn the fundamentals directly in your browser.

Start the Kotlin tour

Install Kotlin

Kotlin is included in each IntelliJ IDEA and Android Studio release. Download and install one of these IDEs to start using Kotlin.

Choose your Kotlin use case

Console

Here you'll learn how to develop a console application and create unit tests with Kotlin.

1. Create a basic JVM application with the IntelliJ IDEA project wizard.
2. Write your first unit test.

Backend

Here you'll learn how to develop a backend application with Kotlin server-side.

- **Introduce Kotlin to your Java project:**
  - Configure a Java project to work with Kotlin
  - Add Kotlin tests to your Java Maven project
- **Create a backend app from scratch with Kotlin:**
  - Create a RESTful web service with Spring Boot
  - Create HTTP APIs with Ktor

Cross-platform

Here you'll learn how to develop a cross-platform application using Kotlin Multiplatform.

1. Set up your environment for cross-platform development.
2. **Create your first application for iOS and Android:**
   - Create a cross-platform application from scratch and:
     - Share business logic while keeping the UI native
     - Share business logic and UI
   - Make your existing Android application work on iOS
   - Create a cross-platform application using Ktor and SQLdelight
3. **Explore** sample projects.

Android

To start using Kotlin for Android development, read Google's recommendation for getting started with Kotlin on Android.

Data analysis

From building data pipelines to productionizing machine learning models, Kotlin is a great choice for working with data and getting the most out of it.

1. **Explore and experiment with your data:**
   - DataFrame – a library for data analysis and manipulation.
   - Kandy – a plotting tool for data visualization.
2. **Follow Kotlin for Data Analysis on Twitter:** KotlinForData.

Get support

If you encounter any difficulties or problems, ask for help in Slack: get an invite or report an issue in our issue tracker.

If anything is missing or seems confusing on this page, please share your feedback.

Welcome to our tour of Kotlin!

These tours can be completed entirely within your browser. There is no installation required.

Quickly learn the essentials of the Kotlin programming language through our tours.

Beginner

Grasp the fundamentals.

Start

Intermediate

Take your understanding of Kotlin to the next level.

Start

Hello world

Here is a simple program that prints "Hello, world!":

```
fun main() {
    println("Hello, world!")
    // Hello, world!
}
```

In Kotlin:

- `fun` is used to declare a function
- The `main()` function is where your program starts from
- The body of a function is written within curly braces `{}`
- `println()` and `print()` functions print their arguments to standard output

A function is a set of instructions that performs a specific task. Once you create a function, you can use it whenever you need to perform that task, without having to write the instructions all over again. Functions are discussed in more detail in a couple of chapters. Until then, all examples use the `main()` function.

Variables

All programs need to be able to store data, and variables help you to do just that. In Kotlin, you can declare:

- Read-only variables with `val`
- Mutable variables with `var`

> You can't change a read-only variable once you have given it a value.

To assign a value, use the assignment operator `=`.

For example:

```
fun main() {
    val popcorn = 5    // There are 5 boxes of popcorn
    val hotdog = 7     // There are 7 hotdogs
    var customers = 10 // There are 10 customers in the queue

    // Some customers leave the queue
    customers = 8
    println(customers)
    // 8
}
```

> Variables can be declared outside the `main()` function at the beginning of your program. Variables declared in this way are said to be declared at **top level**.

As `customers` is a mutable variable, its value can be reassigned after declaration.

> We recommend declaring all variables as read-only (`val`) by default. Only use mutable variables (`var`) if you really need to. That way, you're less likely to accidentally change something that wasn't meant to change.

String templates

It's useful to know how to print the contents of variables to standard output. You can do this with **string templates**. You can use template expressions to access data stored in variables and other objects, and convert them into strings. A string value is a sequence of characters in double quotes `"`. Template expressions always start with a dollar sign `$`.

To evaluate a piece of code in a template expression, place the code within curly braces `{}` after the dollar sign `$`.

For example:

```
fun main() {
    val customers = 10
    println("There are $customers customers")
    // There are 10 customers

    println("There are ${customers + 1} customers")
    // There are 11 customers
}
```

For more information, see String templates.

You will notice that there aren't any types declared for variables. Kotlin has inferred the type itself: `Int`. This tour explains the different Kotlin basic types and how to declare them in the next chapter.

Practice

Exercise

Complete the code to make the program print `"Mary is 20 years old"` to standard output:

```
fun main() {
    val name = "Mary"
    val age = 20
    // Write your code here
}
```

```
fun main() {
    val name = "Mary"
    val age = 20
    println("$name is $age years old")
}
```

- Next step

Basic types

Every variable and data structure in Kotlin has a type. Types are important because they tell the compiler what you are allowed to do with that variable or data structure. In other words, what functions and properties it has.

In the last chapter, Kotlin was able to tell in the previous example that `customers` has type `Int`. Kotlin's ability to **infer** the type is called **type inference**. `customers` is assigned an integer value. From this, Kotlin infers that `customers` has a numerical type `Int`. As a result, the compiler knows that you can perform arithmetic operations with `customers`:

```
fun main() {
    var customers = 10

    // Some customers leave the queue
    customers = 8

    customers = customers + 3 // Example of addition: 11
    customers += 7            // Example of addition: 18
    customers -= 3            // Example of subtraction: 15
    customers *= 2            // Example of multiplication: 30
    customers /= 3            // Example of division: 10

    println(customers) // 10
}
```

`+=`, `-=`, `*=`, `/=`, and `%=` are augmented assignment operators. For more information, see Augmented assignments.

In total, Kotlin has the following basic types:

Category	Basic types	Example code
Integers	`Byte`, `Short`, `Int`, `Long`	`val year: Int = 2020` `val amount: Long = 350_000_000`
Unsigned integers	`UByte`, `UShort`, `UInt`, `ULong`	`val score: UInt = 100u`
Floating-point numbers	`Float`, `Double`	`val currentTemp: Float = 24.5f` `val price: Double = 19.99`
Booleans	`Boolean`	`val isEnabled: Boolean = true`
Characters	`Char`	`val separator: Char = ','`
Strings	`String`	`val message: String = "Hello, world!"`

For more information on basic types and their properties, see Types overview.

With this knowledge, you can declare variables and initialize them later. Kotlin can manage this as long as variables are initialized before the first read.

To declare a variable without initializing it, specify its type with `:`. For example:

```
fun main() {
    // Variable declared without initialization
    val d: Int
    // Variable initialized
    d = 3

    // Variable explicitly typed and initialized
    val e: String = "hello"

    // Variables can be read because they have been initialized
    println(d) // 3
    println(e) // hello
}
```

If you don't initialize a variable before it is read, you see an error:

```
fun main() {
    // Variable declared without initialization
    val d: Int

    // Triggers an error
    println(d)
    // Variable 'd' must be initialized
}
```

Now that you know how to declare basic types, it's time to learn about collections.

Practice

Exercise

Explicitly declare the correct type for each variable:

```
fun main() {
    val a: Int = 1000
    val b = "log message"
    val c = 3.14
    val d = 100_000_000_000_000
    val e = false
    val f = '\n'
}
```

```
fun main() {
    val a: Int = 1000
    val b: String = "log message"
    val c: Double = 3.14
    val d: Long = 100_000_000_000_000
    val e: Boolean = false
    val f: Char = '\n'
}
```

Previous step

Next step

Collections

When programming, it is useful to be able to group data into structures for later processing. Kotlin provides collections for exactly this purpose.

Kotlin has the following collections for grouping items:

Collection type	Description
Lists	Ordered collections of items
Sets	Unique unordered collections of items
Maps	Sets of key-value pairs where keys are unique and map to only one value

Each collection type can be mutable or read only.

List

Lists store items in the order that they are added, and allow for duplicate items.

To create a read-only list (`List`), use the `listOf()` function.

To create a mutable list (`MutableList`), use the `mutableListOf()` function.

When creating lists, Kotlin can infer the type of items stored. To declare the type explicitly, add the type within angled brackets `<>` after the list declaration:

```
fun main() {
    // Read only list
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println(readOnlyShapes)
    // [triangle, square, circle]

    // Mutable list with explicit type declaration
    val shapes: MutableList<String> = mutableListOf("triangle", "square", "circle")
    println(shapes)
    // [triangle, square, circle]
}
```

To prevent unwanted modifications, you can create a read-only view of a mutable list by assigning it to a `List` :

```
val shapes: MutableList<String> = mutableListOf("triangle", "square", "circle")
    val shapesLocked: List<String> = shapes
```

This is also called **casting**.

Lists are ordered so to access an item in a list, use the indexed access operator `[]` :

```
fun main() {
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("The first item in the list is: ${readOnlyShapes[0]}")
    // The first item in the list is: triangle
}
```

To get the first or last item in a list, use `.first()` and `.last()` functions respectively:

```
fun main() {
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("The first item in the list is: ${readOnlyShapes.first()}")
    // The first item in the list is: triangle
}
```

`.first()` and `.last()` functions are examples of **extension** functions. To call an extension function on an object, write the function name after the object appended with a period `.`

Extension functions are covered in detail in the intermediate tour. For now, you only need to know how to call them.

To get the number of items in a list, use the `.count()` function:

```
fun main() {
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("This list has ${readOnlyShapes.count()} items")
    // This list has 3 items
}
```

To check that an item is in a list, use the `in` operator:

```
fun main() {
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("circle" in readOnlyShapes)
    // true
}
```

To add or remove items from a mutable list, use `.add()` and `.remove()` functions respectively:

```
fun main() {
    val shapes: MutableList<String> = mutableListOf("triangle", "square", "circle")
    // Add "pentagon" to the list
    shapes.add("pentagon")
    println(shapes)
    // [triangle, square, circle, pentagon]

    // Remove the first "pentagon" from the list
    shapes.remove("pentagon")
    println(shapes)
    // [triangle, square, circle]
}
```

Set

Whereas lists are ordered and allow duplicate items, sets are **unordered** and only store **unique** items.

To create a read-only set (`Set`), use the `setOf()` function.

To create a mutable set (`MutableSet`), use the `mutableSetOf()` function.

When creating sets, Kotlin can infer the type of items stored. To declare the type explicitly, add the type within angled brackets `<>` after the set declaration:

```
fun main() {
    // Read-only set
    val readOnlyFruit = setOf("apple", "banana", "cherry", "cherry")
    // Mutable set with explicit type declaration
    val fruit: MutableSet<String> = mutableSetOf("apple", "banana", "cherry", "cherry")

    println(readOnlyFruit)
    // [apple, banana, cherry]
}
```

You can see in the previous example that because sets only contain unique elements, the duplicate `"cherry"` item is dropped.

To prevent unwanted modifications, you can create a read-only view of a mutable set by assigning it to a `Set` :

```
val fruit: MutableSet<String> = mutableSetOf("apple", "banana", "cherry", "cherry")
    val fruitLocked: Set<String> = fruit
```

As sets are **unordered**, you can't access an item at a particular index.

To get the number of items in a set, use the `.count()` function:

```
fun main() {
    val readOnlyFruit = setOf("apple", "banana", "cherry", "cherry")
    println("This set has ${readOnlyFruit.count()} items")
    // This set has 3 items
}
```

To check that an item is in a set, use the `in` operator:

```
fun main() {
    val readOnlyFruit = setOf("apple", "banana", "cherry", "cherry")
    println("banana" in readOnlyFruit)
    // true
}
```

To add or remove items from a mutable set, use `.add()` and `.remove()` functions respectively:

```
fun main() {
    val fruit: MutableSet<String> = mutableSetOf("apple", "banana", "cherry", "cherry")
    fruit.add("dragonfruit")    // Add "dragonfruit" to the set
    println(fruit)              // [apple, banana, cherry, dragonfruit]

    fruit.remove("dragonfruit") // Remove "dragonfruit" from the set
    println(fruit)              // [apple, banana, cherry]
}
```

Map

Maps store items as key-value pairs. You access the value by referencing the key. You can imagine a map like a food menu. You can find the price (value), by finding the food (key) you want to eat. Maps are useful if you want to look up a value without using a numbered index, like in a list.

- Every key in a map must be unique so that Kotlin can understand which value you want to get.
- You can have duplicate values in a map.

To create a read-only map (`Map`), use the `mapOf()` function.

To create a mutable map (`MutableMap`), use the `mutableMapOf()` function.

When creating maps, Kotlin can infer the type of items stored. To declare the type explicitly, add the types of the keys and values within angled brackets `<>` after the map declaration. For example: `MutableMap<String, Int>` . The keys have type `String` and the values have type `Int` .

The easiest way to create maps is to use `to` between each key and its related value:

```
fun main() {
    // Read-only map
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
    println(readOnlyJuiceMenu)
    // {apple=100, kiwi=190, orange=100}

    // Mutable map with explicit type declaration
    val juiceMenu: MutableMap<String, Int> = mutableMapOf("apple" to 100, "kiwi" to 190, "orange" to
100)
    println(juiceMenu)
    // {apple=100, kiwi=190, orange=100}
}
```

To prevent unwanted modifications, you can create a read-only view of a mutable map by assigning it to a `Map` :

```
val juiceMenu: MutableMap<String, Int> = mutableMapOf("apple" to 100, "kiwi" to 190, "orange"
to 100)
    val juiceMenuLocked: Map<String, Int> = juiceMenu
```

To access a value in a map, use the indexed access operator `[]` with its key:

```
fun main() {
    // Read-only map
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
    println("The value of apple juice is: ${readOnlyJuiceMenu["apple"]}")
    // The value of apple juice is: 100
}
```

If you try to access a key-value pair with a key that doesn't exist in a map, you see a `null` value:

```
fun main() {
    // Read-only map
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
    println("The value of pineapple juice is: ${readOnlyJuiceMenu["pineapple"]}")
    // The value of pineapple juice is: null
}
```

This tour explains null values later in the Null safety chapter.

You can also use the indexed access operator `[]` to add items to a mutable map:

```
fun main() {
    val juiceMenu: MutableMap<String, Int> = mutableMapOf("apple" to 100, "kiwi" to 190, "orange" to
100)
    juiceMenu["coconut"] = 150 // Add key "coconut" with value 150 to the map
    println(juiceMenu)
    // {apple=100, kiwi=190, orange=100, coconut=150}
}
```

To remove items from a mutable map, use the `.remove()` function:

```
fun main() {
    val juiceMenu: MutableMap<String, Int> = mutableMapOf("apple" to 100, "kiwi" to 190, "orange" to
100)
    juiceMenu.remove("orange")     // Remove key "orange" from the map
    println(juiceMenu)
    // {apple=100, kiwi=190}
}
```

To get the number of items in a map, use the `.count()` function:

```
fun main() {
    // Read-only map
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
    println("This map has ${readOnlyJuiceMenu.count()} key-value pairs")
    // This map has 3 key-value pairs
}
```

To check if a specific key is already included in a map, use the `.containsKey()` function:

```
fun main() {
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
    println(readOnlyJuiceMenu.containsKey("kiwi"))
    // true
}
```

To obtain a collection of the keys or values of a map, use the `keys` and `values` properties respectively:

```
fun main() {
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
    println(readOnlyJuiceMenu.keys)
    // [apple, kiwi, orange]
    println(readOnlyJuiceMenu.values)
    // [100, 190, 100]
}
```

`keys` and `values` are examples of **properties** of an object. To access the property of an object, write the property name after the object appended with a period `.`

Properties are discussed in more detail in the Classes chapter. At this point in the tour, you only need to know how to access them.

To check that a key or value is in a map, use the `in` operator:

```
fun main() {
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
    println("orange" in readOnlyJuiceMenu.keys)
    // true

    // Alternatively, you don't need to use the keys property
    println("orange" in readOnlyJuiceMenu)
    // true

    println(200 in readOnlyJuiceMenu.values)
    // false
}
```

For more information on what you can do with collections, see Collections.

Now that you know about basic types and how to manage collections, it's time to explore the control flow that you can use in your programs.

Practice

Exercise 1

You have a list of “green” numbers and a list of “red” numbers. Complete the code to print how many numbers there are in total.

```
fun main() {
    val greenNumbers = listOf(1, 4, 23)
    val redNumbers = listOf(17, 2)
    // Write your code here
}
```

```
fun main() {
    val greenNumbers = listOf(1, 4, 23)
    val redNumbers = listOf(17, 2)
    val totalCount = greenNumbers.count() + redNumbers.count()
    println(totalCount)
}
```

Exercise 2

You have a set of protocols supported by your server. A user requests to use a particular protocol. Complete the program to check whether the requested protocol is supported or not (`isSupported` must be a Boolean value).

```
fun main() {
    val SUPPORTED = setOf("HTTP", "HTTPS", "FTP")
    val requested = "smtp"
    val isSupported = // Write your code here
    println("Support for $requested: $isSupported")
}
```

Hint

Make sure that you check the requested protocol in upper case. You can use the `.uppercase()` function to help you with this.

```
fun main() {
    val SUPPORTED = setOf("HTTP", "HTTPS", "FTP")
    val requested = "smtp"
    val isSupported = requested.uppercase() in SUPPORTED
    println("Support for $requested: $isSupported")
}
```

Exercise 3

Define a map that relates integer numbers from 1 to 3 to their corresponding spelling. Use this map to spell the given number.

```
fun main() {
    val number2word = // Write your code here
    val n = 2
    println("$n is spelled as '${<Write your code here >}'")
}
```

```
fun main() {
    val number2word = mapOf(1 to "one", 2 to "two", 3 to "three")
    val n = 2
    println("$n is spelled as '${number2word[n]}'")
}
```

Previous step

Next step

Control flow

Like other programming languages, Kotlin is capable of making decisions based on whether a piece of code is evaluated to be true. Such pieces of code are called **conditional expressions**. Kotlin is also able to create and iterate through loops.

Conditional expressions

Kotlin provides `if` and `when` for checking conditional expressions.

If you have to choose between `if` and `when`, we recommend using `when` because it:

- Makes your code easier to read.
- Makes it easier to add another branch.
- Leads to fewer mistakes in your code.

If

To use `if`, add the conditional expression within parentheses `()` and the action to take if the result is true within curly braces `{}`:

```
fun main() {
    val d: Int
    val check = true

    if (check) {
        d = 1
    } else {
        d = 2
    }

    println(d)
    // 1
}
```

There is no ternary operator `condition ? then : else` in Kotlin. Instead, `if` can be used as an expression. If there is only one line of code per action, the curly braces `{}` are optional:

```
fun main() {
    val a = 1
    val b = 2

    println(if (a > b) a else b) // Returns a value: 2
}
```

When

Use `when` when you have a conditional expression with multiple branches.

To use `when`:

- Place the value you want to evaluate within parentheses `()`.
- Place the branches within curly braces `{}`.
- Use `->` in each branch to separate each check from the action to take if the check is successful.

`when` can be used either as a statement or as an expression. A **statement** doesn't return anything but performs actions instead.

Here is an example of using `when` as a statement:

```
fun main() {
    val obj = "Hello"

    when (obj) {
        // Checks whether obj equals to "1"
        "1" -> println("One")
        // Checks whether obj equals to "Hello"
        "Hello" -> println("Greeting")
        // Default statement
        else -> println("Unknown")
    }
    // Greeting
}
```

Note that all branch conditions are checked sequentially until one of them is satisfied. So only the first suitable branch is executed.

An **expression** returns a value that can be used later in your code.

Here is an example of using `when` as an expression. The `when` expression is assigned immediately to a variable which is later used with the `println()` function:

```
fun main() {
//sampleStart
    val obj = "Hello"

    val result = when (obj) {
        // If obj equals "1", sets result to "one"
        "1" -> "One"
        // If obj equals "Hello", sets result to "Greeting"
        "Hello" -> "Greeting"
        // Sets result to "Unknown" if no previous condition is satisfied
        else -> "Unknown"
    }
    println(result)
    // Greeting
}
```

The examples of `when` that you've seen so far both had a subject: `obj`. But `when` can also be used without a subject.

This example uses a `when` expression **without** a subject to check a chain of Boolean expressions:

```
fun main() {
    val trafficLightState = "Red" // This can be "Green", "Yellow", or "Red"

    val trafficAction = when {
        trafficLightState == "Green" -> "Go"
        trafficLightState == "Yellow" -> "Slow down"
        trafficLightState == "Red" -> "Stop"
        else -> "Malfunction"
    }

    println(trafficAction)
    // Stop
}
```

However, you can have the same code but with `trafficLightState` as the subject:

```
fun main() {
    val trafficLightState = "Red" // This can be "Green", "Yellow", or "Red"

    val trafficAction = when (trafficLightState) {
        "Green" -> "Go"
        "Yellow" -> "Slow down"
        "Red" -> "Stop"
        else -> "Malfunction"
    }

    println(trafficAction)
    // Stop
}
```

Using `when` with a subject makes your code easier to read and maintain. When you use a subject with a `when` expression, it also helps Kotlin check that all possible cases are covered. Otherwise, if you don't use a subject with a `when` expression, you need to provide an else branch.

Conditional expressions practice

Exercise 1

Create a simple game where you win if throwing two dice results in the same number. Use `if` to print `You win :)` if the dice match or `You lose :(` otherwise.

> In this exercise, you import a package so that you can use the `Random.nextInt()` function to give you a random `Int`. For more information about importing packages, see Packages and imports.

Hint

Use the equality operator (`==`) to compare the dice results.

```
import kotlin.random.Random

fun main() {
    val firstResult = Random.nextInt(6)
    val secondResult = Random.nextInt(6)
    // Write your code here
}
```

```
import kotlin.random.Random

fun main() {
    val firstResult = Random.nextInt(6)
    val secondResult = Random.nextInt(6)
    if (firstResult == secondResult)
        println("You win :)")
    else
        println("You lose :(")
}
```

Exercise 2

Using a `when` expression, update the following program so that it prints the corresponding actions when you input the names of game console buttons.

Button	Action
A	Yes
B	No
X	Menu
Y	Nothing
Other	There is no such button

```
fun main() {
    val button = "A"

    println(
        // Write your code here
    )
}
```

```
fun main() {
    val button = "A"

    println(
        when (button) {
            "A" -> "Yes"
            "B" -> "No"
            "X" -> "Menu"
            "Y" -> "Nothing"
            else -> "There is no such button"
        }
    )
}
```

Ranges

Before talking about loops, it's useful to know how to construct ranges for loops to iterate over.

The most common way to create a range in Kotlin is to use the `..` operator. For example, `1..4` is equivalent to `1, 2, 3, 4`.

To declare a range that doesn't include the end value, use the `..<` operator. For example, `1..<4` is equivalent to `1, 2, 3`.

To declare a range in reverse order, use `downTo`. For example, `4 downTo 1` is equivalent to `4, 3, 2, 1`.

To declare a range that increments in a step that isn't 1, use `step` and your desired increment value. For example, `1..5 step 2` is equivalent to `1, 3, 5`.

You can also do the same with `Char` ranges:

- `'a'..'d'` is equivalent to `'a', 'b', 'c', 'd'`
- `'z' downTo 's' step 2` is equivalent to `'z', 'x', 'v', 't'`

Loops

The two most common loop structures in programming are `for` and `while`. Use `for` to iterate over a range of values and perform an action. Use `while` to continue an action until a particular condition is satisfied.

For

Using your new knowledge of ranges, you can create a `for` loop that iterates over numbers 1 to 5 and prints the number each time.

Place the iterator and range within parentheses `()` with keyword `in`. Add the action you want to complete within curly braces `{}`:

```
fun main() {
    for (number in 1..5) {
        // number is the iterator and 1..5 is the range
        print(number)
    }
    // 12345
}
```

Collections can also be iterated over by loops:

```
fun main() {
    val cakes = listOf("carrot", "cheese", "chocolate")

    for (cake in cakes) {
        println("Yummy, it's a $cake cake!")
    }
    // Yummy, it's a carrot cake!
    // Yummy, it's a cheese cake!
    // Yummy, it's a chocolate cake!
}
```

While

`while` can be used in two ways:

- To execute a code block while a conditional expression is true. (`while`)
- To execute the code block first and then check the conditional expression. (`do-while`)

In the first use case (`while`):

- Declare the conditional expression for your while loop to continue within parentheses `()`.
- Add the action you want to complete within curly braces `{}`.

The following examples use the increment operator `++` to increment the value of the `cakesEaten` variable.

```
fun main() {
    var cakesEaten = 0
    while (cakesEaten < 3) {
        println("Eat a cake")
        cakesEaten++
    }
    // Eat a cake
    // Eat a cake
    // Eat a cake
}
```

In the second use case (`do-while`):

- Declare the conditional expression for your while loop to continue within parentheses `()` .
- Define the action you want to complete within curly braces `{}` with the keyword `do` .

```
fun main() {
    var cakesEaten = 0
    var cakesBaked = 0
    while (cakesEaten < 3) {
        println("Eat a cake")
        cakesEaten++
    }
    do {
        println("Bake a cake")
        cakesBaked++
    } while (cakesBaked < cakesEaten)
    // Eat a cake
    // Eat a cake
    // Eat a cake
    // Bake a cake
    // Bake a cake
    // Bake a cake
}
```

For more information and examples of conditional expressions and loops, see Conditions and loops.

Now that you know the fundamentals of Kotlin control flow, it's time to learn how to write your own functions.

Loops practice

Exercise 1

You have a program that counts pizza slices until there's a whole pizza with 8 slices. Refactor this program in two ways:

- Use a `while` loop.
- Use a `do-while` loop.

```
fun main() {
    var pizzaSlices = 0
    // Start refactoring here
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    // End refactoring here
    println("There are $pizzaSlices slices of pizza. Hooray! We have a whole pizza! :D")
}
```

```
fun main() {
    var pizzaSlices = 0
    while ( pizzaSlices < 7 ) {
        pizzaSlices++
        println("There's only $pizzaSlices slice/s of pizza :(")
    }
    pizzaSlices++
    println("There are $pizzaSlices slices of pizza. Hooray! We have a whole pizza! :D")
}
```

```
fun main() {
    var pizzaSlices = 0
    pizzaSlices++
    do {
        println("There's only $pizzaSlices slice/s of pizza :(")
        pizzaSlices++
    } while ( pizzaSlices < 8 )
    println("There are $pizzaSlices slices of pizza. Hooray! We have a whole pizza! :D")
}
```

Exercise 2

Write a program that simulates the Fizz buzz game. Your task is to print numbers from 1 to 100 incrementally, replacing any number divisible by three with the word "fizz", and any number divisible by five with the word "buzz". Any number divisible by both 3 and 5 must be replaced with the word "fizzbuzz".

Hint 1

Use a `for` loop to count numbers and a `when` expression to decide what to print at each step.

Hint 2

Use the modulo operator (`%`) to return the remainder of a number being divided. Use the equality operator (`==`) to check if the remainder equals zero.

```
fun main() {
    // Write your code here
}
```

```
fun main() {
    for (number in 1..100) {
        println(
            when {
                number % 15 == 0 -> "fizzbuzz"
                number % 3 == 0 -> "fizz"
                number % 5 == 0 -> "buzz"
                else -> "$number"
            }
        )
    }
}
```

Exercise 3

You have a list of words. Use `for` and `if` to print only the words that start with the letter `l`.

Hint

Use the `.startsWith()` function for `String` type.

```
fun main() {
    val words = listOf("dinosaur", "limousine", "magazine", "language")
    // Write your code here
}
```

```
fun main() {
    val words = listOf("dinosaur", "limousine", "magazine", "language")
    for (w in words) {
        if (w.startsWith("l"))
            println(w)
    }
}
```

Previous step

Next step

Functions

You can declare your own functions in Kotlin using the `fun` keyword.

```
fun hello() {
    return println("Hello, world!")
}

fun main() {
    hello()
    // Hello, world!
}
```

In Kotlin:

- Function parameters are written within parentheses `()`.
- Each parameter must have a type, and multiple parameters must be separated by commas `,`.
- The return type is written after the function's parentheses `()`, separated by a colon `:`.
- The body of a function is written within curly braces `{}`.
- The `return` keyword is used to exit or return something from a function.

If a function doesn't return anything useful, the return type and `return` keyword can be omitted. Learn more about this in Functions without return.

In the following example:

- `x` and `y` are function parameters.
- `x` and `y` have type `Int`.
- The function's return type is `Int`.
- The function returns a sum of `x` and `y` when called.

```
fun sum(x: Int, y: Int): Int {
    return x + y
}

fun main() {
    println(sum(1, 2))
    // 3
}
```

We recommend in our coding conventions that you name functions starting with a lowercase letter and use camel case with no underscores.

Named arguments

For concise code, when calling your function, you don't have to include parameter names. However, including parameter names does make your code easier to read. This is called using **named arguments**. If you do include parameter names, then you can write the parameters in any order.

In the following example, string templates (`$`) are used to access the parameter values, convert them to `String` type, and then concatenate them into a string for printing.

```
fun printMessageWithPrefix(message: String, prefix: String) {
    println("[$prefix] $message")
}

fun main() {
    // Uses named arguments with swapped parameter order
    printMessageWithPrefix(prefix = "Log", message = "Hello")
    // [Log] Hello
}
```

Default parameter values

You can define default values for your function parameters. Any parameter with a default value can be omitted when calling your function. To declare a default value, use the assignment operator `=` after the type:

```
fun printMessageWithPrefix(message: String, prefix: String = "Info") {
    println("[$prefix] $message")
}

fun main() {
    // Function called with both parameters
    printMessageWithPrefix("Hello", "Log")
    // [Log] Hello

    // Function called only with message parameter
    printMessageWithPrefix("Hello")
    // [Info] Hello

    printMessageWithPrefix(prefix = "Log", message = "Hello")
    // [Log] Hello
}
```

You can skip specific parameters with default values, rather than omitting them all. However, after the first skipped parameter, you must name all subsequent parameters.

Functions without return

If your function doesn't return a useful value, then its return type is `Unit`. `Unit` is a type with only one value – `Unit`. You don't have to declare that `Unit` is returned explicitly in your function body. This means that you don't have to use the `return` keyword or declare a return type:

```
fun printMessage(message: String) {
    println(message)
    // `return Unit` or `return` is optional
}

fun main() {
    printMessage("Hello")
    // Hello
}
```

Single-expression functions

To make your code more concise, you can use single-expression functions. For example, the `sum()` function can be shortened:

```
fun sum(x: Int, y: Int): Int {
    return x + y
}

fun main() {
    println(sum(1, 2))
    // 3
}
```

You can remove the curly braces `{}` and declare the function body using the assignment operator `=`. When you use the assignment operator `=`, Kotlin uses type inference, so you can also omit the return type. The `sum()` function then becomes one line:

```
fun sum(x: Int, y: Int) = x + y

fun main() {
    println(sum(1, 2))
    // 3
}
```

However, if you want your code to be quickly understood by other developers, it's a good idea to explicitly define the return type even when using the assignment operator `=`.

If you use `{}` curly braces to declare your function body, you must declare the return type unless it is the `Unit` type.

Early returns in functions

To stop the code in your function from being processed further than a certain point, use the `return` keyword. This example uses `if` to return from a function early if the conditional expression is found to be true:

```kotlin
// A list of registered usernames
val registeredUsernames = mutableListOf("john_doe", "jane_smith")

// A list of registered emails
val registeredEmails = mutableListOf("john@example.com", "jane@example.com")

fun registerUser(username: String, email: String): String {
    // Early return if the username is already taken
    if (username in registeredUsernames) {
        return "Username already taken. Please choose a different username."
    }

    // Early return if the email is already registered
    if (email in registeredEmails) {
        return "Email already registered. Please use a different email."
    }

    // Proceed with the registration if the username and email are not taken
    registeredUsernames.add(username)
    registeredEmails.add(email)

    return "User registered successfully: $username"
}

fun main() {
    println(registerUser("john_doe", "newjohn@example.com"))
    // Username already taken. Please choose a different username.
    println(registerUser("new_user", "newuser@example.com"))
    // User registered successfully: new_user
}
```

Functions practice

Exercise 1

Write a function called `circleArea` that takes the radius of a circle in integer format as a parameter and outputs the area of that circle.

> In this exercise, you import a package so that you can access the value of π via `PI`. For more information about importing packages, see Packages and imports.

Hint

The formula for calculating the area of a circle is πr^2, where r is the radius.

```kotlin
import kotlin.math.PI

// Write your code here

fun main() {
    println(circleArea(2))
}
```

```
import kotlin.math.PI

fun circleArea(radius: Int): Double {
    return PI * radius * radius
}

fun main() {
    println(circleArea(2)) // 12.566370614359172
}
```

Exercise 2

Rewrite the `circleArea` function from the previous exercise as a single-expression function.

```
import kotlin.math.PI

// Write your code here

fun main() {
    println(circleArea(2))
}
```

```
import kotlin.math.PI

fun circleArea(radius: Int): Double = PI * radius * radius

fun main() {
    println(circleArea(2)) // 12.566370614359172
}
```

Exercise 3

You have a function that translates a time interval given in hours, minutes, and seconds into seconds. In most cases, you need to pass only one or two function parameters while the rest are equal to 0. Improve the function and the code that calls it by using default parameter values and named arguments so that the code is easier to read.

```
fun intervalInSeconds(hours: Int, minutes: Int, seconds: Int) =
    ((hours * 60) + minutes) * 60 + seconds

fun main() {
    println(intervalInSeconds(1, 20, 15))
    println(intervalInSeconds(0, 1, 25))
    println(intervalInSeconds(2, 0, 0))
    println(intervalInSeconds(0, 10, 0))
    println(intervalInSeconds(1, 0, 1))
}
```

```
fun intervalInSeconds(hours: Int = 0, minutes: Int = 0, seconds: Int = 0) =
    ((hours * 60) + minutes) * 60 + seconds

fun main() {
    println(intervalInSeconds(1, 20, 15))
    println(intervalInSeconds(minutes = 1, seconds = 25))
    println(intervalInSeconds(hours = 2))
    println(intervalInSeconds(minutes = 10))
    println(intervalInSeconds(hours = 1, seconds = 1))
}
```

Lambda expressions

Kotlin allows you to write even more concise code for functions by using lambda expressions.

For example, the following `uppercaseString()` function:

```
fun uppercaseString(text: String): String {
    return text.uppercase()
}
fun main() {
    println(uppercaseString("hello"))
    // HELLO
}
```

Can also be written as a lambda expression:

```
fun main() {
    val upperCaseString = { text: String -> text.uppercase() }
    println(upperCaseString("hello"))
    // HELLO
}
```

Lambda expressions can be hard to understand at first glance, so let's break it down. Lambda expressions are written within curly braces `{}`.

Within the lambda expression, you write:

- The parameters followed by an `->`.
- The function body after the `->`.

In the previous example:

- `text` is a function parameter.
- `text` has type `String`.
- The function returns the result of the `.uppercase()` function called on `text`.
- The entire lambda expression is assigned to the `upperCaseString` variable with the assignment operator `=`.
- The lambda expression is called by using the variable `upperCaseString` like a function and the string `"hello"` as a parameter.
- The `println()` function prints the result.

If you declare a lambda without parameters, then there is no need to use `->`. For example:

```
{ println("Log message") }
```

Lambda expressions can be used in a number of ways. You can:

- Pass a lambda expression as a parameter to another function
- Return a lambda expression from a function
- Invoke a lambda expression on its own

Pass to another function

A great example of when it is useful to pass a lambda expression to a function, is using the `.filter()` function on collections:

```
fun main() {
        val numbers = listOf(1, -2, 3, -4, 5, -6)

    val positives = numbers.filter ({ x -> x > 0 })

    val isNegative = { x: Int -> x < 0 }
    val negatives = numbers.filter(isNegative)

    println(positives)
    // [1, 3, 5]
    println(negatives)
    // [-2, -4, -6]
    //sampleEnd
}
```

The `.filter()` function accepts a lambda expression as a predicate and applies it to each element of the list. The function keeps an element only if the predicate returns `true`:

- `{ x -> x > 0 }` returns `true` if the element is positive.
- `{ x -> x < 0 }` returns `true` if the element is negative.

This example demonstrates two ways of passing a lambda expression to a function:

- For positive numbers, the example adds the lambda expression directly in the `.filter()` function.
- For negative numbers, the example assigns the lambda expression to the `isNegative` variable. Then the `isNegative` variable is used as a function parameter in the `.filter()` function. In this case, you have to specify the type of function parameters (`x`) in the lambda expression.

If a lambda expression is the only function parameter, you can drop the function parentheses `()`:

```
val positives = numbers.filter { x -> x > 0 }
```

This is an example of a trailing lambda, which is discussed in more detail at the end of this chapter.

Another good example is using the `.map()` function to transform items in a collection:

```
fun main() {
        val numbers = listOf(1, -2, 3, -4, 5, -6)
    val doubled = numbers.map { x -> x * 2 }

    val isTripled = { x: Int -> x * 3 }
    val tripled = numbers.map(isTripled)

    println(doubled)
    // [2, -4, 6, -8, 10, -12]
    println(tripled)
    // [3, -6, 9, -12, 15, -18]
    //sampleEnd
}
```

The `.map()` function accepts a lambda expression as a transform function:

- `{ x -> x * 2 }` takes each element of the list and returns that element multiplied by 2.
- `{ x -> x * 3 }` takes each element of the list and returns that element multiplied by 3.

Function types

Before you can return a lambda expression from a function, you first need to understand **function types**.

You have already learned about basic types but functions themselves also have a type. Kotlin's type inference can infer a function's type from the parameter type. But there may be times when you need to explicitly specify the function type. The compiler needs the function type so that it knows what is and isn't allowed for that function.

The syntax for a function type has:

- Each parameter's type written within parentheses `()` and separated by commas `,`.
- The return type written after `->`.

For example: `(String) -> String` or `(Int, Int) -> Int`.

This is what a lambda expression looks like if a function type for `upperCaseString()` is defined:

```
val upperCaseString: (String) -> String = { text -> text.uppercase() }

fun main() {
    println(upperCaseString("hello"))
    // HELLO
}
```

If your lambda expression has no parameters, then the parentheses `()` are left empty. For example: `() -> Unit`

> You must declare parameter and return types either in the lambda expression or as a function type. Otherwise, the compiler won't be able to know what type your lambda expression is.
>
> For example, the following won't work:
>
> `val upperCaseString = { str -> str.uppercase() }`

Return from a function

Lambda expressions can be returned from a function. So that the compiler understands what type the lambda expression returned is, you must declare a function type.

In the following example, the `toSeconds()` function has function type `(Int) -> Int` because it always returns a lambda expression that takes a parameter of type `Int` and returns an `Int` value.

This example uses a `when` expression to determine which lambda expression is returned when `toSeconds()` is called:

```
fun toSeconds(time: String): (Int) -> Int = when (time) {
    "hour" -> { value -> value * 60 * 60 }
    "minute" -> { value -> value * 60 }
    "second" -> { value -> value }
    else -> { value -> value }
}

fun main() {
    val timesInMinutes = listOf(2, 10, 15, 1)
    val min2sec = toSeconds("minute")
    val totalTimeInSeconds = timesInMinutes.map(min2sec).sum()
    println("Total time is $totalTimeInSeconds secs")
    // Total time is 1680 secs
}
```

Invoke separately

Lambda expressions can be invoked on their own by adding parentheses `()` after the curly braces `{}` and including any parameters within the parentheses:

```
fun main() {
        println({ text: String -> text.uppercase() }("hello"))
    // HELLO
    //sampleEnd
}
```

Trailing lambdas

As you have already seen, if a lambda expression is the only function parameter, you can drop the function parentheses `()`. If a lambda expression is passed as the last parameter of a function, then the expression can be written outside the function parentheses `()`. In both cases, this syntax is called a **trailing lambda**.

For example, the `.fold()` function accepts an initial value and an operation:

```
fun main() {
        // The initial value is zero.
    // The operation sums the initial value with every item in the list cumulatively.
    println(listOf(1, 2, 3).fold(0, { x, item -> x + item })) // 6

    // Alternatively, in the form of a trailing lambda
    println(listOf(1, 2, 3).fold(0) { x, item -> x + item })  // 6
    //sampleEnd
}
```

For more information on lambda expressions, see Lambda expressions and anonymous functions.

The next step in our tour is to learn about classes in Kotlin.

Lambda expressions practice

Exercise 1

You have a list of actions supported by a web service, a common prefix for all requests, and an ID of a particular resource. To request an action `title` over the resource with ID: 5, you need to create the following URL: `https://example.com/book-info/5/title`. Use a lambda expression to create a list of URLs from the list of actions.

```
fun main() {
    val actions = listOf("title", "year", "author")
    val prefix = "https://example.com/book-info"
    val id = 5
    val urls = // Write your code here
    println(urls)
}
```

```
fun main() {
    val actions = listOf("title", "year", "author")
    val prefix = "https://example.com/book-info"
    val id = 5
    val urls = actions.map { action -> "$prefix/$id/$action" }
    println(urls)
}
```

Exercise 2

Write a function that takes an `Int` value and an action (a function with type `() -> Unit`) which then repeats the action the given number of times. Then use this function to print “Hello” 5 times.

```
fun repeatN(n: Int, action: () -> Unit) {
    // Write your code here
}

fun main() {
    // Write your code here
}
```

```
fun repeatN(n: Int, action: () -> Unit) {
    for (i in 1..n) {
        action()
    }
}

fun main() {
    repeatN(5) {
        println("Hello")
    }
}
```

Previous step

Next step

Classes

Kotlin supports object-oriented programming with classes and objects. Objects are useful for storing data in your program. Classes allow you to declare a set of characteristics for an object. When you create objects from a class, you can save time and effort because you don't have to declare these characteristics every time.

To declare a class, use the `class` keyword:

```
class Customer
```

Properties

Characteristics of a class's object can be declared in properties. You can declare properties for a class:

- Within parentheses `()` after the class name.

```
class Contact(val id: Int, var email: String)
```

- Within the class body defined by curly braces `{}`.

```
class Contact(val id: Int, var email: String) {
    val category: String = ""
}
```

We recommend that you declare properties as read-only (`val`) unless they need to be changed after an instance of the class is created.

You can declare properties without `val` or `var` within parentheses but these properties are not accessible after an instance has been created.



- The content contained within parentheses `()` is called the **class header**.
- You can use a trailing comma when declaring class properties.

Just like with function parameters, class properties can have default values:

```
class Contact(val id: Int, var email: String = "example@gmail.com") {
    val category: String = "work"
}
```

Create instance

To create an object from a class, you declare a class **instance** using a **constructor**.

By default, Kotlin automatically creates a constructor with the parameters declared in the class header.

For example:

```
class Contact(val id: Int, var email: String)

fun main() {
    val contact = Contact(1, "mary@gmail.com")
}
```

In the example:

- `Contact` is a class.
- `contact` is an instance of the `Contact` class.
- `id` and `email` are properties.
- `id` and `email` are used with the default constructor to create `contact`.

Kotlin classes can have many constructors, including ones that you define yourself. To learn more about how to declare multiple constructors, see Constructors.

Access properties

To access a property of an instance, write the name of the property after the instance name appended with a period `.`:

```
class Contact(val id: Int, var email: String)

fun main() {
    val contact = Contact(1, "mary@gmail.com")

    // Prints the value of the property: email
    println(contact.email)
    // mary@gmail.com

    // Updates the value of the property: email
    contact.email = "jane@gmail.com"

    // Prints the new value of the property: email
    println(contact.email)
    // jane@gmail.com
}
```

To concatenate the value of a property as part of a string, you can use string templates (`$`). For example:

```
println("Their email address is: ${contact.email}")
```

Member functions

In addition to declaring properties as part of an object's characteristics, you can also define an object's behavior with member functions.

In Kotlin, member functions must be declared within the class body. To call a member function on an instance, write the function name after the instance name appended with a period `.`. For example:

```
class Contact(val id: Int, var email: String) {
    fun printId() {
        println(id)
    }
}

fun main() {
    val contact = Contact(1, "mary@gmail.com")
    // Calls member function printId()
    contact.printId()
    // 1
}
```

Data classes

Kotlin has **data classes** which are particularly useful for storing data. Data classes have the same functionality as classes, but they come automatically with additional member functions. These member functions allow you to easily print the instance to readable output, compare instances of a class, copy instances, and more. As these functions are automatically available, you don't have to spend time writing the same boilerplate code for each of your classes.

To declare a data class, use the keyword `data`:

```
data class User(val name: String, val id: Int)
```

The Kotlin compiler only uses the properties defined inside the primary constructor when generating member functions. If you declare properties in the data class body, they aren't included in the output of the generated functions.

The most useful predefined member functions of data classes are:

Function	Description
`toString()`	Prints a readable string of the class instance and its properties.
`equals()` or `==`	Compares instances of a class.
`copy()`	Creates a class instance by copying another, potentially with some different properties.

See the following sections for examples of how to use each function:

- Print as string
- Compare instances
- Copy instance

Print as string

To print a readable string of a class instance, you can explicitly call the `toString()` function, or use print functions (`println()` and `print()`) which automatically call `toString()` for you:

```
data class User(val name: String, val id: Int)

fun main() {
        val user = User("Alex", 1)

    // Automatically uses toString() function so that output is easy to read
    println(user)
    // User(name=Alex, id=1)
    //sampleEnd
}
```

This is particularly useful when debugging or creating logs.

Compare instances

To compare data class instances, use the equality operator `==` :

```
data class User(val name: String, val id: Int)

fun main() {
        val user = User("Alex", 1)
    val secondUser = User("Alex", 1)
    val thirdUser = User("Max", 2)

    // Compares user to second user
    println("user == secondUser: ${user == secondUser}")
    // user == secondUser: true

    // Compares user to third user
    println("user == thirdUser: ${user == thirdUser}")
    // user == thirdUser: false
    //sampleEnd
}
```

Copy instance

To create an exact copy of a data class instance, call the `copy()` function on the instance.

To create a copy of a data class instance **and** change some properties, call the `copy()` function on the instance **and** add replacement values for properties as function parameters.

For example:

```
data class User(val name: String, val id: Int)

fun main() {
        val user = User("Alex", 1)

    // Creates an exact copy of user
    println(user.copy())
    // User(name=Alex, id=1)

    // Creates a copy of user with name: "Max"
    println(user.copy("Max"))
    // User(name=Max, id=1)

    // Creates a copy of user with id: 3
    println(user.copy(id = 3))
    // User(name=Alex, id=3)
    //sampleEnd
}
```

Creating a copy of an instance is safer than modifying the original instance because any code that relies on the original instance isn't affected by the copy and what you do with it.

For more information about data classes, see Data classes.

The last chapter of this tour is about Kotlin's null safety.

Practice

Exercise 1

Define a data class `Employee` with two properties: one for a name, and another for a salary. Make sure that the property for salary is mutable, otherwise you won't get a salary boost at the end of the year! The main function demonstrates how you can use this data class.

```
// Write your code here

fun main() {
    val emp = Employee("Mary", 20)
    println(emp)
    emp.salary += 10
    println(emp)
}
```

```
data class Employee(val name: String, var salary: Int)

fun main() {
    val emp = Employee("Mary", 20)
    println(emp)
    emp.salary += 10
    println(emp)
}
```

Exercise 2

Declare the additional data classes that are needed for this code to compile.

```
data class Person(val name: Name, val address: Address, val ownsAPet: Boolean = true)
// Write your code here
// data class Name(...)

fun main() {
    val person = Person(
        Name("John", "Smith"),
        Address("123 Fake Street", City("Springfield", "US")),
        ownsAPet = false
    )
}
```

```
data class Person(val name: Name, val address: Address, val ownsAPet: Boolean = true)
data class Name(val first: String, val last: String)
data class Address(val street: String, val city: City)
data class City(val name: String, val countryCode: String)

fun main() {
    val person = Person(
        Name("John", "Smith"),
        Address("123 Fake Street", City("Springfield", "US")),
        ownsAPet = false
    )
}
```

Exercise 3

To test your code, you need a generator that can create random employees. Define a `RandomEmployeeGenerator` class with a fixed list of potential names (inside the class body). Configure the class with a minimum and maximum salary (inside the class header). In the class body, define the `generateEmployee()` function. Once again, the main function demonstrates how you can use this class.

> In this exercise, you import a package so that you can use the `Random.nextInt()` function. For more information about importing packages, see Packages and imports.

Hint 1

Lists have an extension function called `.random()` that returns a random item within a list.

Hint 2

`Random.nextInt(from = ..., until = ...)` gives you a random `Int` number within specified limits.

```
import kotlin.random.Random

data class Employee(val name: String, var salary: Int)

// Write your code here

fun main() {
    val empGen = RandomEmployeeGenerator(10, 30)
    println(empGen.generateEmployee())
    println(empGen.generateEmployee())
    println(empGen.generateEmployee())
    empGen.minSalary = 50
    empGen.maxSalary = 100
    println(empGen.generateEmployee())
}
```

```
import kotlin.random.Random

data class Employee(val name: String, var salary: Int)

class RandomEmployeeGenerator(var minSalary: Int, var maxSalary: Int) {
    val names = listOf("John", "Mary", "Ann", "Paul", "Jack", "Elizabeth")
    fun generateEmployee() =
        Employee(names.random(),
            Random.nextInt(from = minSalary, until = maxSalary))
}

fun main() {
    val empGen = RandomEmployeeGenerator(10, 30)
    println(empGen.generateEmployee())
    println(empGen.generateEmployee())
    println(empGen.generateEmployee())
    empGen.minSalary = 50
    empGen.maxSalary = 100
    println(empGen.generateEmployee())
}
```

Previous step

Next step

Null safety

In Kotlin, it's possible to have a `null` value. Kotlin uses `null` values when something is missing or not yet set. You've already seen an example of Kotlin returning a `null` value in the Collections chapter when you tried to access a key-value pair with a key that doesn't exist in the map. Although it's useful to use `null` values in this way, you might run into problems if your code isn't prepared to handle them.

To help prevent issues with `null` values in your programs, Kotlin has null safety in place. Null safety detects potential problems with `null` values at compile time, rather than at run time.

Null safety is a combination of features that allow you to:

- Explicitly declare when `null` values are allowed in your program.
- Check for `null` values.
- Use safe calls to properties or functions that may contain `null` values.

- Declare actions to take if `null` values are detected.

Nullable types

Kotlin supports nullable types which allows the possibility for the declared type to have `null` values. By default, a type is **not** allowed to accept `null` values. Nullable types are declared by explicitly adding `?` after the type declaration.

For example:

```
fun main() {
    // neverNull has String type
    var neverNull: String = "This can't be null"

    // Throws a compiler error
    neverNull = null

    // nullable has nullable String type
    var nullable: String? = "You can keep a null here"

    // This is OK
    nullable = null

    // By default, null values aren't accepted
    var inferredNonNull = "The compiler assumes non-nullable"

    // Throws a compiler error
    inferredNonNull = null

    // notNull doesn't accept null values
    fun strLength(notNull: String): Int {
        return notNull.length
    }

    println(strLength(neverNull)) // 18
    println(strLength(nullable))  // Throws a compiler error
}
```

`length` is a property of the String class that contains the number of characters within a string.

Check for null values

You can check for the presence of `null` values within conditional expressions. In the following example, the `describeString()` function has an `if` statement that checks whether `maybeString` is **not** `null` and if its `length` is greater than zero:

```
fun describeString(maybeString: String?): String {
    if (maybeString != null && maybeString.length > 0) {
        return "String of length ${maybeString.length}"
    } else {
        return "Empty or null string"
    }
}

fun main() {
    val nullString: String? = null
    println(describeString(nullString))
    // Empty or null string
}
```

Use safe calls

To safely access properties of an object that might contain a `null` value, use the safe call operator `?.`. The safe call operator returns `null` if either the object or one of its accessed properties is `null`. This is useful if you want to avoid the presence of `null` values triggering errors in your code.

In the following example, the `lengthString()` function uses a safe call to return either the length of the string or `null`:

```
fun lengthString(maybeString: String?): Int? = maybeString?.length

fun main() {
    val nullString: String? = null
    println(lengthString(nullString))
    // null
}
```

Safe calls can be chained so that if any property of an object contains a `null` value, then `null` is returned without an error being thrown. For example:

```
person.company?.address?.country
```

The safe call operator can also be used to safely call an extension or member function. In this case, a null check is performed before the function is called. If the check detects a `null` value, then the call is skipped and `null` is returned.

In the following example, `nullString` is `null` so the invocation of `.uppercase()` is skipped and `null` is returned:

```
fun main() {
    val nullString: String? = null
    println(nullString?.uppercase())
    // null
}
```

Use Elvis operator

You can provide a default value to return if a `null` value is detected by using the **Elvis operator** `?:`.

Write on the left-hand side of the Elvis operator what should be checked for a `null` value. Write on the right-hand side of the Elvis operator what should be returned if a `null` value is detected.

In the following example, `nullString` is `null` so the safe call to access the `length` property returns a `null` value. As a result, the Elvis operator returns `0`:

```
fun main() {
    val nullString: String? = null
    println(nullString?.length ?: 0)
    // 0
}
```

For more information about null safety in Kotlin, see Null safety.

Practice

Exercise

You have the `employeeById` function that gives you access to a database of employees of a company. Unfortunately, this function returns a value of the `Employee?` type, so the result can be `null`. Your goal is to write a function that returns the salary of an employee when their `id` is provided, or `0` if the employee is missing from the database.

```
data class Employee (val name: String, var salary: Int)

fun employeeById(id: Int) = when(id) {
    1 -> Employee("Mary", 20)
    2 -> null
    3 -> Employee("John", 21)
    4 -> Employee("Ann", 23)
    else -> null
}

fun salaryById(id: Int) = // Write your code here

fun main() {
    println((1..5).sumOf { id -> salaryById(id) })
}
```

```
data class Employee (val name: String, var salary: Int)

fun employeeById(id: Int) = when(id) {
    1 -> Employee("Mary", 20)
    2 -> null
    3 -> Employee("John", 21)
    4 -> Employee("Ann", 23)
    else -> null
}

fun salaryById(id: Int) = employeeById(id)?.salary ?: 0

fun main() {
    println((1..5).sumOf { id -> salaryById(id) })
}
```

What's next?

Congratulations! Now that you have completed the beginner tour, take your understanding of Kotlin to the next level with our intermediate tour:

Previous step

Start intermediate Kotlin tour

Extension functions

In this chapter, you'll explore special Kotlin functions that make your code more concise and readable. Learn how they can help you use efficient design patterns to take your projects to the next level.

Extension functions

In software development, you often need to modify a program's behavior without changing the original source code. For example, you might want to add extra functionality to a class from a third-party library.

You can do this by adding *extension functions* to extend a class. You call extension functions the same way you call member functions of a class, using a period `.`.

Before introducing the complete syntax for extension functions, you need to understand what a **receiver** is. The receiver is what the function is called on. In other words, the receiver is where or with whom the information is shared.

```
fun main() {
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("The first item in the list is: ${readOnlyShapes.first()}")
    // The first item in the list is: triangle
}
```

Receiver

An example of sender and receiver

In this example, the `main()` function calls the `.first()` function to return the first element in a list. The `.first()` function is called **on** the `readOnlyShapes` variable, so the `readOnlyShapes` variable is the receiver.

To create an extension function, write the name of the class that you want to extend followed by a `.` and the name of your function. Continue with the rest of the function declaration, including its parameters and return type.

For example:

```
fun String.bold(): String = "<b>$this</b>"

fun main() {
    // "hello" is the receiver
    println("hello".bold())
    // <b>hello</b>
}
```

In this example:

- `String` is the extended class.
- `bold` is the name of the extension function.

- The `.bold()` extension function's return type is `String`.
- `"hello"`, an instance of `String`, as the receiver.
- The receiver is accessed inside the body by the keyword: `this`.
- A string template (`$`) is used to access the value of `this`.
- The `.bold()` extension function takes a string and returns it in a `<b>` HTML element for bold text.

Extension-oriented design

You can define extension functions anywhere, which enables you to create extension-oriented designs. These designs separate core functionality from useful but non-essential features, making your code easier to read and maintain.

A good example is the `HttpClient` class from the Ktor library, which helps perform network requests. The core of its functionality is a single function `request()`, which takes all the information needed for an HTTP request:

```
class HttpClient {
    fun request(method: String, url: String, headers: Map<String, String>): HttpResponse {
        // Network code
    }
}
```

In practice, the most popular HTTP requests are GET or POST requests. It makes sense for the library to provide shorter names for these common use cases. However, these don't require writing new network code, only a specific request call. In other words, they are perfect candidates to be defined as separate `.get()` and `.post()` extension functions:

```
fun HttpClient.get(url: String): HttpResponse = request("GET", url, emptyMap())
fun HttpClient.post(url: String): HttpResponse = request("POST", url, emptyMap())
```

These `.get()` and `.post()` functions extend the `HttpClient` class. They can directly use the `request()` function from the `HttpClient` class because they're called on an instance of the `HttpClient` class as the receiver. You can use these extension functions to call the `request()` function with the appropriate HTTP method, which simplifies your code and makes it easier to understand:

```
class HttpClient {
    fun request(method: String, url: String, headers: Map<String, String>): HttpResponse {
        println("Requesting $method to $url with headers: $headers")
        return HttpResponse("Response from $url")
    }
}

fun HttpClient.get(url: String): HttpResponse = request("GET", url, emptyMap())

fun main() {
    val client = HttpClient()

    // Making a GET request using request() directly
    val getResponseWithMember = client.request("GET", "https://example.com", emptyMap())

    // Making a GET request using the get() extension function
    // The client instance is the receiver
    val getResponseWithExtension = client.get("https://example.com")
}
```

This extension-oriented approach is widely used in Kotlin's standard library and other libraries. For example, the `String` class has many extension functions to help you work with strings.

For more information about extension functions, see Extensions.

Practice

Exercise 1

Write an extension function called `isPositive` that takes an integer and checks whether it is positive.

```
fun Int.// Write your code here

fun main() {
    println(1.isPositive())
    // true
}
```

```
fun Int.isPositive(): Boolean = this > 0

fun main() {
    println(1.isPositive())
    // true
}
```

Exercise 2

Write an extension function called `toLowercaseString` that takes a string and returns a lowercase version.

Hint

Use the `.lowercase()` function for the `String` type.

```
fun // Write your code here

fun main() {
    println("Hello World!".toLowercaseString())
    // hello world!
}
```

```
fun String.toLowercaseString(): String = this.lowercase()

fun main() {
    println("Hello World!".toLowercaseString())
    // hello world!
}
```

Previous step

Next step

Scope functions

In this chapter, you'll build on your understanding of extension functions to learn how to use scope functions to write more idiomatic code.

Scope functions

In programming, a scope is the area in which your variable or object is recognized. The most commonly referred to scopes are the global scope and the local scope:

- **Global scope** – a variable or object that is accessible from anywhere in the program.
- **Local scope** – a variable or object that is only accessible within the block or function where it is defined.

In Kotlin, there are also scope functions that allow you to create a temporary scope around an object and execute some code.

Scope functions make your code more concise because you don't have to refer to the name of your object within the temporary scope. Depending on the scope function, you can access the object either by referencing it via the keyword `this` or using it as an argument via the keyword `it`.

Kotlin has five scope functions in total: `let`, `apply`, `run`, `also`, and `with`.

Each scope function takes a lambda expression and returns either the object or the result of the lambda expression. In this tour, we explain each scope function and how to use it.

> You can also watch the Back to the Stdlib: Making the Most of Kotlin's Standard Library talk on scope functions by Sebastian Aigner, Kotlin developer advocate.

Let

Use the `let` scope function when you want to perform null checks in your code and later perform further actions with the returned object.

Consider the example:

```
fun sendNotification(recipientAddress: String): String {
    println("Yo $recipientAddress!")
    return "Notification sent!"
}

fun getNextAddress(): String {
    return "sebastian@jetbrains.com"
}

fun main() {
    val address: String? = getNextAddress()
    sendNotification(address)
}
```

The example has two functions:

- `sendNotification()`, which has a function parameter `recipientAddress` and returns a string.
- `getNextAddress()`, which has no function parameters and returns a string.

The example creates a variable `address` that has a nullable `String` type. But this becomes a problem when you call the `sendNotification()` function because this function doesn't expect that `address` could be a `null` value. The compiler reports an error as a result:

```
Argument type mismatch: actual type is 'String?', but 'String' was expected.
```

From the beginner tour, you already know that you can perform a null check with an if condition or use the Elvis operator `?:`. But what if you want to use the returned object later in your code? You could achieve this with an if condition **and** an else branch:

```
fun sendNotification(recipientAddress: String): String {
    println("Yo $recipientAddress!")
    return "Notification sent!"
}

fun getNextAddress(): String {
    return "sebastian@jetbrains.com"
}

fun main() {
        val address: String? = getNextAddress()
    val confirm = if(address ≠ null) {
        sendNotification(address)
    } else { null }
    //sampleEnd
}
```

However, a more concise approach is to use the `let` scope function:

```
fun sendNotification(recipientAddress: String): String {
    println("Yo $recipientAddress!")
    return "Notification sent!"
}

fun getNextAddress(): String {
    return "sebastian@jetbrains.com"
}

fun main() {
        val address: String? = getNextAddress()
    val confirm = address?.let {
        sendNotification(it)
    }
    //sampleEnd
}
```

The example:

- Creates variables called `address` and `confirm`.
- Uses a safe call for the `let` scope function on the `address` variable.
- Creates a temporary scope within the `let` scope function.
- Passes the `sendNotification()` function as a lambda expression into the `let` scope function.
- Refers to the `address` variable via `it`, using the temporary scope.
- Assigns the result to the `confirm` variable.

With this approach, your code can handle the `address` variable potentially being a `null` value, and you can use the `confirm` variable later in your code.

Apply

Use the `apply` scope function to initialize objects, like a class instance, at the time of creation rather than later on in your code. This approach makes your code easier to read and manage.

Consider the example:

```
class Client() {
    var token: String? = null
    fun connect() = println("connected!")
    fun authenticate() = println("authenticated!")
    fun getData() : String {
        println("getting data!")
        return "Mock data"
    }
}

val client = Client()

fun main() {
    client.token = "asdf"
    client.connect()
    // connected!
    client.authenticate()
    // authenticated!
    client.getData()
    // getting data!
}
```

The example has a `Client` class that contains one property called `token` and three member functions: `connect()`, `authenticate()`, and `getData()`.

The example creates `client` as an instance of the `Client` class before initializing its `token` property and calling its member functions in the `main()` function.

Although this example is compact, in the real world, it can be a while before you can configure and use the class instance (and its member functions) after you've created it. However, if you use the `apply` scope function you can create, configure and use member functions on your class instance all in the same place in your code:

```
class Client() {
    var token: String? = null
    fun connect() = println("connected!")
    fun authenticate() = println("authenticated!")
    fun getData() : String {
        println("getting data!")
        return "Mock data"
    }
}
val client = Client().apply {
    token = "asdf"
    connect()
    // connected!
    authenticate()
    // authenticated!
}

fun main() {
    client.getData()
    // getting data!
}
```

The example:

- Creates `client` as an instance of the `Client` class.
- Uses the `apply` scope function on the `client` instance.
- Creates a temporary scope within the `apply` scope function so that you don't have to explicitly refer to the `client` instance when accessing its properties or functions.
- Passes a lambda expression to the `apply` scope function that updates the `token` property and calls the `connect()` and `authenticate()` functions.
- Calls the `getData()` member function on the `client` instance in the `main()` function.

As you can see, this strategy is convenient when you are working with large pieces of code.

Run

Similar to `apply`, you can use the `run` scope function to initialize an object, but it's better to use `run` to initialize an object at a specific moment in your code **and** immediately compute a result.

Let's continue the previous example for the `apply` function, but this time, you want the `connect()` and `authenticate()` functions to be grouped so that they are called on every request.

For example:

```
class Client() {
    var token: String? = null
    fun connect() = println("connected!")
    fun authenticate() = println("authenticated!")
    fun getData() : String {
        println("getting data!")
        return "Mock data"
    }
}

val client: Client = Client().apply {
    token = "asdf"
}

fun main() {
    val result: String = client.run {
        connect()
        // connected!
        authenticate()
        // authenticated!
        getData()
        // getting data!
    }
}
```

The example:

- Creates `client` as an instance of the `Client` class.
- Uses the `apply` scope function on the `client` instance.
- Creates a temporary scope within the `apply` scope function so that you don't have to explicitly refer to the `client` instance when accessing its properties or functions.
- Passes a lambda expression to the `apply` scope function that updates the `token` property.

The `main()` function:

- Creates a `result` variable with type `String`.
- Uses the `run` scope function on the `client` instance.
- Creates a temporary scope within the `run` scope function so that you don't have to explicitly refer to the `client` instance when accessing its properties or functions.
- Passes a lambda expression to the `run` scope function that calls the `connect()`, `authenticate()`, and `getData()` functions.
- Assigns the result to the `result` variable.

Now you can use the returned result further in your code.

Also

Use the `also` scope function to complete an additional action with an object and then return the object to continue using it in your code, like writing a log.

Consider the example:

```
fun main() {
    val medals: List<String> = listOf("Gold", "Silver", "Bronze")
    val reversedLongUppercaseMedals: List<String> =
        medals
            .map { it.uppercase() }
            .filter { it.length > 4 }
            .reversed()
    println(reversedLongUppercaseMedals)
    // [BRONZE, SILVER]
}
```

The example:

- Creates the `medals` variable that contains a list of strings.
- Creates the `reversedLongUpperCaseMedals` variable that has the `List<String>` type.
- Uses the `.map()` extension function on the `medals` variable.
- Passes a lambda expression to the `.map()` function that refers to `medals` via the `it` keyword and calls the `.uppercase()` extension function on it.
- Uses the `.filter()` extension function on the `medals` variable.
- Passes a lambda expression as a predicate to the `.filter()` function that refers to `medals` via the `it` keyword and checks if the item in the list has more than 4 characters.
- Uses the `.reversed()` extension function on the `medals` variable.
- Assigns the result to the `reversedLongUpperCaseMedals` variable.
- Prints the list contained in the `reversedLongUpperCaseMedals` variable.

It would be useful to add some logging in between the function calls to see what is happening to the `medals` variable. The `also` function helps with that:

```
fun main() {
    val medals: List<String> = listOf("Gold", "Silver", "Bronze")
    val reversedLongUppercaseMedals: List<String> =
        medals
            .map { it.uppercase() }
            .also { println(it) }
            // [GOLD, SILVER, BRONZE]
            .filter { it.length > 4 }
            .also { println(it) }
            // [SILVER, BRONZE]
            .reversed()
    println(reversedLongUppercaseMedals)
    // [BRONZE, SILVER]
}
```

Now the example:

- Uses the `also` scope function on the `medals` variable.
- Creates a temporary scope within the `also` scope function so that you don't have to explicitly refer to the `medals` variable when using it as a function parameter.
- Passes a lambda expression to the `also` scope function that calls the `println()` function using the `medals` variable as a function parameter via the `it` keyword.

Since the `also` function returns the object, it is useful for not only logging but debugging, chaining multiple operations, and performing other side effect operations that don't affect the main flow of your code.

With

Unlike the other scope functions, `with` is not an extension function, so the syntax is different. You pass the receiver object to `with` as an argument.

Use the `with` scope function when you want to call multiple functions on an object.

Consider this example:

```
class Canvas {
    fun rect(x: Int, y: Int, w: Int, h: Int): Unit = println("$x, $y, $w, $h")
    fun circ(x: Int, y: Int, rad: Int): Unit = println("$x, $y, $rad")
    fun text(x: Int, y: Int, str: String): Unit = println("$x, $y, $str")
}

fun main() {
    val mainMonitorPrimaryBufferBackedCanvas = Canvas()

    mainMonitorPrimaryBufferBackedCanvas.text(10, 10, "Foo")
    mainMonitorPrimaryBufferBackedCanvas.rect(20, 30, 100, 50)
    mainMonitorPrimaryBufferBackedCanvas.circ(40, 60, 25)
    mainMonitorPrimaryBufferBackedCanvas.text(15, 45, "Hello")
    mainMonitorPrimaryBufferBackedCanvas.rect(70, 80, 150, 100)
    mainMonitorPrimaryBufferBackedCanvas.circ(90, 110, 40)
    mainMonitorPrimaryBufferBackedCanvas.text(35, 55, "World")
    mainMonitorPrimaryBufferBackedCanvas.rect(120, 140, 200, 75)
    mainMonitorPrimaryBufferBackedCanvas.circ(160, 180, 55)
    mainMonitorPrimaryBufferBackedCanvas.text(50, 70, "Kotlin")
}
```

The example creates a `Canvas` class that has three member functions: `rect()`, `circ()`, and `text()`. Each of these member functions prints a statement constructed from the function parameters that you provide.

The example creates `mainMonitorPrimaryBufferBackedCanvas` as an instance of the `Canvas` class before calling a sequence of member functions on the instance with different function parameters.

You can see that this code is hard to read. If you use the `with` function, the code is streamlined:

```
class Canvas {
    fun rect(x: Int, y: Int, w: Int, h: Int): Unit = println("$x, $y, $w, $h")
    fun circ(x: Int, y: Int, rad: Int): Unit = println("$x, $y, $rad")
    fun text(x: Int, y: Int, str: String): Unit = println("$x, $y, $str")
}

fun main() {
        val mainMonitorSecondaryBufferBackedCanvas = Canvas()
    with(mainMonitorSecondaryBufferBackedCanvas) {
        text(10, 10, "Foo")
        rect(20, 30, 100, 50)
        circ(40, 60, 25)
        text(15, 45, "Hello")
        rect(70, 80, 150, 100)
        circ(90, 110, 40)
        text(35, 55, "World")
        rect(120, 140, 200, 75)
        circ(160, 180, 55)
        text(50, 70, "Kotlin")
    }
    //sampleEnd
}
```

This example:

- Uses the `with` scope function with the `mainMonitorSecondaryBufferBackedCanvas` instance as the receiver.
- Creates a temporary scope within the `with` scope function so that you don't have to explicitly refer to the `mainMonitorSecondaryBufferBackedCanvas` instance when calling its member functions.
- Passes a lambda expression to the `with` scope function that calls a sequence of member functions with different function parameters.

Now that this code is much easier to read, you are less likely to make mistakes.

Use case overview

This section has covered the different scope functions available in Kotlin and their main use cases for making your code more idiomatic. You can use this table as a quick reference. It's important to note that you don't need a complete understanding of how these functions work in order to use them in your code.

Function	Access to `x` via	Return value	Use case
`let`	`it`	Lambda result	Perform null checks in your code and later perform further actions with the returned object.
`apply`	`this`	`x`	Initialize objects at the time of creation.
`run`	`this`	Lambda result	Initialize objects at the time of creation **AND** compute a result.
`also`	`it`	`x`	Complete additional actions before returning the object.

Function	Access to `x` via	Return value	Use case
`with`	`this`	Lambda result	Call multiple functions on an object.

For more information about scope functions, see Scope functions.

Practice

Exercise 1

Rewrite the `.getPriceInEuros()` function as a single-expression function that uses safe call operators `?.` and the `let` scope function.

Hint

Use safe call operators `?.` to safely access the `priceInDollars` property from the `getProductInfo()` function. Then, use the `let` scope function to convert the value of `priceInDollars` into euros.

```
data class ProductInfo(val priceInDollars: Double?)

class Product {
    fun getProductInfo(): ProductInfo? {
        return ProductInfo(100.0)
    }
}

// Rewrite this function
fun Product.getPriceInEuros(): Double? {
    val info = getProductInfo()
    if (info == null) return null
    val price = info.priceInDollars
    if (price == null) return null
    return convertToEuros(price)
}

fun convertToEuros(dollars: Double): Double {
    return dollars * 0.85
}

fun main() {
    val product = Product()
    val priceInEuros = product.getPriceInEuros()

    if (priceInEuros != null) {
        println("Price in Euros: €$priceInEuros")
        // Price in Euros: €85.0
    } else {
        println("Price information is not available.")
    }
}
```

```
data class ProductInfo(val priceInDollars: Double?)

class Product {
    fun getProductInfo(): ProductInfo? {
        return ProductInfo(100.0)
    }
}

fun Product.getPriceInEuros() = getProductInfo()?.priceInDollars?.let { convertToEuros(it) }

fun convertToEuros(dollars: Double): Double {
    return dollars * 0.85
}

fun main() {
    val product = Product()
    val priceInEuros = product.getPriceInEuros()

    if (priceInEuros != null) {
        println("Price in Euros: €$priceInEuros")
        // Price in Euros: €85.0
    } else {
        println("Price information is not available.")
    }
}
```

Exercise 2

You have an `updateEmail()` function that updates the email address of a user. Use the `apply` scope function to update the email address and then the `also` scope function to print a log message: `Updating email for user with ID: ${it.id}`.

```
data class User(val id: Int, var email: String)

fun updateEmail(user: User, newEmail: String): User = // Write your code here

fun main() {
    val user = User(1, "old_email@example.com")
    val updatedUser = updateEmail(user, "new_email@example.com")
    // Updating email for user with ID: 1

    println("Updated User: $updatedUser")
    // Updated User: User(id=1, email=new_email@example.com)
}
```

```
data class User(val id: Int, var email: String)

fun updateEmail(user: User, newEmail: String): User = user.apply {
    this.email = newEmail
}.also { println("Updating email for user with ID: ${it.id}") }

fun main() {
    val user = User(1, "old_email@example.com")
    val updatedUser = updateEmail(user, "new_email@example.com")
    // Updating email for user with ID: 1

    println("Updated User: $updatedUser")
    // Updated User: User(id=1, email=new_email@example.com)
}
```

Previous step

Next step

Lambda expressions with receiver

In this chapter, you'll learn how to use receivers with another type of function, lambda expressions, and how they can help you create a domain-specific language.

Lambda expressions with receiver

In the beginner tour, you learned how to use lambda expressions. Lambda expressions can also have a receiver. In this case, lambda expressions can access any member functions or properties of the receiver without having to explicitly specify the receiver each time. Without these additional references, your code is easier to read and maintain.

> Lambda expressions with receiver are also known as function literals with receiver.

The syntax for a lambda expression with receiver is different when you define the function type. First, write the receiver that you want to extend. Next, put a `.` and then complete the rest of your function type definition. For example:

```
MutableList<Int>.() -> Unit
```

This function type has:

- `MutableList<Int>` as the receiver.
- No function parameters within the parentheses `()`.
- No return value: `Unit`.

Consider this example that draws shapes on a canvas:

```
class Canvas {
    fun drawCircle() = println("🟠 Drawing a circle")
    fun drawSquare() = println("🟥 Drawing a square")
}

// Lambda expression with receiver definition
fun render(block: Canvas.() → Unit): Canvas {
    val canvas = Canvas()
    // Use the lambda expression with receiver
    canvas.block()
    return canvas
}

fun main() {
    render {
        drawCircle()
        // 🟠 Drawing a circle
        drawSquare()
        // 🟥 Drawing a square
    }
}
```

In this example:

- The `Canvas` class has two functions that simulate drawing a circle or a square.
- The `render()` function takes a `block` parameter and returns an instance of the `Canvas` class.
- The `block` parameter is a lambda expression with receiver, where the `Canvas` class is the receiver.
- The `render()` function creates an instance of the `Canvas` class and calls the `block()` lambda expression on the `canvas` instance, using it as the receiver.
- The `main()` function calls the `render()` function with a lambda expression, which is passed to the `block` parameter.
- Inside the lambda passed to the `render()` function, the program calls the `drawCircle()` and `drawSquare()` functions on an instance of the `Canvas` class.

  Because the `drawCircle()` and `drawSquare()` functions are called in the lambda expression with receiver, they can be called directly as if they are inside the `Canvas` class.

Lambda expressions with receiver are helpful when you want to create a domain-specific language (DSL). Since you have access to the receiver's member functions and properties without explicitly referencing the receiver, your code becomes leaner.

To demonstrate this, consider an example that configures items in a menu. Let's begin with a `MenuItem` class and a `Menu` class that contains a function to add items to the menu called `item()`, as well as a list of all menu items `items`:

```
class MenuItem(val name: String)

class Menu(val name: String) {
    val items = mutableListOf<MenuItem>()

    fun item(name: String) {
        items.add(MenuItem(name))
    }
}
```

Let's use a lambda expression with receiver passed as a function parameter (`init`) to the `menu()` function that builds a menu as a starting point:

```
fun menu(name: String, init: Menu.() -> Unit): Menu {
    // Creates an instance of the Menu class
    val menu = Menu(name)
    // Calls the lambda expression with receiver init() on the class instance
    menu.init()
    return menu
}
```

Now you can use the DSL to configure a menu and create a `printMenu()` function to print the menu structure to the console:

```
class MenuItem(val name: String)

class Menu(val name: String) {
    val items = mutableListOf<MenuItem>()

    fun item(name: String) {
        items.add(MenuItem(name))
    }
}

fun menu(name: String, init: Menu.() -> Unit): Menu {
    val menu = Menu(name)
    menu.init()
    return menu
}

fun printMenu(menu: Menu) {
    println("Menu: ${menu.name}")
    menu.items.forEach { println("  Item: ${it.name}") }
}

// Use the DSL
fun main() {
    // Create the menu
    val mainMenu = menu("Main Menu") {
        // Add items to the menu
        item("Home")
        item("Settings")
        item("Exit")
    }

    // Print the menu
    printMenu(mainMenu)
    // Menu: Main Menu
    //   Item: Home
    //   Item: Settings
    //   Item: Exit
}
```

As you can see, using a lambda expression with receiver greatly simplifies the code needed to create your menu. Lambda expressions are not only useful for setup and creation but also for configuration. They are commonly used in building DSLs for APIs, UI frameworks, and configuration builders to produce streamlined code, allowing you to focus more easily on the underlying code structure and logic.

Kotlin's ecosystem has many examples of this design pattern, such as in the `buildList()` and `buildString()` functions from the standard library.

Lambda expressions with receivers can be combined with **type-safe builders** in Kotlin to make DSLs that detect any problems with types at compile time rather than at runtime. To learn more, see Type-safe builders.

Practice

Exercise 1

You have a `fetchData()` function that accepts a lambda expression with receiver. Update the lambda expression to use the `append()` function so that the output of your code is: `Data received - Processed`.

```
fun fetchData(callback: StringBuilder.() -> Unit) {
    val builder = StringBuilder("Data received")
    builder.callback()
}

fun main() {
    fetchData {
        // Write your code here
        // Data received - Processed
    }
}
```

```
fun fetchData(callback: StringBuilder.() -> Unit) {
    val builder = StringBuilder("Data received")
    builder.callback()
}

fun main() {
    fetchData {
        append(" - Processed")
        println(this.toString())
        // Data received - Processed
    }
}
```

Exercise 2

You have a `Button` class and `ButtonEvent` and `Position` data classes. Write some code that triggers the `onEvent()` member function of the `Button` class to trigger a double-click event. Your code should print `"Double click!"`.

```
class Button {
    fun onEvent(action: ButtonEvent.() -> Unit) {
        // Simulate a double-click event (not a right-click)
        val event = ButtonEvent(isRightClick = false, amount = 2, position = Position(100, 200))
        event.action() // Trigger the event callback
    }
}

data class ButtonEvent(
    val isRightClick: Boolean,
    val amount: Int,
    val position: Position
)

data class Position(
    val x: Int,
    val y: Int
)

fun main() {
    val button = Button()

    button.onEvent {
        // Write your code here
        // Double click!
    }
}
```

```
class Button {
    fun onEvent(action: ButtonEvent.() -> Unit) {
        // Simulate a double-click event (not a right-click)
        val event = ButtonEvent(isRightClick = false, amount = 2, position = Position(100, 200))
        event.action() // Trigger the event callback
    }
}

data class ButtonEvent(
    val isRightClick: Boolean,
    val amount: Int,
    val position: Position
)

data class Position(
    val x: Int,
    val y: Int
)

fun main() {
    val button = Button()

    button.onEvent {
        if (!isRightClick && amount == 2) {
            println("Double click!")
            // Double click!
        }
    }
}
```

Exercise 3

Write a function that creates a copy of a list of integers where every element is incremented by 1. Use the provided function skeleton that extends `List<Int>` with an `incremented` function.

```
fun List<Int>.incremented(): List<Int> {
    val originalList = this
    return buildList {
        // Write your code here
    }
}

fun main() {
    val originalList = listOf(1, 2, 3)
    val newList = originalList.incremented()
    println(newList)
    // [2, 3, 4]
}
```

```
fun List<Int>.incremented(): List<Int> {
    val originalList = this
    return buildList {
        for (n in originalList) add(n + 1)
    }
}

fun main() {
    val originalList = listOf(1, 2, 3)
    val newList = originalList.incremented()
    println(newList)
    // [2, 3, 4]
}
```

Previous step

Next step

Classes and interfaces

In the beginner tour, you learned how to use classes and data classes to store data and maintain a collection of characteristics that can be shared in your code. Eventually, you will want to create a hierarchy to efficiently share code within your projects. This chapter explains the options Kotlin provides for sharing code and how they can make your code safer and easier to maintain.

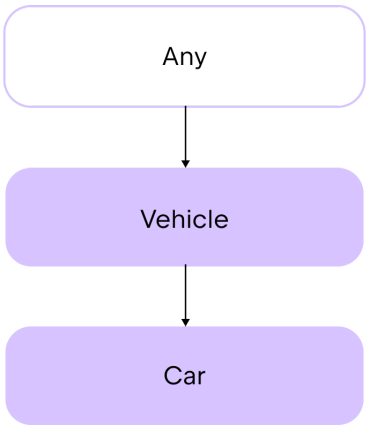
Class inheritance

In a previous chapter, we covered how you can use extension functions to extend classes without modifying the original source code. But what if you are working on something complex where sharing code **between** classes would be useful? In such cases, you can use class inheritance.

By default, classes in Kotlin can't be inherited. Kotlin is designed this way to prevent unintended inheritance and make your classes easier to maintain.

Kotlin classes only support **single inheritance**, meaning it is only possible to inherit from **one class at a time**. This class is called the **parent**.

The parent of a class inherits from another class (the grandparent), forming a hierarchy. At the top of Kotlin's class hierarchy is the common parent class: `Any`. All classes ultimately inherit from the `Any` class:



An example of the class hierarchy with Any type

The `Any` class provides the `toString()` function as a member function automatically. Therefore, you can use this inherited function in any of your classes. For example:

```
class Car(val make: String, val model: String, val numberOfDoors: Int)

fun main() {
        val car1 = Car("Toyota", "Corolla", 4)

    // Uses the .toString() function via string templates to print class properties
    println("Car1: make=${car1.make}, model=${car1.model}, numberOfDoors=${car1.numberOfDoors}")
    // Car1: make=Toyota, model=Corolla, numberOfDoors=4
    //sampleEnd
}
```

If you want to use inheritance to share some code between classes, first consider using abstract classes.

Abstract classes

Abstract classes can be inherited by default. The purpose of abstract classes is to provide members that other classes inherit or implement. As a result, they have a constructor, but you can't create instances from them. Within the child class, you define the behavior of the parent's properties and functions with the `override` keyword. In this way, you can say that the child class "overrides" the members of the parent class.

When you define the behavior of an inherited function or property, we call that an **implementation**.

Abstract classes can contain both functions and properties **with** implementation as well as functions and properties **without** implementation, known as abstract functions and properties.

To create an abstract class, use the `abstract` keyword:

```
abstract class Animal
```

To declare a function or a property **without** an implementation, you also use the `abstract` keyword:

```
abstract fun makeSound()
abstract val sound: String
```

For example, let's say that you want to create an abstract class called `Product` that you can create child classes from to define different product categories:

```
abstract class Product(val name: String, var price: Double) {
    // Abstract property for the product category
    abstract val category: String

    // A function that can be shared by all products
    fun productInfo(): String {
        return "Product: $name, Category: $category, Price: $price"
    }
}
```

In the abstract class:

- The constructor has two parameters for the product's `name` and `price`.
- There is an abstract property that contains the product category as a string.
- There is a function that prints information about the product.

Let's create a child class for electronics. Before you define an implementation for the `category` property in the child class, you must use the `override` keyword:

```
class Electronic(name: String, price: Double, val warranty: Int) : Product(name, price) {
    override val category = "Electronic"
}
```

The `Electronic` class:

- Inherits from the `Product` abstract class.
- Has an additional parameter in the constructor: `warranty`, which is specific to electronics.
- Overrides the `category` property to contain the string `"Electronic"`.

Now, you can use these classes like this:

```
abstract class Product(val name: String, var price: Double) {
    // Abstract property for the product category
    abstract val category: String

    // A function that can be shared by all products
    fun productInfo(): String {
        return "Product: $name, Category: $category, Price: $price"
    }
}

class Electronic(name: String, price: Double, val warranty: Int) : Product(name, price) {
    override val category = "Electronic"
}

fun main() {
    // Creates an instance of the Electronic class
    val laptop = Electronic(name = "Laptop", price = 1000.0, warranty = 2)

    println(laptop.productInfo())
    // Product: Laptop, Category: Electronic, Price: 1000.0
}
```

Although abstract classes are great for sharing code in this way, they are restricted because classes in Kotlin only support single inheritance. If you need to inherit from multiple sources, consider using interfaces.

Interfaces

Interfaces are similar to classes, but they have some differences:

- You can't create an instance of an interface. They don't have a constructor or header.
- Their functions and properties are implicitly inheritable by default. In Kotlin, we say that they are "open."
- You don't need to mark their functions as `abstract` if you don't give them an implementation.

Similar to abstract classes, you use interfaces to define a set of functions and properties that classes can inherit and implement later. This approach helps you focus on the abstraction described by the interface, rather than the specific implementation details. Using interfaces makes your code:

- More modular, as it isolates different parts, allowing them to evolve independently.
- Easier to understand by grouping related functions into a cohesive set.
- Easier to test, as you can quickly swap an implementation with a mock for testing.

To declare an interface, use the `interface` keyword:

```
interface PaymentMethod
```

Interface implementation

Interfaces support multiple inheritance so a class can implement multiple interfaces at once. First, let's consider the scenario where a class implements **one** interface.

To create a class that implements an interface, add a colon after your class header, followed by the interface name that you want to implement. You don't use parentheses `()` after the interface name because interfaces don't have a constructor:

```
class CreditCardPayment : PaymentMethod
```

For example:

```
interface PaymentMethod {
    // Functions are inheritable by default
    fun initiatePayment(amount: Double): String
}

class CreditCardPayment(val cardNumber: String, val cardHolderName: String, val expiryDate: String) :
PaymentMethod {
    override fun initiatePayment(amount: Double): String {
        // Simulate processing payment with credit card
        return "Payment of $$amount initiated using Credit Card ending in ${cardNumber.takeLast(4)}."
    }
}

fun main() {
    val paymentMethod = CreditCardPayment("1234 5678 9012 3456", "John Doe", "12/25")
    println(paymentMethod.initiatePayment(100.0))
    // Payment of $100.0 initiated using Credit Card ending in 3456.
}
```

In the example:

- `PaymentMethod` is an interface that has an `initiatePayment()` function without an implementation.
- `CreditCardPayment` is a class that implements the `PaymentMethod` interface.
- The `CreditCardPayment` class overrides the inherited `initiatePayment()` function.
- `paymentMethod` is an instance of the `CreditCardPayment` class.
- The overridden `initiatePayment()` function is called on the `paymentMethod` instance with a parameter of `100.0`.

To create a class that implements **multiple** interfaces, add a colon after your class header followed by the name of the interfaces that you want to implement separated by a comma:

```
class CreditCardPayment : PaymentMethod, PaymentType
```

For example:

```
interface PaymentMethod {
    fun initiatePayment(amount: Double): String
}

interface PaymentType {
    val paymentType: String
}

class CreditCardPayment(val cardNumber: String, val cardHolderName: String, val expiryDate: String) :
PaymentMethod,
    PaymentType {
    override fun initiatePayment(amount: Double): String {
        // Simulate processing payment with credit card
        return "Payment of $$amount initiated using Credit Card ending in ${cardNumber.takeLast(4)}."
    }

    override val paymentType: String = "Credit Card"
}

fun main() {
    val paymentMethod = CreditCardPayment("1234 5678 9012 3456", "John Doe", "12/25")
    println(paymentMethod.initiatePayment(100.0))
    // Payment of $100.0 initiated using Credit Card ending in 3456.

    println("Payment is by ${paymentMethod.paymentType}")
    // Payment is by Credit Card
}
```

In the example:

- `PaymentMethod` is an interface that has the `initiatePayment()` function without an implementation.
- `PaymentType` is an interface that has the `paymentType` property that isn't initialized.
- `CreditCardPayment` is a class that implements the `PaymentMethod` and `PaymentType` interfaces.
- The `CreditCardPayment` class overrides the inherited `initiatePayment()` function and the `paymentType` property.
- `paymentMethod` is an instance of the `CreditCardPayment` class.

- The overridden `initiatePayment()` function is called on the `paymentMethod` instance with a parameter of `100.0`.
- The overridden `paymentType` property is accessed on the `paymentMethod` instance.

For more information about interfaces and interface inheritance, see Interfaces.

Delegation

Interfaces are useful, but if your interface contains many functions, its child classes can end up with a lot of boilerplate code. If you only want to override a small part of a class's behavior, you need to repeat yourself a lot.

> Boilerplate code is a chunk of code that is reused with little or no alteration in multiple parts of a software project.

For example, let's say that you have an interface called `DrawingTool` that contains a number of functions and one property called `color`:

```
interface DrawingTool {
    val color: String
    fun draw(shape: String)
    fun erase(area: String)
    fun getToolInfo(): String
}
```

You create a class called `PenTool` which implements the `DrawingTool` interface and provides implementations for all of its members:

```
class PenTool : DrawingTool {
    override val color: String = "black"

    override fun draw(shape: String) {
        println("Drawing $shape using a pen in $color")
    }

    override fun erase(area: String) {
        println("Erasing $area with pen tool")
    }

    override fun getToolInfo(): String {
        return "PenTool(color=$color)"
    }
}
```

You want to create a class like `PenTool` with the same behavior but a different `color` value. One approach is to create a new class that expects an object implementing the `DrawingTool` interface as a parameter, like a `PenTool` class instance. Then, inside the class, you can override the `color` property.

But in this scenario, you need to add implementations for each member of the `DrawingTool` interface:

```
interface DrawingTool {
    val color: String
    fun draw(shape: String)
    fun erase(area: String)
    fun getToolInfo(): String
}

class PenTool : DrawingTool {
    override val color: String = "black"

    override fun draw(shape: String) {
        println("Drawing $shape using a pen in $color")
    }

    override fun erase(area: String) {
        println("Erasing $area with pen tool")
    }

    override fun getToolInfo(): String {
        return "PenTool(color=$color)"
    }
}
class CanvasSession(val tool: DrawingTool) : DrawingTool {
    override val color: String = "blue"

    override fun draw(shape: String) {
        tool.draw(shape)
    }

    override fun erase(area: String) {
        tool.erase(area)
    }

    override fun getToolInfo(): String {
        return tool.getToolInfo()
    }
}
fun main() {
    val pen = PenTool()
    val session = CanvasSession(pen)

    println("Pen color: ${pen.color}")
    // Pen color: black

    println("Session color: ${session.color}")
    // Session color: blue

    session.draw("circle")
    // Drawing circle with pen in black

    session.erase("top-left corner")
    // Erasing top-left corner with pen tool

    println(session.getToolInfo())
    // PenTool(color=black)
}
```

You can see that if you have a large number of member functions in the `DrawingTool` interface, the amount of boilerplate code in the `CanvasSession` class can be large. However, there is an alternative.

In Kotlin, you can delegate the interface implementation to a class instance using the `by` keyword. For example:

```
class CanvasSession(val tool: DrawingTool) : DrawingTool by tool
```

Here, `tool` is the name of the `PenTool` class instance where the implementations of member functions are delegated to.

Now you don't have to add implementations for the member functions in the `CanvasSession` class. The compiler does this for you automatically from the `PenTool` class. This saves you from having to write a lot of boilerplate code. Instead, you add code only for the behavior you want to change for your child class.

For example, if you want to change the value of the `color` property:

```
interface DrawingTool {
    val color: String
    fun draw(shape: String)
    fun erase(area: String)
    fun getToolInfo(): String
}

class PenTool : DrawingTool {
    override val color: String = "black"

    override fun draw(shape: String) {
        println("Drawing $shape using a pen in $color")
    }

    override fun erase(area: String) {
        println("Erasing $area with pen tool")
    }

    override fun getToolInfo(): String {
        return "PenTool(color=$color)"
    }
}

class CanvasSession(val tool: DrawingTool) : DrawingTool by tool {
    // No boilerplate code!
    override val color: String = "blue"
}
fun main() {
    val pen = PenTool()
    val session = CanvasSession(pen)

    println("Pen color: ${pen.color}")
    // Pen color: black

    println("Session color: ${session.color}")
    // Session color: blue

    session.draw("circle")
    // Drawing circle with pen in black

    session.erase("top-left corner")
    // Erasing top-left corner with pen tool

    println(session.getToolInfo())
    // PenTool(color=black)
}
```

If you want to, you can also override the behavior of an inherited member function in the `CanvasSession` class, but now you don't have to add new lines of code for every inherited member function.

For more information, see Delegation.

Practice

Exercise 1

Imagine you're working on a smart home system. A smart home typically has different types of devices that all have some basic features but also unique behaviors. In the code sample below, complete the `abstract` class called `SmartDevice` so that the child class `SmartLight` can compile successfully.

Then, create another child class called `SmartThermostat` that inherits from the `SmartDevice` class and implements `turnOn()` and `turnOff()` functions that return print statements describing which thermostat is heating or turned off. Finally, add another function called `adjustTemperature()` that accepts a temperature measurement as an input and prints: `$name thermostat set to $temperature°C.`

Hint

In the `SmartDevice` class, add the `turnOn()` and `turnOff()` functions so that you can override their behavior later in the `SmartThermostat` class.

```
abstract class // Write your code here

class SmartLight(name: String) : SmartDevice(name) {
    override fun turnOn() {
        println("$name is now ON.")
    }

    override fun turnOff() {
        println("$name is now OFF.")
    }

   fun adjustBrightness(level: Int) {
        println("Adjusting $name brightness to $level%.")
    }
}

class SmartThermostat // Write your code here

fun main() {
    val livingRoomLight = SmartLight("Living Room Light")
    val bedroomThermostat = SmartThermostat("Bedroom Thermostat")

    livingRoomLight.turnOn()
    // Living Room Light is now ON.
    livingRoomLight.adjustBrightness(10)
    // Adjusting Living Room Light brightness to 10%.
    livingRoomLight.turnOff()
    // Living Room Light is now OFF.

    bedroomThermostat.turnOn()
    // Bedroom Thermostat thermostat is now heating.
    bedroomThermostat.adjustTemperature(5)
    // Bedroom Thermostat thermostat set to 5°C.
    bedroomThermostat.turnOff()
    // Bedroom Thermostat thermostat is now off.
}
```

```
abstract class SmartDevice(val name: String) {
    abstract fun turnOn()
    abstract fun turnOff()
}

class SmartLight(name: String) : SmartDevice(name) {
    override fun turnOn() {
        println("$name is now ON.")
    }

    override fun turnOff() {
        println("$name is now OFF.")
    }

   fun adjustBrightness(level: Int) {
        println("Adjusting $name brightness to $level%.")
    }
}

class SmartThermostat(name: String) : SmartDevice(name) {
    override fun turnOn() {
        println("$name thermostat is now heating.")
    }

    override fun turnOff() {
        println("$name thermostat is now off.")
    }

   fun adjustTemperature(temperature: Int) {
        println("$name thermostat set to $temperature°C.")
    }
}


fun main() {
    val livingRoomLight = SmartLight("Living Room Light")
    val bedroomThermostat = SmartThermostat("Bedroom Thermostat")

    livingRoomLight.turnOn()
    // Living Room Light is now ON.
    livingRoomLight.adjustBrightness(10)
    // Adjusting Living Room Light brightness to 10%.
    livingRoomLight.turnOff()
    // Living Room Light is now OFF.

    bedroomThermostat.turnOn()
    // Bedroom Thermostat thermostat is now heating.
    bedroomThermostat.adjustTemperature(5)
    // Bedroom Thermostat thermostat set to 5°C.
    bedroomThermostat.turnOff()
    // Bedroom Thermostat thermostat is now off.
}
```

Exercise 2

Create an interface called `Media` that you can use to implement specific media classes like `Audio`, `Video`, or `Podcast`. Your interface must include:

- A property called `title` to represent the title of the media.
- A function called `play()` to play the media.

Then, create a class called `Audio` that implements the `Media` interface. The `Audio` class must use the `title` property in its constructor as well as have an additional property called `composer` that has `String` type. In the class, implement the `play()` function to print the following: `"Playing audio: $title, composed by $composer"`.

Hint

You can use the `override` keyword in class headers to implement a property from an interface in the constructor.

```
interface // Write your code here

class // Write your code here

fun main() {
    val audio = Audio("Symphony No. 5", "Beethoven")
    audio.play()
   // Playing audio: Symphony No. 5, composed by Beethoven
}
```

```
interface Media {
    val title: String
    fun play()
}

class Audio(override val title: String, val composer: String) : Media {
    override fun play() {
        println("Playing audio: $title, composed by $composer")
    }
}

fun main() {
    val audio = Audio("Symphony No. 5", "Beethoven")
    audio.play()
   // Playing audio: Symphony No. 5, composed by Beethoven
}
```

Exercise 3

You're building a payment processing system for an e-commerce application. Each payment method needs to be able to authorize a payment and process a transaction. Some payments also need to be able to process refunds.

1. In the `Refundable` interface, add a function called `refund()` to process refunds.
2. In the `PaymentMethod` abstract class:
    - Add a function called `authorize()` that takes an amount and prints a message containing the amount.
    - Add an abstract function called `processPayment()` that also takes an amount.
3. Create a class called `CreditCard` that implements the `Refundable` interface and `PaymentMethod` abstract class. In this class, add implementations for the `refund()` and `processPayment()` functions so that they print the following statements:
    - `"Refunding $amount to the credit card."`
    - `"Processing credit card payment of $amount."`

```
interface Refundable {
    // Write your code here
}

abstract class PaymentMethod(val name: String) {
    // Write your code here
}

class CreditCard // Write your code here

fun main() {
    val visa = CreditCard("Visa")

    visa.authorize(100.0)
    // Authorizing payment of $100.0.
    visa.processPayment(100.0)
    // Processing credit card payment of $100.0.
    visa.refund(50.0)
    // Refunding $50.0 to the credit card.
}
```

```
interface Refundable {
    fun refund(amount: Double)
}

abstract class PaymentMethod(val name: String) {
    fun authorize(amount: Double) {
        println("Authorizing payment of $$amount.")
    }

    abstract fun processPayment(amount: Double)
}

class CreditCard(name: String) : PaymentMethod(name), Refundable {
    override fun processPayment(amount: Double) {
        println("Processing credit card payment of $$amount.")
    }

    override fun refund(amount: Double) {
        println("Refunding $$amount to the credit card.")
    }
}

fun main() {
    val visa = CreditCard("Visa")

    visa.authorize(100.0)
    // Authorizing payment of $100.0.
    visa.processPayment(100.0)
    // Processing credit card payment of $100.0.
    visa.refund(50.0)
    // Refunding $50.0 to the credit card.
}
```

Exercise 4

You have a simple messaging app that has some basic functionality, but you want to add some functionality for *smart* messages without significantly duplicating your code.

In the code below, define a class called `SmartMessenger` that inherits from the `Messenger` interface but delegates the implementation to an instance of the `BasicMessenger` class.

In the `SmartMessenger` class, override the `sendMessage()` function to send smart messages. The function must accept a `message` as an input and return a printed statement: `"Sending a smart message: $message"`. In addition, call the `sendMessage()` function from the `BasicMessenger` class and prefix the message with `[smart]`.

> You don't need to rewrite the `receiveMessage()` function in the `SmartMessenger` class.

```
interface Messenger {
    fun sendMessage(message: String)
    fun receiveMessage(): String
}

class BasicMessenger : Messenger {
    override fun sendMessage(message: String) {
        println("Sending message: $message")
    }

    override fun receiveMessage(): String {
        return "You've got a new message!"
    }
}

class SmartMessenger // Write your code here

fun main() {
    val basicMessenger = BasicMessenger()
    val smartMessenger = SmartMessenger(basicMessenger)

    basicMessenger.sendMessage("Hello!")
    // Sending message: Hello!
    println(smartMessenger.receiveMessage())
    // You've got a new message!
    smartMessenger.sendMessage("Hello from SmartMessenger!")
    // Sending a smart message: Hello from SmartMessenger!
    // Sending message: [smart] Hello from SmartMessenger!
}
```

```
interface Messenger {
    fun sendMessage(message: String)
    fun receiveMessage(): String
}

class BasicMessenger : Messenger {
    override fun sendMessage(message: String) {
        println("Sending message: $message")
    }

    override fun receiveMessage(): String {
        return "You've got a new message!"
    }
}

class SmartMessenger(val basicMessenger: BasicMessenger) : Messenger by basicMessenger {
    override fun sendMessage(message: String) {
        println("Sending a smart message: $message")
        basicMessenger.sendMessage("[smart] $message")
    }
}

fun main() {
    val basicMessenger = BasicMessenger()
    val smartMessenger = SmartMessenger(basicMessenger)

    basicMessenger.sendMessage("Hello!")
    // Sending message: Hello!
    println(smartMessenger.receiveMessage())
    // You've got a new message!
    smartMessenger.sendMessage("Hello from SmartMessenger!")
    // Sending a smart message: Hello from SmartMessenger!
    // Sending message: [smart] Hello from SmartMessenger!
}
```

Previous step

Next step

Objects

In this chapter, you'll expand your understanding of classes by exploring object declarations. This knowledge will help you efficiently manage behavior across your projects.

Object declarations

In Kotlin, you can use **object declarations** to declare a class with a single instance. In a sense, you declare the class and create the single instance *at the same time*. Object declarations are useful when you want to create a class to use as a single reference point for your program or to coordinate behavior across a system.

> A class that has only one instance that is easily accessible is called a **singleton**.

Objects in Kotlin are **lazy**, meaning they are created only when accessed. Kotlin also ensures that all objects are created in a thread-safe manner so that you don't have to check this manually.

To create an object declaration, use the `object` keyword:

```
object DoAuth {}
```

Following the name of your `object`, add any properties or member functions within the object body defined by curly braces `{}`.

Objects can't have constructors, so they don't have headers like classes.

For example, let's say that you wanted to create an object called `DoAuth` that is responsible for authentication:

```
object DoAuth {
    fun takeParams(username: String, password: String) {
        println("input Auth parameters = $username:$password")
    }
}

fun main(){
    // The object is created when the takeParams() function is called
    DoAuth.takeParams("coding_ninja", "N1njaC0ding!")
    // input Auth parameters = coding_ninja:N1njaC0ding!
}
```

The object has a member function called `takeParams` that accepts `username` and `password` variables as parameters and prints a string to the console. The `DoAuth` object is only created when the function is called for the first time.

Objects can inherit from classes and interfaces. For example:

```
interface Auth {
    fun takeParams(username: String, password: String)
}

object DoAuth : Auth {
    override fun takeParams(username: String, password: String) {
        println("input Auth parameters = $username:$password")
    }
}
```

Data objects

To make it easier to print the contents of an object declaration, Kotlin has **data** objects. Similar to data classes, which you learned about in the beginner tour, data objects automatically come with additional member functions: `toString()` and `equals()`.

Unlike data classes, data objects do not come automatically with the `copy()` member function because they only have a single instance that can't be copied.

To create a data object, use the same syntax as for object declarations but prefix it with the `data` keyword:

```
data object AppConfig {}
```

For example:

```
data object AppConfig {
    var appName: String = "My Application"
    var version: String = "1.0.0"
}

fun main() {
    println(AppConfig)
    // AppConfig

    println(AppConfig.appName)
    // My Application
}
```

For more information about data objects, see Data objects.

Companion objects

In Kotlin, a class can have an object: a **companion** object. You can only have **one** companion object per class. A companion object is created only when its class is referenced for the first time.

Any properties or functions declared inside a companion object are shared across all class instances.

To create a companion object within a class, use the same syntax for an object declaration but prefix it with the `companion` keyword:

```
companion object Bonger {}
```

A companion object doesn't have to have a name. If you don't define one, the default is `Companion`.

To access any properties or functions of the companion object, reference the class name. For example:

```
class BigBen {
    companion object Bonger {
        fun getBongs(nTimes: Int) {
            repeat(nTimes) { print("BONG ") }
            }
        }
    }

fun main() {
    // Companion object is created when the class is referenced for the
    // first time.
    BigBen.getBongs(12)
    // BONG BONG BONG BONG BONG BONG BONG BONG BONG BONG BONG BONG
}
```

This example creates a class called `BigBen` that contains a companion object called `Bonger`. The companion object has a member function called `getBongs()` that accepts an integer and prints `"BONG"` to the console the same number of times as the integer.

In the `main()` function, the `getBongs()` function is called by referring to the class name. The companion object is created at this point. The `getBongs()` function is called with parameter `12`.

For more information, see Companion objects.

Practice

Exercise 1

You run a coffee shop and have a system for tracking customer orders. Consider the code below and complete the declaration of the second data object so that the following code in the `main()` function runs successfully:

```
interface Order {
    val orderId: String
    val customerName: String
    val orderTotal: Double
}

data object OrderOne: Order {
    override val orderId = "001"
    override val customerName = "Alice"
    override val orderTotal = 15.50
}

data object // Write your code here

fun main() {
    // Print the name of each data object
    println("Order name: $OrderOne")
    // Order name: OrderOne
    println("Order name: $OrderTwo")
    // Order name: OrderTwo

    // Check if the orders are identical
    println("Are the two orders identical? ${OrderOne == OrderTwo}")
    // Are the two orders identical? false

    if (OrderOne == OrderTwo) {
        println("The orders are identical.")
    } else {
        println("The orders are unique.")
        // The orders are unique.
    }

    println("Do the orders have the same customer name? ${OrderOne.customerName ==
OrderTwo.customerName}")
    // Do the orders have the same customer name? false
}
```

```
interface Order {
    val orderId: String
    val customerName: String
    val orderTotal: Double
}

data object OrderOne: Order {
    override val orderId = "001"
    override val customerName = "Alice"
    override val orderTotal = 15.50
}

data object OrderTwo: Order {
    override val orderId = "002"
    override val customerName = "Bob"
    override val orderTotal = 12.75
}

fun main() {
    // Print the name of each data object
    println("Order name: $OrderOne")
    // Order name: OrderOne
    println("Order name: $OrderTwo")
    // Order name: OrderTwo

    // Check if the orders are identical
    println("Are the two orders identical? ${OrderOne == OrderTwo}")
    // Are the two orders identical? false

    if (OrderOne == OrderTwo) {
        println("The orders are identical.")
    } else {
        println("The orders are unique.")
        // The orders are unique.
    }

    println("Do the orders have the same customer name? ${OrderOne.customerName ==
OrderTwo.customerName}")
    // Do the orders have the same customer name? false
}
```

Exercise 2

Create an object declaration that inherits from the `Vehicle` interface to create a unique vehicle type: `FlyingSkateboard`. Implement the `name` property and the `move()` function in your object so that the following code in the `main()` function runs successfully:

```
interface Vehicle {
    val name: String
    fun move(): String
}

object // Write your code here

fun main() {
    println("${FlyingSkateboard.name}: ${FlyingSkateboard.move()}")
    // Flying Skateboard: Glides through the air with a hover engine
    println("${FlyingSkateboard.name}: ${FlyingSkateboard.fly()}")
    // Flying Skateboard: Wooooooooo
}
```

```
interface Vehicle {
    val name: String
    fun move(): String
}

object FlyingSkateboard : Vehicle {
    override val name = "Flying Skateboard"
    override fun move() = "Glides through the air with a hover engine"

   fun fly(): String = "Wooooooooo"
}

fun main() {
    println("${FlyingSkateboard.name}: ${FlyingSkateboard.move()}")
    // Flying Skateboard: Glides through the air with a hover engine
    println("${FlyingSkateboard.name}: ${FlyingSkateboard.fly()}")
    // Flying Skateboard: Wooooooooo
}
```

Exercise 3

You are building a user registration module for an app. You want to keep email validation associated with the `User` class but don't want to create an unnecessary `User` instance if the email address is invalid.

For this exercise, consider an email address valid if it contains both `@` and `.`. Complete the data class so that the following code in the `main()` function runs successfully:

Hint

Add an email validation function in a companion object for the `User` class so that you can call the function directly on `User`.

```
data class User(val name: String, val email: String) {
    // Write your code here
}

fun main() {
    val candidates = listOf(
        Pair("Alice", "alice@example.com"),
        Pair("Bob", "bob2example-com")
    )

    for ((name, email) in candidates) {
        if (User.isValidEmail(email)) {
            val user = User(name, email)
            println("Registered: ${user.name}, ${user.email}")
            // Registered: Alice, alice@example.com
        } else {
            println("Error: '${email}' is not valid. The email should contain '@' and '.'")
            // Error: 'bob2example-com' is not valid. The email should contain '@' and '.'
        }
    }
}
```

```
data class User(val name: String, val email: String) {
    companion object {
        fun isValidEmail(email: String): Boolean =
            email.contains('@') && email.contains('.')
    }
}

fun main() {
    val candidates = listOf(
        Pair("Alice", "alice@example.com"),
        Pair("Bob", "bob2example-com")
    )

    for ((name, email) in candidates) {
        if (User.isValidEmail(email)) {
            val user = User(name, email)
            println("Registered: ${user.name}, ${user.email}")
            // Registered: Alice, alice@example.com
        } else {
            println("Error: '${email}' is not valid. The email should contain '@' and '.'")
            // Error: 'bob2example-com' is not valid. The email should contain '@' and '.'
        }
    }
}
```

As an extension of this exercise, try using functions in companion objects as factory methods to construct instances of a class. For an example and more information about this pattern, see Companion objects.

Previous step

Next step

Open and special classes

In this chapter, you'll learn about open classes, how they work with interfaces, and other special types of classes available in Kotlin.

Open classes

If you can't use interfaces or abstract classes, you can explicitly make a class inheritable by declaring it as **open**. To do this, use the `open` keyword before your class declaration:

```
open class Vehicle(val make: String, val model: String)
```

To create a class that inherits from another, add a colon after your class header followed by a call to the constructor of the parent class that you want to inherit from. In this example, the `Car` class inherits from the `Vehicle` class:

```
open class Vehicle(val make: String, val model: String)

class Car(make: String, model: String, val numberOfDoors: Int) : Vehicle(make, model)

fun main() {
    // Creates an instance of the Car class
    val car = Car("Toyota", "Corolla", 4)

    // Prints the details of the car
    println("Car Info: Make - ${car.make}, Model - ${car.model}, Number of doors -
${car.numberOfDoors}")
    // Car Info: Make - Toyota, Model - Corolla, Number of doors - 4
}
```

Just like when creating a normal class instance, if your class inherits from a parent class, then it must initialize all the parameters declared in the parent class header. So in the example, the `car` instance of the `Car` class initializes the parent class parameters: `make` and `model`.

Overriding inherited behavior

If you want to inherit from a class but change some of the behavior, you can override the inherited behavior.

By default, it's not possible to override a member function or property of a parent class. Just like with abstract classes, you need to add special keywords.

Member functions

To allow a function in the parent class to be overridden, use the `open` keyword before its declaration in the parent class:

```
open fun displayInfo() {}
```

To override an inherited member function, use the `override` keyword before the function declaration in the child class:

```
override fun displayInfo() {}
```

For example:

```
open class Vehicle(val make: String, val model: String) {
    open fun displayInfo() {
        println("Vehicle Info: Make - $make, Model - $model")
    }
}

class Car(make: String, model: String, val numberOfDoors: Int) : Vehicle(make, model) {
    override fun displayInfo() {
        println("Car Info: Make - $make, Model - $model, Number of Doors - $numberOfDoors")
    }
}

fun main() {
    val car1 = Car("Toyota", "Corolla", 4)
    val car2 = Car("Honda", "Civic", 2)

    // Uses the overridden displayInfo() function
    car1.displayInfo()
    // Car Info: Make - Toyota, Model - Corolla, Number of Doors - 4
    car2.displayInfo()
    // Car Info: Make - Honda, Model - Civic, Number of Doors - 2
}
```

This example:

- Creates two instances of the `Car` class that inherit from the `Vehicle` class: `car1` and `car2`.
- Overrides the `displayInfo()` function in the `Car` class to also print the number of doors.
- Calls the overridden `displayInfo()` function on `car1` and `car2` instances.

Properties

In Kotlin, it's not common practice to make a property inheritable by using the `open` keyword and overriding it later. Most of the time, you use an abstract class or an interface where properties are inheritable by default.

Properties inside open classes are accessible by their child class. In general, it's better to access them directly rather than override them with a new property.

For example, let's say that you have a property called `transmissionType` that you want to override later. The syntax for overriding properties is exactly the same as for overriding member functions. You can do this:

```
open class Vehicle(val make: String, val model: String) {
    open val transmissionType: String = "Manual"
}

class Car(make: String, model: String, val numberOfDoors: Int) : Vehicle(make, model) {
    override val transmissionType: String = "Automatic"
}
```

However, this is not good practice. Instead, you can add the property to the constructor of your inheritable class and declare its value when you create the `Car` child class:

```
open class Vehicle(val make: String, val model: String, val transmissionType: String = "Manual")

class Car(make: String, model: String, val numberOfDoors: Int) : Vehicle(make, model, "Automatic")
```

Accessing properties directly, instead of overriding them, leads to simpler and more readable code. By declaring properties once in the parent class and passing their values through the constructor, you eliminate the need for unnecessary overrides in child classes.

For more information about class inheritance and overriding class behavior, see Inheritance.

Open classes and interfaces

You can create a class that inherits a class **and** implements multiple interfaces. In this case, you must declare the parent class first, after the colon, before listing the interfaces:

```
// Define interfaces
interface EcoFriendly {
    val emissionLevel: String
}

interface ElectricVehicle {
    val batteryCapacity: Double
}

// Parent class
open class Vehicle(val make: String, val model: String)

// Child class
open class Car(make: String, model: String, val numberOfDoors: Int) : Vehicle(make, model)

// New class that inherits from Car and implements two interfaces
class ElectricCar(
    make: String,
    model: String,
    numberOfDoors: Int,
    val capacity: Double,
    val emission: String
) : Car(make, model, numberOfDoors), EcoFriendly, ElectricVehicle {
    override val batteryCapacity: Double = capacity
    override val emissionLevel: String = emission
}
```

Special classes

In addition to abstract, open, and data classes, Kotlin has special types of classes designed for various purposes, such as restricting specific behavior or reducing the performance impact of creating small objects.

Sealed classes

There may be times when you want to restrict inheritance. You can do this with sealed classes. Sealed classes are a special type of abstract class. Once you declare that a class is sealed, you can only create child classes from it within the same package. It's not possible to inherit from the sealed class outside of this scope.

> A package is a collection of code with related classes and functions, typically within a directory. To learn more about packages in Kotlin, see Packages and imports.

To create a sealed class, use the `sealed` keyword:

```
sealed class Mammal
```

Sealed classes are particularly useful when combined with a `when` expression. By using a `when` expression, you can define the behavior for all possible child classes. For example:

```
sealed class Mammal(val name: String)

class Cat(val catName: String) : Mammal(catName)
class Human(val humanName: String, val job: String) : Mammal(humanName)

fun greetMammal(mammal: Mammal): String {
    when (mammal) {
        is Human -> return "Hello ${mammal.name}; You're working as a ${mammal.job}"
        is Cat -> return "Hello ${mammal.name}"
    }
}

fun main() {
    println(greetMammal(Cat("Snowy")))
    // Hello Snowy
}
```

In the example:

- There is a sealed class called `Mammal` that has the `name` parameter in the constructor.
- The `Cat` class inherits from the `Mammal` sealed class and uses the `catName` parameter in its own constructor as the `name` parameter from the `Mammal` class.
- The `Human` class inherits from the `Mammal` sealed class and uses the `humanName` parameter in its own constructor as the `name` parameter from the `Mammal` class. It also has the `job` parameter in its constructor.
- The `greetMammal()` function accepts an argument of `Mammal` type and returns a string.
- Within the `greetMammal()` function body, there's a `when` expression that uses the `is` operator to check the type of `mammal` and decide which action to perform.
- The `main()` function calls the `greetMammal()` function with an instance of the `Cat` class and `name` parameter called `Snowy`.



This tour discusses the `is` operator in more detail in the Null safety chapter.

For more information about sealed classes and their recommended use cases, see Sealed classes and interfaces.

Enum classes

Enum classes are useful when you want to represent a finite set of distinct values in a class. An enum class contains enum constants, which are themselves instances of the enum class.

To create an enum class, use the `enum` keyword:

```
enum class State
```

Let's say that you want to create an enum class that contains the different states of a process. Each enum constant must be separated by a comma `,`:

```
enum class State {
    IDLE, RUNNING, FINISHED
}
```

The `State` enum class has enum constants: `IDLE`, `RUNNING`, and `FINISHED`. To access an enum constant, use the class name followed by a `.` and the name of the enum constant:

```
val state = State.RUNNING
```

You can use this enum class with a `when` expression to define the action to take depending on the value of the enum constant:

```
enum class State {
    IDLE, RUNNING, FINISHED
}

fun main() {
    val state = State.RUNNING
    val message = when (state) {
        State.IDLE -> "It's idle"
        State.RUNNING -> "It's running"
        State.FINISHED -> "It's finished"
    }
    println(message)
    // It's running
}
```

Enum classes can have properties and member functions just like normal classes.

For example, let's say you're working with HTML and you want to create an enum class containing some colors. You want each color to have a property, let's call it `rgb`, that contains their RGB value as a hexadecimal. When creating the enum constants, you must initialize it with this property:

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF),
    YELLOW(0xFFFF00)
}
```

Kotlin stores hexadecimals as integers, so the `rgb` property has the `Int` type, not the `String` type.

To add a member function to this class, separate it from the enum constants with a semicolon `;`:

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF),
    YELLOW(0xFFFF00);

    fun containsRed() = (this.rgb and 0xFF0000 != 0)
}

fun main() {
    val red = Color.RED

    // Calls containsRed() function on enum constant
    println(red.containsRed())
    // true

    // Calls containsRed() function on enum constants via class names
    println(Color.BLUE.containsRed())
    // false

    println(Color.YELLOW.containsRed())
    // true
}
```

In this example, the `containsRed()` member function accesses the value of the enum constant's `rgb` property using the `this` keyword and checks if the hexadecimal value contains `FF` as its first bits to return a boolean value.

For more information, see Enum classes.

Inline value classes

Sometimes in your code, you may want to create small objects from classes and use them only briefly. This approach can have a performance impact. Inline value classes are a special type of class that avoids this performance impact. However, they can only contain values.

To create an inline value class, use the `value` keyword and the `@JvmInline` annotation:

```
@JvmInline
value class Email
```

> The `@JvmInline` annotation instructs Kotlin to optimize the code when it is compiled. To learn more, see Annotations.

An inline value class **must** have a single property initialized in the class header.

Let's say that you want to create a class that collects an email address:

```
// The address property is initialized in the class header.
@JvmInline
value class Email(val address: String)

fun sendEmail(email: Email) {
    println("Sending email to ${email.address}")
}

fun main() {
    val myEmail = Email("example@example.com")
    sendEmail(myEmail)
    // Sending email to example@example.com
}
```

In the example:

- `Email` is an inline value class that has one property in the class header: `address`.
- The `sendEmail()` function accepts objects with type `Email` and prints a string to the standard output.
- The `main()` function:
  - Creates an instance of the `Email` class called `myEmail`.
  - Calls the `sendEmail()` function on the `myEmail` object.

By using an inline value class, you make the class inlined and can use it directly in your code without creating an object. This can significantly reduce memory footprint and improve your code's runtime performance.

For more information about inline value classes, see Inline value classes.

Practice

Exercise 1

You manage a delivery service and need a way to track the status of packages. Create a sealed class called `DeliveryStatus`, containing data classes to represent the following statuses: `Pending`, `InTransit`, `Delivered`, `Canceled`. Complete the `DeliveryStatus` class declaration so that the code in the `main()` function runs successfully:

```
sealed class // Write your code here

fun printDeliveryStatus(status: DeliveryStatus) {
    when (status) {
        is DeliveryStatus.Pending -> {
            println("The package is pending pickup from ${status.sender}.")
        }
        is DeliveryStatus.InTransit -> {
            println("The package is in transit and expected to arrive by
${status.estimatedDeliveryDate}.")
        }
        is DeliveryStatus.Delivered -> {
            println("The package was delivered to ${status.recipient} on ${status.deliveryDate}.")
        }
        is DeliveryStatus.Canceled -> {
            println("The delivery was canceled due to: ${status.reason}.")
        }
    }
}

fun main() {
    val status1: DeliveryStatus = DeliveryStatus.Pending("Alice")
    val status2: DeliveryStatus = DeliveryStatus.InTransit("2024-11-20")
    val status3: DeliveryStatus = DeliveryStatus.Delivered("2024-11-18", "Bob")
    val status4: DeliveryStatus = DeliveryStatus.Canceled("Address not found")

    printDeliveryStatus(status1)
    // The package is pending pickup from Alice.
    printDeliveryStatus(status2)
    // The package is in transit and expected to arrive by 2024-11-20.
    printDeliveryStatus(status3)
    // The package was delivered to Bob on 2024-11-18.
    printDeliveryStatus(status4)
    // The delivery was canceled due to: Address not found.
}
```

```
sealed class DeliveryStatus {
    data class Pending(val sender: String) : DeliveryStatus()
    data class InTransit(val estimatedDeliveryDate: String) : DeliveryStatus()
    data class Delivered(val deliveryDate: String, val recipient: String) : DeliveryStatus()
    data class Canceled(val reason: String) : DeliveryStatus()
}

fun printDeliveryStatus(status: DeliveryStatus) {
    when (status) {
        is DeliveryStatus.Pending -> {
            println("The package is pending pickup from ${status.sender}.")
        }
        is DeliveryStatus.InTransit -> {
            println("The package is in transit and expected to arrive by
${status.estimatedDeliveryDate}.")
        }
        is DeliveryStatus.Delivered -> {
            println("The package was delivered to ${status.recipient} on ${status.deliveryDate}.")
        }
        is DeliveryStatus.Canceled -> {
            println("The delivery was canceled due to: ${status.reason}.")
        }
    }
}

fun main() {
    val status1: DeliveryStatus = DeliveryStatus.Pending("Alice")
    val status2: DeliveryStatus = DeliveryStatus.InTransit("2024-11-20")
    val status3: DeliveryStatus = DeliveryStatus.Delivered("2024-11-18", "Bob")
    val status4: DeliveryStatus = DeliveryStatus.Canceled("Address not found")

    printDeliveryStatus(status1)
    // The package is pending pickup from Alice.
    printDeliveryStatus(status2)
    // The package is in transit and expected to arrive by 2024-11-20.
    printDeliveryStatus(status3)
    // The package was delivered to Bob on 2024-11-18.
    printDeliveryStatus(status4)
    // The delivery was canceled due to: Address not found.
}
```

Exercise 2

In your program, you want to be able to handle different statuses and types of errors. You have a sealed class to capture the different statuses which are declared in data classes or objects. Complete the code below by creating an enum class called `Problem` that represents the different problem types: `NETWORK`, `TIMEOUT`, and `UNKNOWN`.

```
sealed class Status {
    data object Loading : Status()
    data class Error(val problem: Problem) : Status() {
        // Write your code here
    }

    data class OK(val data: List<String>) : Status()
}

fun handleStatus(status: Status) {
    when (status) {
        is Status.Loading -> println("Loading...")
        is Status.OK -> println("Data received: ${status.data}")
        is Status.Error -> when (status.problem) {
            Status.Error.Problem.NETWORK -> println("Network issue")
            Status.Error.Problem.TIMEOUT -> println("Request timed out")
            Status.Error.Problem.UNKNOWN -> println("Unknown error occurred")
        }
    }
}

fun main() {
    val status1: Status = Status.Error(Status.Error.Problem.NETWORK)
    val status2: Status = Status.OK(listOf("Data1", "Data2"))

    handleStatus(status1)
    // Network issue
    handleStatus(status2)
    // Data received: [Data1, Data2]
}
```

```
sealed class Status {
    data object Loading : Status()
    data class Error(val problem: Problem) : Status() {
        enum class Problem {
            NETWORK,
            TIMEOUT,
            UNKNOWN
        }
    }

    data class OK(val data: List<String>) : Status()
}

fun handleStatus(status: Status) {
    when (status) {
        is Status.Loading -> println("Loading...")
        is Status.OK -> println("Data received: ${status.data}")
        is Status.Error -> when (status.problem) {
            Status.Error.Problem.NETWORK -> println("Network issue")
            Status.Error.Problem.TIMEOUT -> println("Request timed out")
            Status.Error.Problem.UNKNOWN -> println("Unknown error occurred")
        }
    }
}

fun main() {
    val status1: Status = Status.Error(Status.Error.Problem.NETWORK)
    val status2: Status = Status.OK(listOf("Data1", "Data2"))

    handleStatus(status1)
    // Network issue
    handleStatus(status2)
    // Data received: [Data1, Data2]
}
```

Previous step

Next step

Properties

In the beginner tour, you learned how properties are used to declare characteristics of class instances and how to access them. This chapter digs deeper into how properties work in Kotlin and explores other ways that you can use them in your code.

Backing fields

In Kotlin, properties have default `get()` and `set()` functions, known as property accessors, which handle retrieving and modifying their values. While these default functions are not explicitly visible in the code, the compiler automatically generates them to manage property access behind the scenes. These accessors use a **backing field** to store the actual property value.

Backing fields exist if either of the following is true:

- You use the default `get()` or `set()` functions for the property.
- You try to access the property value in code by using the `field` keyword.

`get()` and `set()` functions are also called getters and setters.

For example, this code has the `category` property that has no custom `get()` or `set()` functions and therefore uses the default implementations:

```
class Contact(val id: Int, var email: String) {
    var category: String = ""
}
```

Under the hood, this is equivalent to this pseudocode:

```
class Contact(val id: Int, var email: String) {
    var category: String = ""
        get() = field
        set(value) {
            field = value
        }
}
```

In this example:

- The `get()` function retrieves the property value from the field: `""`.
- The `set()` function accepts `value` as a parameter and assigns it to the field, where `value` is `""`.

Access to the backing field is useful when you want to add extra logic in your `get()` or `set()` functions without causing an infinite loop. For example, you have a `Person` class with a `name` property:

```
class Person {
    var name: String = ""
}
```

You want to ensure that the first letter of the `name` property is capitalized, so you create a custom `set()` function that uses the `.replaceFirstChar()` and `.uppercase()` extension functions. However, if you refer to the property directly in your `set()` function, you create an infinite loop and see a `StackOverflowError` at runtime:

```
class Person {
    var name: String = ""
        set(value) {
            // This causes a runtime error
            name = value.replaceFirstChar { firstChar -> firstChar.uppercase() }
        }
}

fun main() {
    val person = Person()
    person.name = "kodee"
    println(person.name)
    // Exception in thread "main" java.lang.StackOverflowError
}
```

To fix this, you can use the backing field in your `set()` function instead by referencing it with the `field` keyword:

```
class Person {
    var name: String = ""
        set(value) {
            field = value.replaceFirstChar { firstChar -> firstChar.uppercase() }
        }
}

fun main() {
    val person = Person()
    person.name = "kodee"
    println(person.name)
    // Kodee
}
```

Backing fields are also useful when you want to add logging, send notifications when a property value changes, or use additional logic that compares the old and new property values.

For more information, see Backing fields.

Extension properties

Just like extension functions, there are also extension properties. Extension properties allow you to add new properties to existing classes without modifying their source code. However, extension properties in Kotlin do **not** have backing fields. This means that you need to write the `get()` and `set()` functions yourself. Additionally, the lack of a backing field means that they can't hold any state.

To declare an extension property, write the name of the class that you want to extend followed by a `.` and the name of your property. Just like with normal class properties, you need to declare a type for your property. For example:

```
val String.lastChar: Char
```

Extension properties are most useful when you want a property to contain a computed value without using inheritance. You can think of extension properties working like a function with only one parameter: the receiver.

For example, let's say that you have a data class called `Person` with two properties: `firstName` and `lastName`.

```
data class Person(val firstName: String, val lastName: String)
```

You want to be able to access the person's full name without modifying the `Person` data class or inheriting from it. You can do this by creating an extension property with a custom `get()` function:

```
data class Person(val firstName: String, val lastName: String)

// Extension property to get the full name
val Person.fullName: String
    get() = "$firstName $lastName"

fun main() {
    val person = Person(firstName = "John", lastName = "Doe")

    // Use the extension property
    println(person.fullName)
    // John Doe
}
```

Extension properties can't override existing properties of a class.

Just like with extension functions, the Kotlin standard library uses extension properties widely. For example, see the `lastIndex` property for a `CharSequence`.

Delegated properties

You already learned about delegation in the Classes and interfaces chapter. You can also use delegation with properties to delegate their property accessors to another object. This is useful when you have more complex requirements for storing properties that a simple backing field can't handle, such as storing values in a database table, browser session, or map. Using delegated properties also reduces boilerplate code because the logic for getting and setting your properties is contained only in the object that you delegate to.

The syntax is similar to using delegation with classes but operates on a different level. Declare your property, followed by the `by` keyword and the object you want to delegate to. For example:

```
val displayName: String by Delegate
```

Here, the delegated property `displayName` refers to the `Delegate` object for its property accessors.

Every object you delegate to **must** have a `getValue()` operator function, which Kotlin uses to retrieve the value of the delegated property. If the property is mutable, it must also have a `setValue()` operator function for Kotlin to set its value.

By default, the `getValue()` and `setValue()` functions have the following construction:

```
operator fun getValue(thisRef: Any?, property: KProperty<*>): String {}

operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {}
```

In these functions:

- The `operator` keyword marks these functions as operator functions, enabling them to overload the `get()` and `set()` functions.
- The `thisRef` parameter refers to the object **containing** the delegated property. By default, the type is set to `Any?`, but you may need to declare a more specific type.
- The `property` parameter refers to the property whose value is accessed or changed. You can use this parameter to access information like the property's name or type. By default, the type is set to `KProperty<*>` but you can also use `Any?`. You don't need to worry about changing this in your code.

The `getValue()` function has a return type of `String` by default, but you can adjust this if you want.

The `setValue()` function has an additional parameter `value`, which is used to hold the new value that's assigned to the property.

So, how does this look in practice? Suppose you want to have a computed property, like a user's display name, that is calculated only once because the operation is expensive and your application is performance-sensitive. You can use a delegated property to cache the display name so that it is only computed once but can be accessed anytime without performance impact.

First, you need to create the object to delegate to. In this case, the object will be an instance of the `CachedStringDelegate` class:

```
class CachedStringDelegate {
    var cachedValue: String? = null
}
```

The `cachedValue` property contains the cached value. Within the `CachedStringDelegate` class, add the behavior that you want from the `get()` function of the delegated property to the `getValue()` operator function body:

```
class CachedStringDelegate {
    var cachedValue: String? = null

    operator fun getValue(thisRef: Any?, property: Any?): String {
        if (cachedValue == null) {
            cachedValue = "Default Value"
            println("Computed and cached: $cachedValue")
        } else {
            println("Accessed from cache: $cachedValue")
        }
        return cachedValue ?: "Unknown"
    }
}
```

The `getValue()` function checks whether the `cachedValue` property is `null`. If it is, the function assigns the `"Default value"` and prints a string for logging purposes. If the `cachedValue` property has already been computed, the property isn't `null`. In this case, another string is printed for logging purposes. Finally, the function uses the Elvis operator to return the cached value or `"Unknown"` if the value is `null`.

Now you can delegate the property that you want to cache (`val displayName`) to an instance of the `CachedStringDelegate` class:

```
class CachedStringDelegate {
    var cachedValue: String? = null

    operator fun getValue(thisRef: User, property: Any?): String {
        if (cachedValue == null) {
            cachedValue = "${thisRef.firstName} ${thisRef.lastName}"
            println("Computed and cached: $cachedValue")
        } else {
            println("Accessed from cache: $cachedValue")
        }
        return cachedValue ?: "Unknown"
    }
}

class User(val firstName: String, val lastName: String) {
    val displayName: String by CachedStringDelegate()
}

fun main() {
    val user = User("John", "Doe")

    // First access computes and caches the value
    println(user.displayName)
    // Computed and cached: John Doe
    // John Doe

    // Subsequent accesses retrieve the value from cache
    println(user.displayName)
    // Accessed from cache: John Doe
    // John Doe
}
```

This example:

- Creates a `User` class that has two properties in the header, `firstName`, and `lastName`, and one property in the class body, `displayName`.
- Delegates the `displayName` property to an instance of the `CachedStringDelegate` class.
- Creates an instance of the `User` class called `user`.
- Prints the result of accessing the `displayName` property on the `user` instance.

Note that in the `getValue()` function, the type for the `thisRef` parameter is narrowed from `Any?` type to the object type: `User`. This is so that the compiler can access the `firstName` and `lastName` properties of the `User` class.

Standard delegates

The Kotlin standard library provides some useful delegates for you so you don't have to always create yours from scratch. If you use one of these delegates, you don't need to define `getValue()` and `setValue()` functions because the standard library automatically provides them.

Lazy properties

To initialize a property only when it's first accessed, use a lazy property. The standard library provides the `Lazy` interface for delegation.

To create an instance of the `Lazy` interface, use the `lazy()` function by providing it with a lambda expression to execute when the `get()` function is called for the first time. Any further calls of the `get()` function return the same result that was provided on the first call. Lazy properties use the trailing lambda syntax to pass the lambda expression.

For example:

```
class Database {
    fun connect() {
        println("Connecting to the database...")
    }

    fun query(sql: String): List<String> {
        return listOf("Data1", "Data2", "Data3")
    }
}

val databaseConnection: Database by lazy {
    val db = Database()
    db.connect()
    db
}

fun fetchData() {
    val data = databaseConnection.query("SELECT * FROM data")
    println("Data: $data")
}

fun main() {
    // First time accessing databaseConnection
    fetchData()
    // Connecting to the database...
    // Data: [Data1, Data2, Data3]

    // Subsequent access uses the existing connection
    fetchData()
    // Data: [Data1, Data2, Data3]
}
```

In this example:

- There is a `Database` class with `connect()` and `query()` member functions.
- The `connect()` function prints a string to the console, and the `query()` function accepts an SQL query and returns a list.
- There is a `databaseConnection` property that is a lazy property.
- The lambda expression provided to the `lazy()` function:
  - Creates an instance of the `Database` class.
  - Calls the `connect()` member function on this instance (`db`).
  - Returns the instance.
- There is a `fetchData()` function that:
  - Creates an SQL query by calling the `query()` function on the `databaseConnection` property.
  - Assigns the SQL query to the `data` variable.
  - Prints the `data` variable to the console.

- The `main()` function calls the `fetchData()` function. The first time it is called, the lazy property is initialized. The second time, the same result is returned as the first call.

Lazy properties are useful not only when initialization is resource-intensive but also when a property might not be used in your code. Additionally, lazy properties are thread-safe by default, which is particularly beneficial if you are working in a concurrent environment.

For more information, see Lazy properties.

Observable properties

To monitor whether the value of a property changes, use an observable property. An observable property is useful when you want to detect a change in the property value and use this knowledge to trigger a reaction. The standard library provides the `Delegates` object for delegation.

To create an observable property, you must first import `kotlin.properties.Delegates.observable`. Then, use the `observable()` function and provide it with a lambda expression to execute whenever the property changes. Just like with lazy properties, observable properties use the trailing lambda syntax to pass the lambda expression.

For example:

```
import kotlin.properties.Delegates.observable

class Thermostat {
    var temperature: Double by observable(20.0) { _, old, new →
        if (new > 25) {
            println("Warning: Temperature is too high! ($old°C → $new°C)")
        } else {
            println("Temperature updated: $old°C → $new°C")
        }
    }
}

fun main() {
    val thermostat = Thermostat()
    thermostat.temperature = 22.5
    // Temperature updated: 20.0°C → 22.5°C

    thermostat.temperature = 27.0
    // Warning: Temperature is too high! (22.5°C → 27.0°C)
}
```

In this example:

- There is a `Thermostat` class that contains an observable property: `temperature`.
- The `observable()` function accepts `20.0` as a parameter and uses it to initialize the property.
- The lambda expression provided to the `observable()` function:
  - Has three parameters:
    - `_`, which refers to the property itself.
    - `old`, which is the old value of the property.
    - `new`, which is the new value of the property.
  - Checks if the `new` parameter is greater than `25` and, depending on the result, prints a string to console.
- The `main()` function:

- Creates an instance of the `Thermostat` class called `thermostat`.
- Updates the value of the `temperature` property of the instance to `22.5`, which triggers a print statement with a temperature update.
- Updates the value of the `temperature` property of the instance to `27.0`, which triggers a print statement with a warning.

Observable properties are useful not only for logging and debugging purposes. You can also use them for use cases like updating a UI or to perform additional checks, like verifying the validity of data.

For more information, see Observable properties.

Practice

Exercise 1

You manage an inventory system at a bookstore. The inventory is stored in a list where each item represents the quantity of a specific book. For example, `listOf(3, 0, 7, 12)` means the store has 3 copies of the first book, 0 of the second, 7 of the third, and 12 of the fourth.

Write a function called `findOutOfStockBooks()` that returns a list of indices for all the books that are out of stock.

Hint 1

Use the `indices` extension property from the standard library.

Hint 2

You can use the `buildList()` function to create and manage a list instead of manually creating and returning a mutable list. The `buildList()` function uses a lambda with a receiver, which you learned about in earlier chapters.

```
fun findOutOfStockBooks(inventory: List<Int>): List<Int> {
    // Write your code here
}

fun main() {
    val inventory = listOf(3, 0, 7, 0, 5)
    println(findOutOfStockBooks(inventory))
    // [1, 3]
}
```

```
fun findOutOfStockBooks(inventory: List<Int>): List<Int> {
    val outOfStockIndices = mutableListOf<Int>()
    for (index in inventory.indices) {
        if (inventory[index] == 0) {
            outOfStockIndices.add(index)
        }
    }
    return outOfStockIndices
}

fun main() {
    val inventory = listOf(3, 0, 7, 0, 5)
    println(findOutOfStockBooks(inventory))
    // [1, 3]
}
```

```
fun findOutOfStockBooks(inventory: List<Int>): List<Int> = buildList {
    for (index in inventory.indices) {
        if (inventory[index] == 0) {
            add(index)
        }
    }
}

fun main() {
    val inventory = listOf(3, 0, 7, 0, 5)
    println(findOutOfStockBooks(inventory))
    // [1, 3]
}
```

Exercise 2

You have a travel app that needs to display distances in both kilometers and miles. Create an extension property for the `Double` type called `asMiles` to convert a distance in kilometers to miles:

> The formula to convert kilometers to miles is `miles = kilometers * 0.621371`.

Hint

Remember that extension properties need a custom `get()` function.

```
val // Write your code here

fun main() {
    val distanceKm = 5.0
    println("$distanceKm km is ${distanceKm.asMiles} miles")
    // 5.0 km is 3.106855 miles

    val marathonDistance = 42.195
    println("$marathonDistance km is ${marathonDistance.asMiles} miles")
    // 42.195 km is 26.218757 miles
}
```

```
val Double.asMiles: Double
    get() = this * 0.621371

fun main() {
    val distanceKm = 5.0
    println("$distanceKm km is ${distanceKm.asMiles} miles")
    // 5.0 km is 3.106855 miles

    val marathonDistance = 42.195
    println("$marathonDistance km is ${marathonDistance.asMiles} miles")
    // 42.195 km is 26.218757 miles
}
```

Exercise 3

You have a system health checker that can determine the state of a cloud system. However, the two functions it can run to perform a health check are performance intensive. Use lazy properties to initialize the checks so that the expensive functions are only run when needed:

```
fun checkAppServer(): Boolean {
    println("Performing application server health check...")
    return true
}

fun checkDatabase(): Boolean {
    println("Performing database health check...")
    return false
}

fun main() {
    // Write your code here

    when {
        isAppServerHealthy -> println("Application server is online and healthy")
        isDatabaseHealthy -> println("Database is healthy")
        else -> println("System is offline")
    }
    // Performing application server health check...
    // Application server is online and healthy
}
```

```
fun checkAppServer(): Boolean {
    println("Performing application server health check...")
    return true
}

fun checkDatabase(): Boolean {
    println("Performing database health check...")
    return false
}

fun main() {
    val isAppServerHealthy by lazy { checkAppServer() }
    val isDatabaseHealthy by lazy { checkDatabase() }

    when {
        isAppServerHealthy -> println("Application server is online and healthy")
        isDatabaseHealthy -> println("Database is healthy")
        else -> println("System is offline")
    }
   // Performing application server health check...
   // Application server is online and healthy
}
```

Exercise 4

You're building a simple budget tracker app. The app needs to observe changes to the user's remaining budget and notify them whenever it goes below a certain threshold. You have a `Budget` class that is initialized with a `totalBudget` property that contains the initial budget amount. Within the class, create an observable property called `remainingBudget` that prints:

- A warning when the value is lower than 20% of the initial budget.
- An encouraging message when the budget is increased from the previous value.

```
import kotlin.properties.Delegates.observable

class Budget(val totalBudget: Int) {
    var remainingBudget: Int // Write your code here
}

fun main() {
    val myBudget = Budget(totalBudget = 1000)
    myBudget.remainingBudget = 800
    myBudget.remainingBudget = 150
    // Warning: Your remaining budget (150) is below 20% of your total budget.
    myBudget.remainingBudget = 50
    // Warning: Your remaining budget (50) is below 20% of your total budget.
    myBudget.remainingBudget = 300
    // Good news: Your remaining budget increased to 300.
}
```

```
import kotlin.properties.Delegates.observable

class Budget(val totalBudget: Int) {
    var remainingBudget: Int by observable(totalBudget) { _, oldValue, newValue →
        if (newValue < totalBudget * 0.2) {
            println("Warning: Your remaining budget ($newValue) is below 20% of your total budget.")
        } else if (newValue > oldValue) {
            println("Good news: Your remaining budget increased to $newValue.")
        }
    }
}

fun main() {
    val myBudget = Budget(totalBudget = 1000)
    myBudget.remainingBudget = 800
    myBudget.remainingBudget = 150
    // Warning: Your remaining budget (150) is below 20% of your total budget.
    myBudget.remainingBudget = 50
    // Warning: Your remaining budget (50) is below 20% of your total budget.
    myBudget.remainingBudget = 300
    // Good news: Your remaining budget increased to 300.
}
```

Previous step

Next step

Null safety

In the beginner tour, you learned how to handle `null` values in your code. This chapter covers common use cases for null safety features and how to make the most of them.

Smart casts and safe casts

Kotlin can sometimes infer the type without explicit declaration. When you tell Kotlin to treat a variable or object as if it belongs to a specific type, this process is called **casting**. When a type is automatically cast, like when it's inferred, it's called **smart casting**.

is and !is operators

Before we explore how casting works, let's see how you can check if an object has a certain type. For this, you can use the `is` and `!is` operators with `when` or `if` conditional expressions:

- `is` checks if the object has the type and returns a boolean value.
- `!is` checks if the object **doesn't** have the type and returns a boolean value.

For example:

```
fun printObjectType(obj: Any) {
    when (obj) {
        is Int -> println("It's an Integer with value $obj")
        !is Double -> println("It's NOT a Double")
        else -> println("Unknown type")
    }
}

fun main() {
    val myInt = 42
    val myDouble = 3.14
    val myList = listOf(1, 2, 3)

    // The type is Int
    printObjectType(myInt)
    // It's an Integer with value 42

    // The type is List, so it's NOT a Double.
    printObjectType(myList)
    // It's NOT a Double

    // The type is Double, so the else branch is triggered.
    printObjectType(myDouble)
    // Unknown type
}
```

You've already seen an example of how to use a `when` conditional expression with the `is` and `!is` operators in the Open and other special classes chapter.

as and as? operators

To explicitly *cast* an object to any other type, use the `as` operator. This includes casting from a nullable type to its non-nullable counterpart. If the cast isn't possible, the program crashes **at runtime**. This is why it's called the **unsafe** cast operator.

```
fun main() {
    val a: String? = null
    val b = a as String

    // Triggers an error at runtime
    print(b)
}
```

To explicitly cast an object to a non-nullable type, but return `null` instead of throwing an error on failure, use the `as?` operator. Since the `as?` operator doesn't trigger an error on failure, it is called the **safe** operator.

```
fun main() {
    val a: String? = null
    val b = a as? String

    // Returns null value
    print(b)
    // null
}
```

You can combine the `as?` operator with the Elvis operator `?:` to reduce several lines of code down to one. For example, the following `calculateTotalStringLength()` function calculates the total length of all strings provided in a mixed list:

```
fun calculateTotalStringLength(items: List<Any>): Int {
    var totalLength = 0

    for (item in items) {
        totalLength += if (item is String) {
            item.length
        } else {
            0  // Add 0 for non-String items
        }
    }

    return totalLength
}
```

The example:

- Uses the `totalLength` variable as a counter.
- Uses a `for` loop to loop through every item in the list.
- Uses an `if` and the `is` operator to check if the current item is a string:
  - If it is, the string's length is added to the counter.
  - If it is not, the counter isn't incremented.
- Returns the final value of the `totalLength` variable.

This code can be reduced to:

```
fun calculateTotalStringLength(items: List<Any>): Int {
    return items.sumOf { (it as? String)?.length ?: 0 }
}
```

The example uses the `.sumOf()` extension function and provides a lambda expression that:

- For each item in the list, performs a safe cast to `String` using `as?`.
- Uses a safe call `?.` to access the `length` property if the call doesn't return a `null` value.
- Uses the Elvis operator `?:` to return `0` if the safe call returns a `null` value.

Null values and collections

In Kotlin, working with collections often involves handling `null` values and filtering out unnecessary elements. Kotlin has useful functions that you can use to write clean, efficient, and null-safe code when working with lists, sets, maps, and other types of collections.

To filter `null` values from a list, use the `filterNotNull()` function:

```
fun main() {
    val emails: List<String?> = listOf("alice@example.com", null, "bob@example.com", null,
"carol@example.com")

    val validEmails = emails.filterNotNull()

    println(validEmails)
    // [alice@example.com, bob@example.com, carol@example.com]
}
```

If you want to perform filtering of `null` values directly when creating a list, use the `listOfNotNull()` function:

```
fun main() {
    val serverConfig = mapOf(
        "appConfig.json" to "App Configuration",
        "dbConfig.json" to "Database Configuration"
    )

    val requestedFile = "appConfig.json"
    val configFiles = listOfNotNull(serverConfig[requestedFile])

    println(configFiles)
    // [App Configuration]
}
```

In both of these examples, if all items are `null` values, an empty list is returned.

Kotlin also provides functions that you can use to find values in collections. If a value isn't found, they return `null` values instead of triggering an error:

- `maxOrNull()` finds the highest value. If one doesn't exist, returns a `null` value.
- `minOrNull()` finds the lowest value. If one doesn't exist, returns a `null` value.

For example:

```
fun main() {
    // Temperatures recorded over a week
    val temperatures = listOf(15, 18, 21, 21, 19, 17, 16)

    // Find the highest temperature of the week
    val maxTemperature = temperatures.maxOrNull()
    println("Highest temperature recorded: ${maxTemperature ?: "No data"}")
    // Highest temperature recorded: 21

    // Find the lowest temperature of the week
    val minTemperature = temperatures.minOrNull()
    println("Lowest temperature recorded: ${minTemperature ?: "No data"}")
    // Lowest temperature recorded: 15
}
```

This example uses the Elvis operator `?:` to return a printed statement if the functions return a `null` value.

The `maxOrNull()`, and `minOrNull()` functions are designed to be used with collections that **don't** contain `null` values. Otherwise, you can't tell whether the function couldn't find the desired value or whether it found a `null` value.

You can use the `singleOrNull()` function with a lambda expression to find a single item that matches a condition. If one doesn't exist or there are multiple items that match, the function returns a `null` value:

```
fun main() {
    // Temperatures recorded over a week
    val temperatures = listOf(15, 18, 21, 21, 19, 17, 16)

    // Check if there was exactly one day with 30 degrees
    val singleHotDay = temperatures.singleOrNull{ it == 30 }
    println("Single hot day with 30 degrees: ${singleHotDay ?: "None"}")
    // Single hot day with 30 degrees: None
}
```

The `singleOrNull()` function is designed to be used with collections that **don't** contain `null` values.

Some functions use a lambda expression to transform a collection and return `null` values if they can't fulfill their purpose.

To transform a collection with a lambda expression and return the first value that isn't `null`, use the `firstNotNullOfOrNull()` function. If no such value exists, the function returns a `null` value:

```
fun main() {
    data class User(val name: String?, val age: Int?)

    val users = listOf(
        User(null, 25),
        User("Alice", null),
        User("Bob", 30)
    )

    val firstNonNullName = users.firstNotNullOfOrNull { it.name }
    println(firstNonNullName)
    // Alice
}
```

To use a lambda expression to process each collection item sequentially and create an accumulated value (or return a `null` value if the collection is empty) use the `reduceOrNull()` function:

```
fun main() {
    // Prices of items in a shopping cart
    val itemPrices = listOf(20, 35, 15, 40, 10)

    // Calculate the total price using the reduceOrNull() function
    val totalPrice = itemPrices.reduceOrNull { runningTotal, price -> runningTotal + price }
    println("Total price of items in the cart: ${totalPrice ?: "No items"}")
    // Total price of items in the cart: 120

    val emptyCart = listOf<Int>()
    val emptyTotalPrice = emptyCart.reduceOrNull { runningTotal, price -> runningTotal + price }
    println("Total price of items in the empty cart: ${emptyTotalPrice ?: "No items"}")
    // Total price of items in the empty cart: No items
}
```

This example also uses the Elvis operator `?:` to return a printed statement if the function returns a `null` value.

The `reduceOrNull()` function is designed to be used with collections that **don't** contain `null` values.

Explore Kotlin's standard library to find more functions that you can use to make your code safer.

Early returns and the Elvis operator

In the beginner tour, you learned how to use early returns to stop your function from being processed further than a certain point. You can use the Elvis operator `?:` with an early return to check preconditions in a function. This approach is a great way to keep your code concise because you don't need to use nested checks. The reduced complexity of your code also makes it easier to maintain. For example:

```
data class User(
    val id: Int,
    val name: String,
    // List of friend user IDs
    val friends: List<Int>
)

// Function to get the number of friends for a user
fun getNumberOfFriends(users: Map<Int, User>, userId: Int): Int {
    // Retrieves the user or return -1 if not found
    val user = users[userId] ?: return -1
    // Returns the number of friends
    return user.friends.size
}

fun main() {
    // Creates some sample users
    val user1 = User(1, "Alice", listOf(2, 3))
    val user2 = User(2, "Bob", listOf(1))
    val user3 = User(3, "Charlie", listOf(1))

    // Creates a map of users
    val users = mapOf(1 to user1, 2 to user2, 3 to user3)

    println(getNumberOfFriends(users, 1))
    // 2
    println(getNumberOfFriends(users, 2))
    // 1
    println(getNumberOfFriends(users, 4))
    // -1
}
```

In this example:

- There is a `User` data class that has properties for the user's `id`, `name` and list of friends.
- The `getNumberOfFriends()` function:
  - Accepts a map of `User` instances and a user ID as an integer.
  - Accesses the value of the map of `User` instances with the provided user ID.
  - Uses an Elvis operator to return the function early with the value of `-1` if the map value is a `null` value.
  - Assigns the value found from the map to the `user` variable.
  - Returns the number of friends in the user's friends list by using the `size` property.
- The `main()` function:
  - Creates three `User` instances.

- Creates a map of these `User` instances and assigns them to the `users` variable.
- Calls the `getNumberOfFriends()` function on the `users` variable with values `1` and `2` that returns two friends for `"Alice"` and one friend for `"Bob"`.
- Calls the `getNumberOfFriends()` function on the `users` variable with value `4`, which triggers an early return with a value of `-1`.

You may notice that the code could be more concise without an early return. However, this approach needs multiple safe calls because the `users[userId]` might return a `null` value, making the code slightly harder to read:

```
fun getNumberOfFriends(users: Map<Int, User>, userId: Int): Int {
    // Retrieve the user or return -1 if not found
    return users[userId]?.friends?.size ?: -1
}
```

Although this example checks only one condition with the Elvis operator, you can add multiple checks to cover any critical error paths. Early returns with the Elvis operator prevent your program from doing unnecessary work and make your code safer by stopping as soon as a `null` value or invalid case is detected.

For more information about how you can use `return` in your code, see Returns and jumps.

Practice

Exercise 1

You are developing a notification system for an app where users can enable or disable different types of notifications. Complete the `getNotificationPreferences()` function so that:

1. The `validUser` variable uses the `as?` operator to check if `user` is an instance of the `User` class. If it isn't, return an empty list.
2. The `userName` variable uses the Elvis `?:` operator to ensure that the user's name defaults to `"Guest"` if it is `null`.
3. The final return statement uses the `.takeIf()` function to include email and SMS notification preferences only if they are enabled.
4. The `main()` function runs successfully and prints the expected output.

The `takeIf()` function returns the original value if the given condition is true, otherwise it returns `null`. For example:

```
fun main() {
    // The user is logged in
    val userIsLoggedIn = true
    // The user has an active session
    val hasSession = true

    // Gives access to the dashboard if the user is logged in
    // and has an active session
    val canAccessDashboard = userIsLoggedIn.takeIf { hasSession }

    println(canAccessDashboard ?: "Access denied")
    // true
}
```

```
data class User(val name: String?)

fun getNotificationPreferences(user: Any, emailEnabled: Boolean, smsEnabled: Boolean): List<String> {
    val validUser = // Write your code here
    val userName = // Write your code here

    return listOfNotNull( /* Write your code here */)
}

fun main() {
    val user1 = User("Alice")
    val user2 = User(null)
    val invalidUser = "NotAUser"

    println(getNotificationPreferences(user1, emailEnabled = true, smsEnabled = false))
    // [Email Notifications enabled for Alice]
    println(getNotificationPreferences(user2, emailEnabled = false, smsEnabled = true))
    // [SMS Notifications enabled for Guest]
    println(getNotificationPreferences(invalidUser, emailEnabled = true, smsEnabled = true))
    // []
}
```

```
data class User(val name: String?)

fun getNotificationPreferences(user: Any, emailEnabled: Boolean, smsEnabled: Boolean): List<String> {
    val validUser = user as? User ?: return emptyList()
    val userName = validUser.name ?: "Guest"

    return listOfNotNull(
        "Email Notifications enabled for $userName".takeIf { emailEnabled },
        "SMS Notifications enabled for $userName".takeIf { smsEnabled }
    )
}

fun main() {
    val user1 = User("Alice")
    val user2 = User(null)
    val invalidUser = "NotAUser"

    println(getNotificationPreferences(user1, emailEnabled = true, smsEnabled = false))
    // [Email Notifications enabled for Alice]
    println(getNotificationPreferences(user2, emailEnabled = false, smsEnabled = true))
    // [SMS Notifications enabled for Guest]
    println(getNotificationPreferences(invalidUser, emailEnabled = true, smsEnabled = true))
    // []
}
```

Exercise 2

You are working on a subscription-based streaming service where users can have multiple subscriptions, but **only one can be active at a time**. Complete the `getActiveSubscription()` function so that it uses the `singleOrNull()` function with a predicate to return a `null` value if there is more than one active subscription:

```
data class Subscription(val name: String, val isActive: Boolean)

fun getActiveSubscription(subscriptions: List<Subscription>): Subscription? // Write your code here

fun main() {
    val userWithPremiumPlan = listOf(
        Subscription("Basic Plan", false),
        Subscription("Premium Plan", true)
    )

    val userWithConflictingPlans = listOf(
        Subscription("Basic Plan", true),
        Subscription("Premium Plan", true)
    )

    println(getActiveSubscription(userWithPremiumPlan))
    // Subscription(name=Premium Plan, isActive=true)

    println(getActiveSubscription(userWithConflictingPlans))
    // null
}
```

```
data class Subscription(val name: String, val isActive: Boolean)

fun getActiveSubscription(subscriptions: List<Subscription>): Subscription? {
    return subscriptions.singleOrNull { subscription -> subscription.isActive }
}

fun main() {
    val userWithPremiumPlan = listOf(
        Subscription("Basic Plan", false),
        Subscription("Premium Plan", true)
    )

    val userWithConflictingPlans = listOf(
        Subscription("Basic Plan", true),
        Subscription("Premium Plan", true)
    )

    println(getActiveSubscription(userWithPremiumPlan))
    // Subscription(name=Premium Plan, isActive=true)

    println(getActiveSubscription(userWithConflictingPlans))
    // null
}
```

```
data class Subscription(val name: String, val isActive: Boolean)

fun getActiveSubscription(subscriptions: List<Subscription>): Subscription? =
    subscriptions.singleOrNull { it.isActive }

fun main() {
    val userWithPremiumPlan = listOf(
        Subscription("Basic Plan", false),
        Subscription("Premium Plan", true)
    )

    val userWithConflictingPlans = listOf(
        Subscription("Basic Plan", true),
        Subscription("Premium Plan", true)
    )

    println(getActiveSubscription(userWithPremiumPlan))
    // Subscription(name=Premium Plan, isActive=true)

    println(getActiveSubscription(userWithConflictingPlans))
    // null
}
```

Exercise 3

You are working on a social media platform where users have usernames and account statuses. You want to see the list of currently active usernames. Complete the `getActiveUsernames()` function so that the `mapNotNull()` function has a predicate that returns the username if it is active or a `null` value if it isn't:

```
data class User(val username: String, val isActive: Boolean)

fun getActiveUsernames(users: List<User>): List<String> {
    return users.mapNotNull { /* Write your code here */ }
}

fun main() {
    val allUsers = listOf(
        User("alice123", true),
        User("bob_the_builder", false),
        User("charlie99", true)
    )

    println(getActiveUsernames(allUsers))
    // [alice123, charlie99]
}
```

Just like in Exercise 1, you can use the `takeIf()` function when you check if the user is active.

```
data class User(val username: String, val isActive: Boolean)

fun getActiveUsernames(users: List<User>): List<String> {
    return users.mapNotNull { user ->
        if (user.isActive) user.username else null
    }
}

fun main() {
    val allUsers = listOf(
        User("alice123", true),
        User("bob_the_builder", false),
        User("charlie99", true)
    )

    println(getActiveUsernames(allUsers))
    // [alice123, charlie99]
}
```

```
data class User(val username: String, val isActive: Boolean)

fun getActiveUsernames(users: List<User>): List<String> =
    users.mapNotNull { user -> user.username.takeIf { user.isActive } }

fun main() {
    val allUsers = listOf(
        User("alice123", true),
        User("bob_the_builder", false),
        User("charlie99", true)
    )

    println(getActiveUsernames(allUsers))
    // [alice123, charlie99]
}
```

Exercise 4

You are working on an inventory management system for an e-commerce platform. Before processing a sale, you need to check if the requested quantity of a product is valid based on the available stock.

Complete the `validateStock()` function so that it uses early returns and the Elvis operator (where applicable) to check if:

- The `requested` variable is `null`.
- The `available` variable is `null`.
- The `requested` variable is a negative value.
- The amount in the `requested` variable is higher than in the `available` variable.

In all of the above cases, the function must return early with the value of `-1`.

```
fun validateStock(requested: Int?, available: Int?): Int {
    // Write your code here
}

fun main() {
    println(validateStock(5,10))
    // 5
    println(validateStock(null,10))
    // -1
    println(validateStock(-2,10))
    // -1
}
```

```
fun validateStock(requested: Int?, available: Int?): Int {
    val validRequested = requested ?: return -1
    val validAvailable = available ?: return -1

    if (validRequested < 0) return -1
    if (validRequested > validAvailable) return -1

    return validRequested
}

fun main() {
    println(validateStock(5,10))
    // 5
    println(validateStock(null,10))
    // -1
    println(validateStock(-2,10))
    // -1
}
```

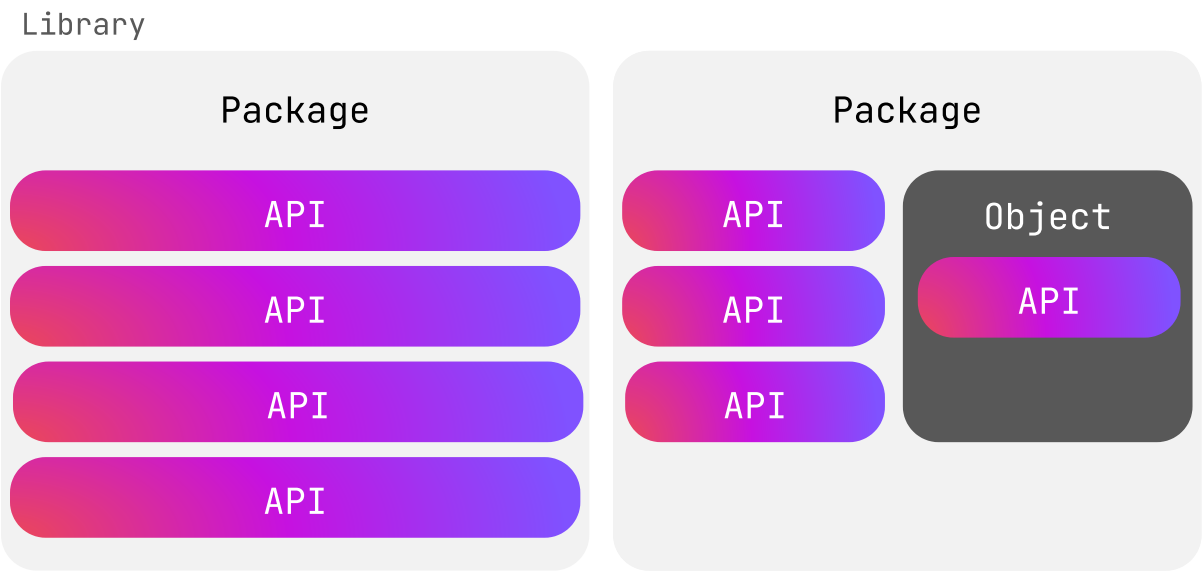
Previous step

Next step

Libraries and APIs

To get the most out of Kotlin, use existing libraries and APIs so you can spend more time coding and less time reinventing the wheel.

Libraries distribute reusable code that simplifies common tasks. Within libraries, there are packages and objects that group related classes, functions, and utilities. Libraries expose APIs (Application Programming Interfaces) as a set of functions, classes, or properties that developers can use in their code.



Kotlin libraries and APIs

Let's explore what's possible with Kotlin.

The standard library

Kotlin has a standard library that provides essential types, functions, collections, and utilities to make your code concise and expressive. A large portion of the standard library (everything in the `kotlin` package) is readily available in any Kotlin file without the need to import it explicitly:

```
fun main() {
    val text = "emosewa si niltoK"

   // Use the reversed() function from the standard library
    val reversedText = text.reversed()

    // Use the print() function from the standard library
    print(reversedText)
    // Kotlin is awesome
}
```

However, some parts of the standard library require an import before you can use them in your code. For example, if you want to use the standard library's time measurement features, you need to import the `kotlin.time` package.

At the top of your file, add the `import` keyword followed by the package that you need:

```
import kotlin.time.*
```

The asterisk `*` is a wildcard import that tells Kotlin to import everything within the package. You can't use the asterisk `*` with companion objects. Instead, you need to explicitly declare the members of a companion object that you want to use.

For example:

```
import kotlin.time.Duration
import kotlin.time.Duration.Companion.hours
import kotlin.time.Duration.Companion.minutes

fun main() {
    val thirtyMinutes: Duration = 30.minutes
    val halfHour: Duration = 0.5.hours
    println(thirtyMinutes == halfHour)
    // true
}
```

This example:

- Imports the `Duration` class and the `hours` and `minutes` extension properties from its companion object.
- Uses the `minutes` property to convert `30` into a `Duration` of 30 minutes.
- Uses the `hours` property to convert `0.5` into a `Duration` of 30 minutes.
- Checks if both durations are equal and prints the result.

Search before you build

Before you decide to write your own code, check the standard library to see if what you're looking for already exists. Here's a list of areas where the standard library already provides a number of classes, functions, and properties for you:

- Collections
- Sequences
- String manipulation
- Time management

To learn more about what else is in the standard library, explore its API reference.

Kotlin libraries

The standard library covers many common use cases, but there are some that it doesn't address. Fortunately, the Kotlin team and the rest of the community have developed a wide range of libraries to complement the standard library. For example, `kotlinx-datetime` helps you manage time across different platforms.

You can find useful libraries on our search platform. To use them, you need to take extra steps, like adding a dependency or plugin. Each library has a GitHub repository with instructions on how to include it in your Kotlin projects.

Once you add the library, you can import any package within it. Here's an example of how to import the `kotlinx-datetime` package to find the current time in New York:

```
import kotlinx.datetime.*

fun main() {
    val now = Clock.System.now() // Get current instant
    println("Current instant: $now")

    val zone = TimeZone.of("America/New_York")
    val localDateTime = now.toLocalDateTime(zone)
    println("Local date-time in NY: $localDateTime")
}
```

This example:

- Imports the `kotlinx.datetime` package.
- Uses the `Clock.System.now()` function to create an instance of the `Instant` class that contains the current time and assigns the result to the `now` variable.
- Prints the current time.
- Uses the `TimeZone.of()` function to find the time zone for New York and assigns the result to the `zone` variable.
- Calls the `.toLocalDateTime()` function on the instance containing the current time, with the New York time zone as an argument.
- Assigns the result to the `localDateTime` variable.
- Prints the time adjusted for the time zone in New York.

To explore the functions and classes that this example uses in more detail, see the API reference.

Opt in to APIs

Library authors may mark certain APIs as requiring opt-in before you can use them in your code. They usually do this when an API is still in development and may change in the future. If you don't opt in, you see warnings or errors like this:

```
This declaration needs opt-in. Its usage should be marked with '@...' or '@OptIn(...)'
```

To opt in, write `@OptIn` followed by parentheses containing the class name that categorizes the API, appended by two colons `::` and `class`.

For example, the `uintArrayOf()` function from the standard library falls under `@ExperimentalUnsignedTypes`, as shown in the API reference:

```
@ExperimentalUnsignedTypes
inline fun uintArrayOf(vararg elements: UInt): UIntArray
```

In your code, the opt-in looks like:

```
@OptIn(ExperimentalUnsignedTypes::class)
```

Here's an example that opts in to use the `uintArrayOf()` function to create an array of unsigned integers and modifies one of its elements:

```
@OptIn(ExperimentalUnsignedTypes::class)
fun main() {
    // Create an unsigned integer array
    val unsignedArray: UIntArray = uintArrayOf(1u, 2u, 3u, 4u, 5u)

    // Modify an element
    unsignedArray[2] = 42u
    println("Updated array: ${unsignedArray.joinToString()}")
    // Updated array: 1, 2, 42, 4, 5
}
```

This is the easiest way to opt in, but there are other ways. To learn more, see Opt-in requirements.

Practice

Exercise 1

You are developing a financial application that helps users calculate the future value of their investments. The formula to calculate compound interest is:

$$A = P \times (1 + \frac{r}{n})^{nt}$$

Where:

- `A` is the amount of money accumulated after interest (principal + interest).
- `P` is the principal amount (the initial investment).
- `r` is the annual interest rate (decimal).
- `n` is the number of times interest is compounded per year.
- `t` is the time the money is invested for (in years).

Update the code to:

1. Import the necessary functions from the `kotlin.math` package.
2. Add a body to the `calculateCompoundInterest()` function that calculates the final amount after applying compound interest.

```
// Write your code here

fun calculateCompoundInterest(P: Double, r: Double, n: Int, t: Int): Double {
    // Write your code here
}

fun main() {
    val principal = 1000.0
    val rate = 0.05
    val timesCompounded = 4
    val years = 5
    val amount = calculateCompoundInterest(principal, rate, timesCompounded, years)
    println("The accumulated amount is: $amount")
    // The accumulated amount is: 1282.0372317085844
}
```

```kotlin
import kotlin.math.*

fun calculateCompoundInterest(P: Double, r: Double, n: Int, t: Int): Double {
    return P * (1 + r / n).pow(n * t)
}

fun main() {
    val principal = 1000.0
    val rate = 0.05
    val timesCompounded = 4
    val years = 5
    val amount = calculateCompoundInterest(principal, rate, timesCompounded, years)
    println("The accumulated amount is: $amount")
    // The accumulated amount is: 1282.0372317085844
}
```

Exercise 2

You want to measure the time it takes to perform multiple data processing tasks in your program. Update the code to add the correct import statements and functions from the `kotlin.time` package:

```kotlin
// Write your code here

fun main() {
    val timeTaken = /* Write your code here */ {
        // Simulate some data processing
        val data = List(1000) { it * 2 }
        val filteredData = data.filter { it % 3 == 0 }

        // Simulate processing the filtered data
        val processedData = filteredData.map { it / 2 }
        println("Processed data")
    }

    println("Time taken: $timeTaken") // e.g. 16 ms
}
```

```kotlin
import kotlin.time.measureTime

fun main() {
    val timeTaken = measureTime {
        // Simulate some data processing
        val data = List(1000) { it * 2 }
        val filteredData = data.filter { it % 3 == 0 }

        // Simulate processing the filtered data
        val processedData = filteredData.map { it / 2 }
        println("Processed data")
    }

    println("Time taken: $timeTaken") // e.g. 16 ms
}
```

Exercise 3

There's a new feature in the standard library available in the latest Kotlin release. You want to try it out, but it requires opt-in. The feature falls under `@ExperimentalStdlibApi`. What should the opt-in look like in your code?

```
@OptIn(ExperimentalStdlibApi::class)
```

What's next?

Congratulations! You've completed the intermediate tour! Would you like to share your feedback about your experience?

As a next step, check out our tutorials for popular Kotlin applications:

- Create a backend application with Spring Boot and Kotlin
- Create a cross-platform application for Android and iOS from scratch and:
  - Share business logic while keeping the UI native
  - Share business logic and UI

- Previous step

What's new in Kotlin 2.4.20

Released: September 7, 2026

Kotlin 2.4.20 is out! Here are the release highlights:

- **Standard library:** Support for coroutine stack trace recovery, new functions for checking the equality and uniqueness of collection elements, and new overloads for `kotlin.test` assertion functions
- **Kotlin/Native:** New Swift export features, improved incremental compilation, and automatically generated `Package.swift` files for SwiftPM dependencies
- **Kotlin/Wasm:** Changes to top-level `require()` calls in `@JsFun` declarations, improved initialization order of companion objects, support for Wasmtime in the Kotlin Gradle plugin, new compilation modes, and reduced binary size for functional interfaces
- **Kotlin/JS:** A new DSL for browser testing, support for exporting suspending lambdas as async functions, improved exportability of data classes
- **Gradle:** Support for Gradle 9.7.0 and improved reporting in the Problems API
- **Build tools API:** Support for new targets: Kotlin/JS, Kotlin/Wasm, and Kotlin metadata
- **Kotlin compiler:** `kotlinr` runner command and a separate native image

For information about the Kotlin release cycle, see Kotlin release process.

Update to Kotlin 2.4.20

The latest version of Kotlin is included in the latest versions of IntelliJ IDEA and Android Studio.

To update to the new Kotlin version, make sure your IDE is updated to the latest version and change the Kotlin version to 2.4.20 in your build scripts.

Stable

New features

Kotlin 2.2.20 introduced experimental support for compiling `when` expressions with `invokedynamic` on JVM 21 and later.

In Kotlin 2.4.20, the feature has now graduated to Stable and is enabled by default.

For more information, see the documentation.

New features

The following pre-stable features are available in this release, including those with Beta, Alpha, and Experimental status:

- Standard library: Support for coroutine stack trace recovery
- Standard library: New functions to check collection elements for equality and uniqueness
- Standard library: New overloads for `kotlin.test` assertion functions
- Kotlin/Native: New Swift export features
- Kotlin/Native: Improved incremental compilation of `klib` artifacts
- Kotlin/JS: A new DSL for browser testing
- Kotlin/JS: Support for exporting suspending lambdas as async functions
- Build tools API: Support for Kotlin/JS, Kotlin/Wasm, and Kotlin metadata
- Kotlin compiler: Separate native image

Standard library

Kotlin 2.4.20 adds support for coroutine stack trace recovery and introduces new functions to check collection elements for equality and uniqueness, as well as new overloads for `kotlin.test` assertion functions.

Experimental

Support for coroutine stack trace recovery

Kotlin 2.4.20 adds the `StackTraceRecoverable` interface to the standard library. This improves integration with the `kotlinx.coroutines` library because it lets you define how to create new exception instances for stack trace recovery without adding a dependency on `kotlinx.coroutines`.

Stack trace recovery helps with debugging when one coroutine throws an exception and another rethrows it. It lets you see where the exception originates and where another coroutine rethrows it.

The `kotlinx.coroutines` library performs stack trace recovery by creating a new exception instance with additional coroutine stack trace information. This happens automatically for exceptions with constructors that take only an exception message, a cause, both, or no arguments.

If an exception constructor has additional required arguments, such as a line number or an error code, implement the `StackTraceRecoverable` interface to define how the `kotlinx.coroutines` library creates a new instance of that exception.

To implement the interface, override the `copyForStackTraceRecovery()` function. In the override, return a new exception instance for stack trace recovery, or `null` if you don't want the `kotlinx.coroutines` library to copy the exception.

> The `StackTraceRecoverable` interface is available on all targets, but the `kotlinx.coroutines` library uses it for stack trace recovery only on the JVM.

These APIs are Experimental and require opt-in with the `@OptIn(ExperimentalStdlibCoroutineSupportApi::class)` annotation.

Here's an example of a custom exception that preserves a `line` property when it creates a new instance for stack trace recovery:

```
import kotlin.coroutines.ExperimentalStdlibCoroutineSupportApi
import kotlin.coroutines.debug.StackTraceRecoverable

@OptIn(ExperimentalStdlibCoroutineSupportApi::class)
class FileEditException
// The implementation requires a private constructor
// to pass the cause to the IllegalStateException constructor
private constructor(
    val line: Int,
    private val detail: String,
    cause: Throwable?,
) : IllegalStateException("When editing line $line: $detail", cause),
    // Implements StackTraceRecoverable for stack trace recovery
    StackTraceRecoverable<FileEditException> {

    constructor(line: Int, detail: String) : this(line, detail, null)

    // Copies the line number and message details
    override fun copyForStackTraceRecovery(): FileEditException =
        FileEditException(line, detail, this)
    }

fun main() {
    val original = FileEditException(15, "Unexpected token")

    // Normally, you don't need to call this function directly unless you're testing its behavior
    // The kotlinx.coroutines library invokes it automatically during stack trace recovery
    val copy = original.copyForStackTraceRecovery()

    println(copy.message)
    // When editing line 15: Unexpected token

    println(copy.cause == original)
    // true
}
```

For more information, see the feature's KEEP.

We would appreciate your feedback in YouTrack.

Experimental

New functions to check collection elements for equality and uniqueness

Before Kotlin 2.4.20, if you wanted to check whether collection elements were all distinct or all equal, you had to use inefficient code patterns.

Kotlin 2.4.20 introduces experimental functions to fill this gap:

Function	Checks
`allDistinct()`	Every value in the collection is unique.
`allDistinctBy()`	Every object has a unique value for the selected property.
`allEqual()`	Every value in the collection is the same.
`allEqualBy()`	Every object has the same value for the selected property.

You can use these functions on collections, sequences, and arrays. They compare elements using structural equality just like other collection operations.

These functions are Experimental and require opt-in with the `@OptIn(ExperimentalStdlibApi::class)` annotation or the `-opt-in=kotlin.ExperimentalStdlibApi` compiler option:

```
@OptIn(ExperimentalStdlibApi::class)
fun main() {
    data class Response(
        val participantId: String,
        val answer: String,
        val responseDate: String
    )

    val responses = listOf(
        Response("P001", "Yes", "2026-07-21"),
        Response("P002", "Maybe", "2026-07-21"),
        Response("P003", "No", "2026-07-21")
    )

    // Checks if all participants gave the same answer
    println(responses.allEqualBy { it.answer })
    // false

    // Checks for duplicate participants
    println(responses.allDistinctBy { it.participantId })
    // true

    // Checks if all responses were submitted on the same date
    println(responses.allEqualBy { it.responseDate })
    // true

    val answers = responses.map { it.answer }

    // Checks if answers are identical
    println(answers.allEqual())
    // false

    // Checks if answers are distinct
    println(answers.allDistinct())
    // true
}
```

We would appreciate your feedback in the KEEP.

Experimental

New overloads for `kotlin.test` assertion functions

Kotlin 2.4.20 adds new overloads for `kotlin.test` assertion functions. They accept a lambda that generates error messages lazily, only when the assertion fails.

Previously, `kotlin.test` assertion functions like `assertTrue()` or `assertEquals()` accepted only pre-formatted error messages built on every assertion, even when the assertion succeeded, and the message was never actually used.

The new overloads align the `kotlin.test` API with JUnit 5 and accept a message supplier through a lambda, instead of a plain string. This improves performance, especially for the Power-assert compiler plugin, which generates detailed error messages for assertions.

The new overloads are available for the following assertion functions:

Function	Description
`assertTrue()` / `assertFalse()`	Checks whether the value is `true` or `false`.
`assertEquals()` / `assertNotEquals()`	Checks whether the values are equal or not.
`assertSame()` / `assertNotSame()`	Checks whether the values refer to the same instance.
`assertIs()` / `assertIsNot()`	Checks whether the value is of the specified type. For `assertIs()`, the function smart-casts it to that type.
`assertNull()`	Checks whether the value is `null`.
`assertContains()`	Checks whether the element (key, character, substring, or regex) is present in the collection, array, sequence, range, or map.
`assertContentEquals()`	Checks whether the collections, sequences, or arrays contain equal elements in the same order.

To use the new API, explicitly opt in with the `@OptIn(ExperimentalKotlinTestApi::class)` annotation:

```
import kotlin.test.ExperimentalKotlinTestApi
import kotlin.test.assertEquals
import kotlin.test.assertTrue

@OptIn(ExperimentalKotlinTestApi::class)
fun testValues(actual: Int, expected: Int, items: List<String>) {
    // The message is built only if the assertion fails
    assertTrue(actual > 0) { "Expected a positive value but got $actual" }

    // Avoids formatting the list unless the assertion fails
    assertEquals(expected, actual) { "Unexpected value for items: ${items.joinToString()}" }
}
```

For more information, see the feature's KEEP.

Kotlin/Native

Kotlin 2.4.20 brings automatic generation of `Package.swift` files for SwiftPM dependencies in Kotlin Multiplatform projects, new Swift export features, including support for sealed classes and cross-language inheritance, and improved incremental compilation.

Generated `Package.swift` for SwiftPM dependencies

When exporting an XCFramework that depends on SwiftPM packages, you must publish the resulting SwiftPM package for it to resolve correctly. To help with this, the `assembleSharedXCFramework` Gradle task now generates a `Package.swift` file to be distributed along with the XCFramework.

For details, see the SwiftPM export page.

New Swift export features

Sealed classes

Kotlin 2.4.20 adds support for sealed classes and interfaces to Swift export.

Previously, you had to write a `default` case for every `switch` statement over a sealed type. Now, sealed hierarchies defined in Kotlin are mapped to Swift enums, enabling exhaustive `switch` statements with full autocompletion in Xcode.

Swift export generates a `sealedType()` method on each sealed type. This method returns a Swift enum whose cases match the direct subclasses of the sealed hierarchy. You can nest these calls to match deeper levels of the hierarchy.

For example, declare a sealed interface with a class hierarchy in Kotlin:

```
// Kotlin
sealed interface Shape

class Circle : Shape {
    override fun toString(): String = "Circle"
}

class Rectangle : Shape {
    override fun toString(): String = "Rectangle"
}

fun createCircle(): Shape = Circle()
```

On the Swift side, you can use an exhaustive `switch` without a `default` case:

```
// Swift
let shape = createCircle()

let name = switch shape.sealedType() {
    case let .circle(type): "It's a \(type.value)"
    case let .rectangle(type): "It's a \(type.value)"
}
// name == "It's a Circle"
```

Because the `switch` is exhaustive, the compiler warns you if a new subclass is added to the sealed hierarchy, so you can handle it immediately instead of relying on a `default` case.

Cross-language inheritance in Swift export

Kotlin 2.4.20 introduces cross-language inheritance support in Swift export.

A common use case for this feature is the reverse import pattern, where you define a contract in Kotlin and provide platform-specific implementations on the Swift side. This is especially useful when you need to use pure Swift libraries that can't be directly imported into Kotlin.

To implement the pattern, declare a Kotlin superclass for the Swift implementation to inherit from and a Kotlin interface. Then implement this interface in Swift and pass the Swift object to Kotlin functions that accept that interface. For example, for the CryptoKit library:

1. On the Kotlin side, declare an interface, a function that accepts it, and an `open` base class:

```
// Kotlin
interface CryptoProvider {
    fun hashMD5(input: String): String
}

fun processHash(provider: CryptoProvider, input: String): String = provider.hashMD5(input)

open class SwiftBase
```

2. On the Swift side, inherit from the exported `SwiftBase` class, implement the interface using a pure Swift library, and pass the object back to Kotlin:

```
// Swift
import CryptoKit

final class IosCryptoProvider: SwiftBase, CryptoProvider {
    func hashMD5(input: String) -> String {
        guard let data = input.data(using: .utf8) else { return "failed" }
        return Insecure.MD5.hash(data: data).description
    }
}

let provider = IosCryptoProvider()

// Calls the Kotlin function, which calls hashMD5() back in Swift
print(processHash(provider: provider, input: "Hello, world!"))
```

When Kotlin receives a Swift object, it treats it like an implementation of a regular interface, calling the Swift code directly.

For more details on Swift export, see our documentation.

Beta

Improved incremental compilation of `klib` artifacts

Kotlin 2.4.20 brings stabilization improvements to incremental compilation of `klib` artifacts, which is now in Beta.

This optimization was first introduced in Kotlin 1.9.20 and proved to drastically reduce compilation time for debug builds. Since then, we've fixed a number of bugs and improved performance.

To try out incremental compilation, add the following option to your `gradle.properties` file:

```
kotlin.incremental.native=true
```

We're actively collecting feedback and planning to enable incremental compilation by default for all projects in the next Kotlin releases. If you encounter any issues, please report them to our issue tracker.

Kotlin/Wasm

Kotlin 2.4.20 changes how Kotlin/Wasm handles top-level `require()` calls in `@JsFun` declarations, aligns companion object initialization order with JVM behavior, reduces binary size for functional interfaces, introduces new compilation modes, and adds support for Wasmtime as a runtime for the `wasmWasi` target in the Kotlin Gradle plugin.

Changes to top-level `require()` calls in `@JsFun` declarations

Kotlin/Wasm now reports an error when a `@JsFun` declaration uses the top-level `require()` function.

Previously, the compiler generated a `require` variable in the `import-object.mjs` file, allowing `@JsFun` declarations to call `require()`.

This behavior unintentionally exposed a compiler implementation detail. To support migration away from it, Kotlin/Wasm removes this generated `require` declaration, and the compiler now reports errors for such calls. For example:

```
// Reports an error
@JsFun("(mod) => require(mod)")
external fun loadModule(mod: String): JsAny
```

To prepare for this change, replace top-level `require()` calls in `@JsFun` declarations with the `@JsModule` annotation:

```
@JsModule("module")
external val module: Module

external interface Module {
    // Defines the expected module members
}
```

For dynamic module loading, use the `import()` expression instead. Add the `/* webpackIgnore: true */` magic comment to prevent webpack from parsing the dynamic import:

```
@JsFun("""
    ((module) => () => module)(
        await import(/* webpackIgnore: true */ "module")
    )
""")
private external fun loadModuleDynamically(): JsAny?
```

You can also use the `import()` expression conditionally. For example, you can load a module only when running in Node.js:

```
@JsFun("""
    ((module) => () => module)(
        ((typeof process !== "undefined") && (process.release.name === "node"))
            ? await import(/* webpackIgnore: true */ "module")
            : null
    )
""")
private external fun loadNodeModule(): JsAny?
```

If your project relies on dependencies that require a top-level `require()` function, add it as a property of `globalThis` as a workaround:

```
@JsFun("""
    ((module) => {
        globalThis.require = module.default.createRequire(import.meta.url)
        return () => {}
    })(await import("node:module"))
""")
external fun defineRequire()
```

If you run into any issues, share your feedback in our issue tracker.

Improved companion object initialization order

Kotlin/Wasm now initializes superclass companion objects before subclass companion objects, matching the JVM behavior. Previously, the initialization could be reversed, leading to inconsistent behavior across platforms.

The update improves cross-platform consistency and reduces platform-specific differences in class initialization behavior. It also enables correct handling of companion object initialization in deeper inheritance hierarchies, including cases where intermediate classes don't declare companion objects.

Support for Wasmtime in the Kotlin Gradle plugin

Kotlin 2.4.20 introduces support for Wasmtime as a runtime for the `wasmWasi` target in the Kotlin Gradle plugin.

Previously, the `wasmWasi` target supported only the Node.js runtime, which required a JavaScript bootstrap to run WASI applications. With Wasmtime support, you can now run Kotlin/Wasm applications on a standalone WebAssembly runtime.

To use Wasmtime as the runtime for the `wasmWasi` target, add `wasmtime()` to your Gradle build file:

```
kotlin {
    wasmWasi {
        wasmtime()
    }
}
```

We would appreciate your feedback in YouTrack.

New compilation modes

Kotlin 2.4.20 adds support for selecting a Kotlin/Wasm compilation mode, including new multi-module modes. Previously, the compiler used the monolith compilation mode, which compiles the project and its dependencies together and generates a single binary. This lets the compiler perform dead code elimination and produce the smallest output.

You can now select one of the following compilation modes:

Compilation mode	Compilation	Output	Optimization behavior
`monolith` (default)	Compiles the project and its dependencies together.	A single binary	Removes unreachable declarations and applies optimizations for the entire program, including dependencies.

Compilation mode	Compilation	Output	Optimization behavior
`multimodule-open-world`	Compiles each module independently and recompiles only modules that change.	A separate, independent binary for each module	Doesn't apply cross-module optimizations, which results in larger binaries.
`multimodule-closed-world`	Processes all modules in one invocation and recompiles only modules that change.	Separate binaries that depend on each other	Removes unreachable declarations but optimizes each Wasm binary independently.

To select a compilation mode, add the `kotlin.wasm.compilationMode` property to your `gradle.properties` file:

```
kotlin.wasm.compilationMode=multimodule-open-world
```

You can also configure Kotlin/Wasm to use closed-world multi-module compilation for development builds and monolith compilation for production builds. This reduces recompilation time during development and produces the smallest output for production builds.

To use this configuration, add the following property to your `gradle.properties` file:

```
kotlin.wasm.compilationMode=multimodule-closed-world-only-in-dev
```

We would appreciate your feedback in YouTrack.

Reduced binary size for lambdas and functional interfaces

Kotlin 2.4.20 changes how Kotlin/Wasm compiles lambdas and functional interfaces. Instead of generating separate anonymous classes, the compiler now generates functions and uses shared base classes.

Tests with the KotlinConf application show that this change reduces Wasm binary size by approximately 5–10%.

Because the change introduces more dynamic calls, it may affect runtime performance. If you experience any issues, report them in our issue tracker.

Kotlin/JS

Kotlin 2.4.20 improves exportability of data classes, introduces a new experimental DSL for browser testing, and adds support for exporting suspending lambdas as JavaScript async functions.

Consistent exportability of synthetic functions on exported data classes

Kotlin 2.4.20 fixes an issue which prevented the `@JsExport.Ignore` annotation from being properly applied to data class properties.

Previously, when you marked a data class with the `@JsExport` annotation, the compiler still reported warnings about the data class exportability because of the automatically generated `copy()` and `componentN()` functions. This happened even if the constructor and the properties were explicitly marked as ignored with `@JsExport.Ignore`.

For example, consider a `Session` data class exported to JavaScript that also has a reference to an internal `DatabaseConnection` type that isn't meant to be exported:

```
// Kotlin
// An internal type that isn't exported to JavaScript
class DatabaseConnection

@JsExport
data class Session @JsExport.Ignore constructor(
    val userId: String,
    @JsExport.Ignore val connection: DatabaseConnection,
)
```

Now that the issue is fixed, the compiler accounts for `@JsExport.Ignore` annotations, so `Session`'s synthetic `copy()` and `componentN()` functions no longer trigger warnings about the non-exported type `DatabaseConnection`. This aligns with the visibility rules introduced by the `@ConsistentCopyVisibility` and `@ExposedCopyVisibility` annotations.

Experimental

A new DSL for browser testing

Kotlin 2.4.20 introduces a new experimental DSL for running Kotlin/JS tests in a browser environment.

Currently, the Kotlin Gradle plugin uses Karma as a browser launcher to run JavaScript tests across different browsers. The Karma project has been deprecated for two years now, which has led us to explore alternative ways to support browser testing.

The new DSL is intended to replace Karma as a manager of different tools under the hood and includes:

- Playwright as a browser driver and a distribution manager that supports the Chromium, Firefox, and WebKit (Safari) browser engines.
- Mocha as a test runner.
- webpack as a bundler (will be replaced with Vite in future releases).

To try out the new DSL for browser testing, add the opt-in `test {}` block inside `browser {}` for your Kotlin/JS target:

```
import org.jetbrains.kotlin.gradle.ExperimentalJsTestDsl
import kotlin.time.Duration.Companion.seconds

kotlin {
    js {
        browser {
            // Add and configure the new test {} block
            @OptIn(ExperimentalJsTestDsl::class)
            test {
                // Configure default timeout for all runners
                timeout = 2.seconds
                // Configure headless mode using Gradle providers
                headless = providers
                    .environmentVariable("IS_IN_CI")
                    .map { it.toBoolean() }
                    .orElse(false)
                // Enable and configure Chromium test runner
                chromium {
                    // Override the common timeout option
                    timeout = 5.seconds
                    // Add extra launch arguments
                    launchArgs.add("--no-sandbox")
                }
                // Enable Firefox test runner
                firefox()
                // Enable WebKit test runner
                webkit()
                // Enable and configure an additional WebKit test runner
                webkit("noheadless") {
                    // Set up custom options
                    headless = false
                }
            }
        }
    }
}
```

The new DSL for browser testing is in active development. We would appreciate your feedback in YouTrack.

For more information, see Run tests in Kotlin/JS.

Experimental

Support for exporting suspending lambdas as async functions

With Kotlin 2.4.20, you can now export suspending lambda expressions as JavaScript `async` functions.

Previously, there was no way to export declarations containing suspending lambdas from Kotlin/JS libraries. Now the Kotlin compiler automatically handles the bridging between Kotlin's `suspend` functions and JavaScript's native `async` / `await` model, which is useful for mixed Kotlin/TypeScript codebases.

To enable this feature, add the following compiler option to your `build.gradle.kts` file:

```
kotlin {
    js {
        compilations.all {
            compileTaskProvider.configure {
                compilerOptions {
                    freeCompilerArgs.add("-Xsuspend-lambda-exporting")
                }
            }
        }
    }
}
```

Then, mark the relevant declarations with `@JsExport`:

```
// Kotlin
@JsExport
class TaskRunner {
    suspend fun runTask(task: suspend () -> String): String {
        return task()
    }
}
```

From the TypeScript side, the suspending lambda appears as a regular `async` function:

```
// TypeScript
import { TaskRunner } from "..."

const runner = new TaskRunner();
const result = await runner.runTask(async () => "done");
console.log(result); // "done"
```

For more information on the `@JsExport` annotation, see our documentation.

Gradle

Kotlin 2.4.20 is fully compatible with Gradle 7.6.3 through 9.7.0. You can also use Gradle versions up to the latest Gradle release. However, be aware that doing so may result in deprecation warnings, and some new Gradle features might not work.

Kotlin 2.4.20 also comes with an improved integration with the Problems API.

Improved reporting in Problems API

Kotlin 2.2.0 was the first release in which the Kotlin Gradle Plugin (KGP) integrated with Gradle's Problems API. Kotlin 2.4.0 added support for writing compiler messages to the Problems API for Kotlin/JVM.

Kotlin 2.4.20 adds compiler diagnostic IDs to the information that the compiler passes to the Problems API. It also groups diagnostics by these IDs, making it easier to identify the source of compilation problems.

Starting with Gradle 8.6, the KGP enables this integration by default. As the API is still evolving, use the most recent Gradle version to benefit from the latest improvements.

Build tools API

Kotlin 2.4.20 adds experimental support for Kotlin/JS, Kotlin/Wasm, and Kotlin metadata to the build tools API.

Experimental

Support for Kotlin/JS, Kotlin/Wasm, and Kotlin metadata

In Kotlin 2.2.0, the build tools API (BTA) became available for Kotlin/JVM. Kotlin 2.4.20 takes the next step toward BTA stabilization by adding support for new targets: Kotlin/JS, Kotlin/Wasm, and Kotlin metadata.

This makes the Kotlin Gradle plugin interact with the compiler more consistently. In some cases, you can also benefit from faster, more stable compilation.

The BTA is a universal API that acts as an abstraction layer between build systems and the Kotlin compiler ecosystem. It helps support Kotlin features and compatibility with the Kotlin compiler in available build tools.

In Kotlin 2.4.20, BTA is available as an opt-in for the new targets. To try it out, add the corresponding properties to your `gradle.properties` file:

```
kotlin.wasm.runViaBuildToolsApi=true
kotlin.js.runViaBuildToolsApi=true
kotlin.metadata.runViaBuildToolsApi=true
```

Starting with Kotlin 2.5.0, we plan to enable BTA in Kotlin/JS, Kotlin/Wasm, and Kotlin metadata by default.

If you're curious about the BTA proposal or want to share your feedback, see this KEEP.

Kotlin compiler

Kotlin 2.4.20 includes an update about the changed Kotlin runner command, `kotlinr`, and introduces an experimental Kotlin compiler native image.

Changed the Kotlin runner command from `kotlin` to `kotlinr`

The `kotlinr` command replaces `kotlin` as the Kotlin runner command to avoid a naming conflict with the `kotlin` command in the Kotlin Toolchain. The Kotlin runner also warns you when you use the `kotlin` command and recommends `kotlinr` instead.

Experimental

Native image

Kotlin 2.4.20 features the first Experimental release of the Kotlin compiler native image. The native image provides a drop-in replacement for the standard `kotlinc` command-line tool, while offering faster startup time and higher performance.

To try out the native image, download the build from GitHub Releases.

The native image also bundles the following compiler plugins you can use with the `-Xplugin` or `-Xcompiler-plugin` CLI options:

- Serialization
- Compose compiler
- All-open
- `no-arg`
- SAM with receiver
- Assignment

- Lombok
- Power-assert

For more information on the Kotlin compiler native image, see its README.

Breaking changes and deprecations

This section highlights important breaking changes and deprecations. For a complete overview, see our Compatibility guide.

- Since Apple is dropping support for its 32-bit watchOS targets, the `watchosArm32` Kotlin/Native target is now deprecated. It's planned for removal in Kotlin 2.5.0 to ensure compatibility with Xcode 27.
- Starting with Kotlin 2.4.20, the Kotlin/Native compiler prohibits AtomicFU atomic operations inside a `public` inline function or inside an `internal` inline function called from another file.
- Kotlin 2.4.20 updates the npm dependency for webpack to 5.108.1. This can affect your project in two ways:
  - webpack has moved its built-in minimizer dependency from `terser-webpack-plugin` to the broader `minimizer-webpack-plugin`. Terser remains the default JavaScript minimizer, but if your project configures or depends on `terser-webpack-plugin` directly, you may need to update its configuration.
  - webpack no longer ignores `import.meta` when determining a JavaScript file's module type. If `import.meta` is present, webpack treats the file as an ES module, which can break files that also use CommonJS constructs. For Kotlin/JS, you can configure the target to use ES modules with the `useEsModules()` Gradle DSL. Kotlin/Wasm should work in most cases without additional configuration. If you encounter an `import.meta` error with Kotlin/Wasm, check whether your project's sources or a direct or transitive dependency uses `import.meta`. Update your own code as needed. If a dependency causes the issue, update it to a compatible version if one is available, or report the issue to the library maintainers.
- Starting with Kotlin 2.4.20, Kotlin/Wasm deprecates the generated JavaScript `wasmExports` API. The compiler prohibits access to all exports except `wasmExports.memory`, which remains temporarily available with a warning. Use the `kotlin.wasm.unsafe.wasmMemory` property to access the module's `WebAssembly.Memory` object.

Documentation updates

Since the last release, we've created new pages and tutorials for the Kotlin ecosystem documentation and revamped existing ones:

- Configure an iOS delivery pipeline – Set up continuous delivery for a Kotlin Multiplatform iOS app with TeamCity.
- Compose Multiplatform updates:
  - Popups – Learn how to create and configure popups in Compose Multiplatform.
  - Window and dialog API v2 – Explore the new API for managing desktop windows and dialogs in Compose Multiplatform.
  - Tray and notifications – Learn how to add an application icon to the system tray and send system notifications in Compose Multiplatform for desktop.
  - Menu bar – Learn how to create a menu bar for a specific window in Compose Multiplatform for desktop.

    - Drag and drop – Handle platform-specific data when implementing drag and drop in Compose Multiplatform.
    - UIKit alternative for Liquid Glass – Explore an alternative approach to Liquid Glass that uses UIKit navigation instead of SwiftUI.
    - MCP server for AI agents – Learn how to use the MCP server in Compose Hot Reload to connect AI agents to your development workflow.
- Caching with Spring – Learn how to add caching to a Spring application with new Kotlin examples.
- Exposed IntelliJ IDEA plugin – Learn how to work with Exposed in IntelliJ IDEA using code completion, database-aware inspections, and live templates.
- Kotlin serialization – Learn how to serialize Kotlin data, customize JSON structure and type representation, and work with more advanced serialization scenarios.
- Flow and Flow operators – Learn how to create and collect cold and hot flows, handle exceptions, and use a wide range of flow operators.
- Debug coroutines – Learn how to debug coroutines on the JVM using debug mode, stack trace recovery, and the debug agent.
- Lincheck – Learn how model checking works in Lincheck, how to use operation execution options, and how to verify the test results.
- kapt compiler plugin – Learn how to configure the kapt compiler plugin in Gradle, Maven, and the command-line compiler.
- Code quality tools in Kotlin projects – Explore tools for analyzing JVM bytecode and Kotlin code.
- Power-assert plugin with Maven – Learn how to use the Power-assert plugin to get more detailed test failure messages.
- Multiple-round processing with KSP – Explore how KSP works across multiple processing rounds, including generated files, deferred symbols, and validation.
- Non-denotable types – Learn about platform types, captured types, and intersection types in Kotlin.
- Type aliases – Learn about type alias scope and visibility.
- This expressions – Learn how implicit `this` is resolved and when to use `this` explicitly to refer to a receiver.
- Strings – Learn about string templates, common string operations, building strings, and type conversion.
- Packages and imports – Learn how to organize Kotlin code using packages and imports.

What's new in Kotlin 2.4.0

Released: July 14, 2026

The Kotlin 2.4.0 release is out! Here are the main highlights:

- **Language:** Stable context parameters, explicit backing fields, and multiple features for annotation use-site targets
- **Standard library:** Stabilized support for the UUID API and support for checking sorted order
- **Kotlin/JVM:** Support for Java 26 and annotations in metadata enabled by default

- **Kotlin/Native:** Support for Swift packages as dependencies, updates on Swift export, and the CMS GC enabled by default
- **Kotlin/Wasm:** Incremental compilation enabled by default and support for WebAssembly Component Model
- **Kotlin/JS**: Support for value class export and ES2015 features in JS code inlining
- **Gradle:** Compatibility with Gradle 9.5.0
- **Maven:** Automatic alignment between Java and JVM target versions
- **Kotlin compiler:** More consistent inline function behavior during `.klib` compilation

You can also find an overview of the updates in this video:

Watch video online.

For information about the Kotlin release cycle, see the Kotlin release process.

Update to Kotlin 2.4.0

The latest version of Kotlin is included in the latest versions of IntelliJ IDEA and Android Studio.

To update to the new Kotlin version, make sure your IDE is updated to the latest version and change the Kotlin version to 2.4.0 in your build scripts.

Stable

New features

In previous Kotlin releases, several new features were introduced as Experimental. The following features have now graduated to Stable in Kotlin 2.4.0, so you no longer need to opt in to use them:

- Context parameters, except for context arguments and callable references
- `@all` meta-target for properties
- New defaulting rules for use-site annotation targets
- Explicit backing fields
- Stable UUID API in the common Kotlin standard library
- New API for converting unsigned integers to `BigInteger` on the JVM
- Support for checking sorted order
- Support for value class export to JavaScript/TypeScript
- Support for ES2015 features when inlining JS code
- Maven: Automatic alignment between Java and JVM target versions
- Support for Maven Toolchains

New features

- Explicit context arguments for context parameters

- Support for collection literals
- Improved compile-time constants
- Improved unused result checks for higher-order functions
- New `@IntroducedAt` annotation to generate version-based overloads for optional parameters
- New map fallback functions to distinguish `null` values and missing keys
- Swift package import
- Swift export goes Alpha with improved concurrency support
- Support for the WebAssembly Component Model

Language

Kotlin 2.4.0 promotes context parameters, explicit backing fields, and annotation use-site targets features to Stable. This release also introduces explicit context arguments for context parameters.

Stable features

Kotlin 2.2.0 and 2.3.0 introduced a few language features as Experimental. We're happy to announce that the following language features are now Stable in this release:

- Context parameters, except for context arguments and callable references
- `@all` meta-target for properties
- New defaulting rules for use-site annotation targets
- Explicit backing fields

See the full list of Kotlin language design features and proposals.

No more deprecation warnings on the last segments of imports

In previous Kotlin versions, when a deprecated class was imported, the deprecation error was reported at the call site as well as at the import directive itself. As there's no way to suppress deprecation errors on imports, you may have worked around this by suppressing deprecation reports for the entire file or by using star imports.

Since reporting the deprecation on the import of a called symbol isn't useful in most cases, Kotlin 2.4.0 doesn't issue a warning when the deprecated symbol is referenced in the last segment of the import directive.

For more information, see KT-30155.

Experimental

Explicit context arguments for context parameters

Kotlin 2.4.0 introduces explicit context arguments for context parameters.

Kotlin 2.3.20 changed the overload resolution for context parameters. As a result, calls to overloads that differ only by context parameters can become ambiguous.

You can now resolve this ambiguity by passing an explicit context argument at the call site.

Here's an example:

```
class EmailSender
class SmsSender

context(emailSender: EmailSender)
fun sendNotification() {
    println("Sent email notification")
}

context(smsSender: SmsSender)
fun sendNotification() {
    println("Sent SMS notification")
}

context(defaultEmailSender: EmailSender, defaultSmsSender: SmsSender)
fun notifyUser() {

    // Selects the overload with the EmailSender context parameter
    sendNotification(emailSender = defaultEmailSender)

    // Selects the overload with the SmsSender context parameter
    sendNotification(smsSender = defaultSmsSender)
}
```

You can also use explicit context arguments instead of the `context()` function to reduce nesting and make some calls easier to read. If you need to use the same context arguments in multiple calls, use the `context()` function instead.

This feature is Experimental. To opt in, add the following compiler option to your build file:

Gradle

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xexplicit-context-arguments")
    }
}
```

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-Xexplicit-context-arguments</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

For more information, see the feature's KEEP.

Experimental

Support for collection literals

Kotlin 2.4.0 introduces experimental support for collection literals. You can now create collections in a simpler and more concise way using brackets `[]`.

For example:

```
fun main() {
    // Mutable list with explicit type declaration
    // val shapes: MutableList<String> = mutableListOf("triangle", "square", "circle")

    // Mutable list with brackets syntax
    val shapes: MutableList<String> = ["triangle", "square", "circle"]
    println(shapes)
    // [triangle, square, circle]
}
```

Currently, collection literals can't be used to construct collections defined in Java. For more information, see KT-80494.

If the compiler doesn't have enough information to infer the collection type, it defaults to the `List` type:

```
fun main() {
    val fruit = ["apple", "banana", "cherry"]

    println(fruit)
    // [apple, banana, cherry]
}
```

You can also declare custom `operator fun of` functions to use bracket syntax with your own types. For example, if you have the following `DoubleMatrix` class:

```
class DoubleMatrix(vararg val rows: Row) {
    companion object {
        operator fun of(vararg rows: Row) = DoubleMatrix(*rows)
    }
    class Row(vararg val elements: Double) {
        companion object {
            operator fun of(vararg elements: Double) = Row(*elements)
        }
    }
}
```

You can create an `identityMatrix` class instance like this:

```
fun main() {
    val identityMatrix: DoubleMatrix = [
        [1.0, 0.0, 0.0],
        [0.0, 1.0, 0.0],
        [0.0, 0.0, 1.0],
    ]
}
```

In this example, the compiler translates the nested collection literals into calls to the corresponding `operator fun of` functions. The compiler resolves these calls recursively and uses the expected types to choose the correct overloads.

This feature is Experimental. To opt in, add the following compiler option to your build file:

Gradle

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xcollection-literals")
    }
}
```

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-Xcollection-literals</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

For more information, see the feature's KEEP.

Experimental

Improved compile-time constants

Kotlin 2.4.0 brings experimental improvements to compile-time constants, making support for numeric and string types more consistent and easier to use. These improvements include support for:

- Unsigned type operations.
- Standard library functions for strings, like `.lowercase()`, `.uppercase()`, and `.trim()` functions.
- Evaluation of the `.name` property of enum constants and the `KCallable` interface.

To make it clear which functions are evaluated at compile time, Kotlin 2.4.0 introduces the `IntrinsicConstEvaluation` annotation. Some functions are evaluated at compile-time but don't have the annotation yet. Later releases will add the annotation to the remaining functions. For a list of supported functions, see the KEEP appendix.

This feature is Experimental. To opt in, add the following compiler option to your build file:

Gradle

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xintrinsic-const-evaluation")
    }
}
```

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-Xintrinsic-const-evaluation</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

For more information, see the feature's KEEP.

Experimental

Improved unused result checks for higher-order functions

Kotlin 2.4.0 introduces a new Experimental `returnsResultOf()` contract to improve the unused return value checker.

This contract enables the checker to distinguish between unused results that can be ignored and meaningful unused results from higher-order functions that return the result of a lambda, such as the `let` scope function.

Kotlin contracts are Experimental. To opt in, add the `@OptIn(ExperimentalContracts::class)` annotation when declaring a function with a contract.

To use this feature, add `returnsResultOf()` to the function's contract:

```
import kotlin.contracts.ExperimentalContracts
import kotlin.contracts.contract

@OptIn(ExperimentalContracts::class)
inline fun <T, R> T.customLet(block: (T) -> R): R {
    contract {
        returnsResultOf(block)
    }
    return block(this)
}
```

Here's an example that uses a custom `.customLet()` function with a nullable value:

```
fun handleNullablePackageName(packageName: String?, builder: StringBuilder) {
    // The checker doesn't report a warning
    // because the return value of the append() function can be ignored
    packageName?.customLet { builder.append(it) }

    // The checker reports a warning because the returned string is unused
    packageName?.customLet { "kotlin.$it" }
}
```

The unused return value checker is Experimental and must be enabled to report unused return values. For more information about enabling and configuring the checker, see Unused return value checker.

How to enable

The `returnsResultOf()` contract is Experimental. Be aware that using it produces pre-release binaries that earlier Kotlin compiler versions can't read. To opt in, add the following compiler option to your build file:

Gradle

```
// build.gradle(.kts)
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xallow-returns-result-of")
    }
}
```

Maven

```
<!-- pom.xml -->
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-Xallow-returns-result-of</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Experimental

New `@IntroducedAt` annotation to generate version-based overloads for optional parameters

Kotlin 2.4.0 introduces the `@IntroducedAt` annotation for preserving binary compatibility when adding new optional parameters to published APIs.

Previously, adding optional parameters to a function often required using `@JvmOverloads`, which can generate more overloads than needed. Alternatively, preserving binary compatibility required you to keep older signatures as hidden deprecated overloads.

With the `@IntroducedAt` annotation, you can annotate newly added optional parameters with the version in which they were introduced. The compiler uses this information to automatically generate the corresponding hidden overloads.

This annotation is Experimental. To opt in, use the `@OptIn(ExperimentalVersionOverloading::class)` annotation.

Here's an example:

```
@OptIn(ExperimentalVersionOverloading::class)
fun Button(
    label: String = "",
    color: Color = DefaultColor,
    @IntroducedAt("1.1") borderColor: Color = DefaultBorderColor,
    @IntroducedAt("1.2") borderStyle: Style = DefaultBorderStyle,
    @IntroducedAt("1.2") borderWidth: Int = 1,
    onClick: () -> Unit
) {
    // Function body
}
```

In this example, the compiler generates hidden overloads for the older versions of the `Button()` function.

Since both `@IntroducedAt` and `@JvmOverloads` generate overloads, using them together can cause conflicting overloads. If you use both annotations, the compiler reports a warning. If you suppress the warning, the compiler prioritizes overloads generated from the `@IntroducedAt` annotation.

Standard library

Kotlin 2.4.0 stabilizes support for UUIDs in the common Kotlin standard library. It also adds new extension functions for converting unsigned integers to `BigInteger` on the JVM and support for checking sorted order.

Stable UUID API in the common Kotlin standard library

Kotlin 2.0.20 introduced a class for generating UUIDs (universally unique identifiers) and added support for converting between Kotlin and Java UUIDs. Later releases gradually improved this experimental feature by adding support for:

- Comparing UUIDs with `<` and `>` operators
- Parsing UUIDs from hex-and-dash and plain text formats
- Returning `null` when parsing invalid UUIDs.

In Kotlin 2.4.0, the `kotlin.uuid.Uuid` API becomes Stable. The only exceptions are the functions for generating V4 and V7 UUIDs, which remain Experimental and still require opt-in.

For more information about how to work with UUIDs, see UUIDs.

Support for checking sorted order

Kotlin 2.4.0 adds new extension functions for checking sorted order in iterables, arrays, and sequences.

This includes the following extension functions:

- `.isSorted()`
- `.isSortedDescending()`
- `.isSortedWith(comparator)`
- `.isSortedBy(selector)`
- `.isSortedByDescending(selector)`

You can use these extension functions to check whether elements are already sorted, without sorting them again or creating your own helper functions. They return `true` if the elements are in the specified order, or if there are

fewer than two elements, and `false` otherwise. These functions stop as soon as they encounter an out-of-order pair, which makes them efficient for large inputs.

Here's an example of checking sorted order with `.isSorted()` and `.isSortedBy()` functions:

```
data class User(val name: String, val age: Int)

fun main() {
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.isSorted())
    // true

    val users = listOf(
        User("Alice", 24),
        User("Bob", 31),
        User("Charlie", 29),
    )
    println(users.isSortedBy(User::age))
    // false
}
```

New API for converting unsigned integers to `BigInteger` on the JVM

Kotlin 2.4.0 introduces the `UInt.toBigInteger()` and `ULong.toBigInteger()` extension functions on the JVM.

Previously, converting `UInt` and `ULong` values to `BigInteger` required string-based workarounds or custom conversion logic. Starting with Kotlin 2.4.0, you can now use `.toBigInteger()` to convert unsigned integer values directly to `BigInteger`.

Here's an example:

```
fun main() {
        val unsignedLong = Long.MAX_VALUE.toULong() + 1uL
    val unsignedInt = UInt.MAX_VALUE

    println(unsignedLong.toBigInteger())
    // 9223372036854775808

    println(unsignedInt.toBigInteger())
    // 4294967295
   //sampleEnd
}
```

Experimental

New map fallback functions to distinguish `null` values and missing keys

Kotlin 2.4.0 adds new variants of the existing `.getOrElse()` and `.getOrPut()` map extension functions for maps with nullable values. These functions retrieve a value for a key or use a default value as a fallback. For maps with nullable values, the new variants let you choose whether a stored `null` value behaves like a missing key or an existing value, and they make that choice clear in their function names.

The new extension functions include the following:

- `.getOrElseIfNull(key, defaultValue)` and `.getOrPutIfNull(key, defaultValue)`, which return the default value if the key is missing or has a `null` value, similar to the existing `.getOrElse()` and `.getOrPut()` functions.

- `.getOrElseIfMissing(key, defaultValue)` and `.getOrPutIfMissing(key, defaultValue)`, which return the default value only when the map doesn't contain the specified key.

These APIs are Experimental and require opt-in with the `@OptIn(ExperimentalStdlibApi::class)` annotation.

Here's an example that demonstrates the difference between `.getOrPutIfNull()` and `.getOrPutIfMissing()` when the key exists with a `null` value:

```
@OptIn(ExperimentalStdlibApi::class)
fun main() {
    val mapForNull = mutableMapOf<String, String?>("user" to null)
    val mapForMissing = mutableMapOf<String, String?>("user" to null)

    // Replaces the value if "user" has a null value
    mapForNull.getOrPutIfNull("user") { "default_user" }

    println(mapForNull)
    // {user=default_user}

    // Keeps the null value because "user" exists in the map
    mapForMissing.getOrPutIfMissing("user") { "default_user" }

    println(mapForMissing)
    // {user=null}
}
```

You can also use the `.getOrElseIfMissing()` and `.getOrPutIfMissing()` functions for caches that store nullable values. If `defaultValue` returns `null`, the map stores it and doesn't call `defaultValue` again for the same key.

Here's an example:

```
data class Response(val body: String)

class Service {
    var queryCount = 0

    fun query(key: String): Response? {
        queryCount += 1
        return null
    }
}

@OptIn(ExperimentalStdlibApi::class)
fun main() {
    val service = Service()
    val cache = mutableMapOf<String, Response?>()

    fun getCachedResponseOrQuery(key: String): Response? =
        cache.getOrPutIfMissing(key) { service.query(key) }

    // Stores null because the cache doesn't contain "user"
    getCachedResponseOrQuery("user")

    println(cache)
    // {user=null}

    // Uses the cached null and doesn't query the service again
    getCachedResponseOrQuery("user")

    println(service.queryCount)
    // 1
}
```

We would appreciate your feedback in YouTrack.

Kotlin/JVM

Kotlin 2.4.0 supports a new Java version and enables annotations in metadata by default.

Support for Java 26

Starting with Kotlin 2.4.0, the compiler can generate classes containing Java 26 bytecode.

Annotations in metadata enabled by default

The Kotlin Metadata JVM library in Kotlin 2.2.0 introduced support for reading annotations stored in Kotlin metadata. With this support, the Kotlin compiler writes annotations into metadata alongside the JVM bytecode, making them accessible to the Kotlin Metadata JVM library. As a result, annotation processors and other tools can understand and manipulate these annotations at the metadata level without using reflection or modifying source code.

In Kotlin 2.4.0, this support is enabled by default.

Kotlin/Native

Starting with Kotlin 2.4.0, Swift export is promoted to Alpha. This release also brings support for Swift package import, Xcode 26.4, improvements for memory consumption, and garbage collection.

Default concurrent marking in garbage collector

In Kotlin 2.0.20, the Kotlin team introduced experimental support for the concurrent mark and sweep garbage collector (CMS GC). After processing user feedback and fixing regressions, we are now ready to enable CMS by default, starting with Kotlin 2.4.0.

The previous default parallel mark concurrent sweep (PMCS) setup in the garbage collector had to pause application threads while the GC marked objects in the heap. In contrast, CMS allows the marking phase to run concurrently with application threads.

This significantly improves GC pause duration and app responsiveness, which is important for the performance of latency-critical applications. CMS has already demonstrated its effectiveness in benchmarks for UI applications built with Compose Multiplatform.

If you face problems, you can switch back to PMCS. To do that, set the following binary option in your `gradle.properties` file:

```
kotlin.native.binary.gc=pmcs
```

For more information on the Kotlin/Native garbage collector, see our documentation.

Reduced memory consumption during devirtualization analysis

Previously, devirtualization analysis was one of the most memory-consuming phases in the Kotlin/Native compiler. Namely, the link release task consumed too much memory, especially in large projects.

Kotlin 2.4.0 introduces improvements that help reduce peak memory consumption during link release tasks.

According to benchmarks from one of our EAP users, the improved devirtualization analysis reduced memory consumption by link release tasks by half, saving at least 13 GB.

Support for Xcode 26.4

Starting with Kotlin 2.4.0, the Kotlin/Native compiler supports Xcode 26.4 – one of the latest stable versions of Xcode.

You can now update your Xcode and get access to the latest APIs to continue working on your Kotlin projects for Apple operating systems.

LLVM update to version 21

In Kotlin 2.4.0, we updated LLVM from version 19 to 21. The new version includes performance improvements and helps keep the Kotlin/Native compiler up to date.

This update shouldn't affect your code, but if you encounter any issues, please report them to our issue tracker.

Changes to Apple target support

Kotlin 2.4.0 raises the default minimum supported versions of Apple targets:

- For iOS and tvOS, from 14.0 to 15.0.
- For macOS, from 11.0 to 12.0.
- For watchOS, from 7.0 to 8.0.

If you need to support a lower version in your project than the default one, use the `freeCompilerArgs` option in your build file:

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget>().configureEach {
        binaries.configureEach {
            freeCompilerArgs += "-Xoverride-konan-properties=minVersion.ios=14.0"
            freeCompilerArgs += "-Xoverride-konan-properties=minVersion.macos=11.0"
            freeCompilerArgs += "-Xoverride-konan-properties=minVersion.tvos=14.0"
            freeCompilerArgs += "-Xoverride-konan-properties=minVersion.watchos=7.0"
        }
    }
}
```

Swift export goes Alpha with improved concurrency support

Starting with Kotlin 2.4.0, Kotlin's interoperability with Swift through Swift export is officially in Alpha! This release brings major improvements to concurrency support, adding native and direct structured concurrency to Swift export and the ability to export `kotlinx.coroutines` flows to Swift.

Support for structured concurrency

You can now seamlessly call suspending Kotlin code from Swift. Kotlin `suspend` functions and suspend functional types are exported as Swift's idiomatic `async` counterparts:

```
// Kotlin
suspend fun hello(): String {
    delay(1000)
    return "Hello Swift! This is Kotlin."
}
```

```
// Swift
let msg = try await hello()
```

Export of flow types to Swift

This update also adds support for exporting `kotlinx.coroutines` flows to Swift. Flows in `kotlinx.coroutines` represent an asynchronous stream of data that can be emitted and consumed concurrently. They are commonly used for reactive programming patterns, such as listening for database updates, network requests, or UI events.

Previously, the only way to expose the `Flow` interface from `kotlinx.coroutines.flow` to Swift was through third-party solutions. Now you can export flows out of the box into Swift's idiomatic counterpart: `AsyncSequence`.

The feature is enabled by default. You can export any public API with the `Flow` type to Swift while preserving type information. For example:

```
// Kotlin
// Type String is preserved when exporting Flow
fun flowOfStrings(): Flow<String> = flowOf("hello", "any", "world")
```

```
// Swift
var actual: [String] = []

// Type String is correctly inferred from Kotlin
for try await element in flowOfStrings().asAsyncSequence() {
    actual.append(element)
}
```

For more information about Swift export, see our documentation.

Experimental

Swift package import

Kotlin Multiplatform projects now can declare Swift packages as dependencies for an iOS app in their Gradle configuration:

```
// build.gradle.kts
kotlin {
    swiftPMDependencies {
        swiftPackage(
            url = url("https://github.com/firebase/firebase-ios-sdk.git"),
            version = from("12.11.0"),
            products = listOf(
                product("FirebaseAI"),
                product("FirebaseAnalytics"),
                ...
}
```

For working samples and more detailed information, see SwiftPM import.

If your project relies on CocoaPods dependencies, you can migrate the current setup to use Swift packages. The KMP tooling accounts for this use case and helps you reconfigure the project automatically. For details, see our CocoaPods migration guide.

Kotlin/Wasm

Kotlin 2.4.0 enables incremental compilation for Kotlin/Wasm by default and introduces support for the WebAssembly Component Model.

Incremental compilation enabled by default

Kotlin/Wasm introduced incremental compilation in Kotlin 2.1.0. Starting with Kotlin 2.4.0, it is Stable and enabled by default. With this feature, the compiler rebuilds only the files affected by recent changes, which significantly reduces build time.

To disable incremental compilation, add the following line to your project's `local.properties` or `gradle.properties` file:

```
# gradle.properties
kotlin.incremental.wasm=false
```

If you run into any issues, report them in YouTrack

Improved display of internal variables in Chrome DevTools

Kotlin 2.4.0 improves the debugging experience for Kotlin/Wasm in Chrome DevTools by making temporary, synthetic, and internal variables easier to distinguish from user-defined variables.

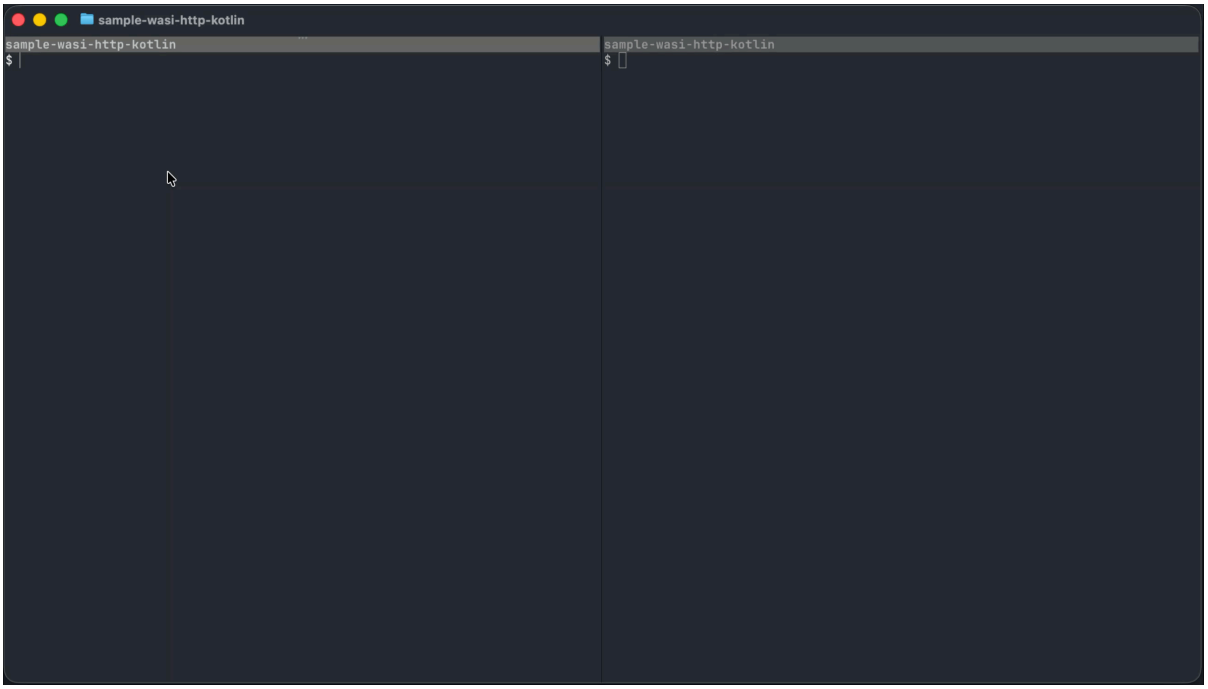
The Kotlin compiler and compiler plugins, such as Compose, can generate these variables. They now use the `~` prefix by default, so they are grouped together and moved to the end of the variable list, which Chrome DevTools sorts by name.

Experimental

Support for the WebAssembly Component Model

Kotlin/Wasm goes a step further in Kotlin 2.4.0 by introducing experimental support for the WebAssembly Component Model. The proposal defines a way to build components from Wasm modules through standardized interfaces and types. This approach helps Wasm evolve from a low-level binary instruction format into a system for composing reusable, language-agnostic components. It enables Kotlin/Wasm to go beyond the browser. For example, Kotlin and WebAssembly are well suited for Function-as-a-Service, also known as FaaS or serverless, applications.

To try this feature, check out a simple server built with `wasi:http` .



Share your feedback in YouTrack.

Kotlin/JS

Kotlin 2.4.0 further improves export to JavaScript/TypeScript, including support for exporting value classes, interfaces, and type variance, as well as ES2015 features when inlining JS code.

Support for value class export to JavaScript/TypeScript

Previously, only regular Kotlin classes could be exported to JavaScript/TypeScript. Kotlin 2.4.0 lifts that limitation. You can now export Kotlin's inline value classes as regular TypeScript classes.

To export a value class, mark it with the `@JsExport` annotation on the Kotlin side:

```
// Kotlin
@JsExport
@JvmInline
value class Email(val address: String) {
    init { require(address.contains("@")) { "Invalid email" } }
}

@JsExport
class AuthService {
    suspend fun login(email: Email): String = ...
}
```

From the TypeScript side, it looks like a regular class:

```
// TypeScript
import { AuthService, Email } from "..."
const auth = new AuthService();

console.log(await auth.login(new Email("jane@example.com")));
// "Welcome, jane@example.com!"
console.log(await auth.login(new Email("not-an-email")));
// "Invalid email"
```

For more information, see `@JsExport` annotation.

Support for ES2015 features when inlining JS code

Starting with Kotlin 2.4.0, JavaScript code inlining has full support for ES2015 features.

It's useful for interoperability with third-party libraries, as well as for direct control over automatic application code generation.

Now you can use modern JS features inside `js()` calls, including:

- `const` and `let` variable declarations
- ES classes
- Generators
- Lambdas (arrow functions)
- Spread and rest operators
- Template strings

Remember that the parameter of the `js()` function should be a string constant because it's parsed at compile time and translated to JavaScript code "as-is". For example, to inline the spread operator, use:

```
fun spreadExample(): dynamic = js("""
    const add = (a, b, c) => a + b + c;

    const nums = [1, 2, 3];
    const sum = add(...nums);

    const a = [1, 2, 3];
    const b = [...a, 4, 5, 6];

    return { sum, b: b };
""")
```

For more information on inlining JavaScript code, see our documentation.

Preserve type variance when exporting to TypeScript

Previously, Kotlin variance information in generic positions was lost when exporting types to TypeScript.

With Kotlin 2.4.0, variance annotation is now saved during export and mapped to TypeScript's variance annotations.

In your Kotlin code, define the variance of your generic type parameters:

```
// Kotlin
// 'out' signals covariance (the interface only produces T)
interface Producer<out T> {
    fun produce(): T
}

// 'in' signals contravariance (the interface only consumes T)
interface Consumer<in T> {
    fun consume(item: T)
}
```

With Kotlin 2.4.0, the `in` and `out` keywords are preserved in the generated TypeScript output:

```
// Generated .d.ts
export interface Producer<out T> {
    produce(): T;
}

export interface Consumer<in T> {
    consume(item: T): void;
}
```

Improved interface export to JavaScript/TypeScript

Kotlin 2.4.0 makes it more convenient to export Kotlin interfaces to JavaScript/TypeScript.

The new `@JsNoRuntime` annotation removes the previously required metadata for implementing Kotlin interfaces, allowing the direct mapping to regular TypeScript interfaces, similar to how external interfaces already behave by default.

To export a Kotlin interface, for example in your Kotlin Multiplatform project, annotate it with `@JsNoRuntime` in the common code:

```
// commonMain
import kotlin.js.JsNoRuntime

@JsNoRuntime
expect interface DataProcessor {
    fun process(data: String): Int
}
```

Then provide the actual implementation in your JS-specific source code:

```
// jsMain
@JsNoRuntime
actual interface DataProcessor {
    actual fun process(data: String)
}
```

Because the required metadata for implementing Kotlin interfaces is removed, the interface is mapped to a regular TypeScript interface:

```
// Generated .d.ts
export interface DataProcessor {
    process(data: string): void;
}
```

The `@JsNoRuntime` annotation is only allowed on standard interfaces, so that TypeScript can treat Kotlin interfaces as regular TypeScript interfaces. Therefore, the following operations are prohibited:

- `is` and `as` type checks.
- Class references with the `::class` syntax.
- Passing an interface as a reified type argument.

Avoid annotating external interfaces with `@JsNoRuntime`, as this results in a compiler warning.

Experimental

Lifting restrictions on exporting interfaces

Kotlin 2.4.0 makes another step toward the stabilization of `@JsExport`, improving how Kotlin interfaces are exported.

Now you can export Kotlin interfaces with nested classes and named companion objects:

```
@JsExport
interface Identity {
    class Metadata(val tag: String)

    companion object Registry {
        val defaultTag = "GUEST"
    }
}
```

For more information, see `@JsExport` annotation.

Gradle

Kotlin 2.4.0 is fully compatible with Gradle 7.6.3 through 9.5.0. You can also use Gradle versions up to the latest Gradle release. However, be aware that doing so may result in deprecation warnings, and some new Gradle features might not work. Kotlin 2.4.0 also brings improvements like consistent default module names across platforms and compiler messages written to the Problems API for the Kotlin/JVM.

Minimum supported AGP version bumped to 8.5.2

Starting with Kotlin 2.4.0, the minimum supported Android Gradle plugin version is 8.5.2.

Consistent module names across platforms

Prior to Kotlin 2.4.0, default module names differed across platforms. This inconsistency could cause naming conflicts and resolution issues. Kotlin 2.4.0 standardizes the default names to `{group}:{project_name}` across all platforms.

If you need to revert the JVM module name to its previous version, add the following to your `build.gradle.kts` file for a Kotlin/JVM project:

```
kotlin {
    compilerOptions.moduleName(project.name)
}
```

For a multiplatform project:

```
kotlin {
    jvm {
        compilerOptions.moduleName(project.name)
    }
}
```

Compiler messages written to Problems API for Kotlin/JVM

In Kotlin 2.2.0, the Kotlin Gradle plugin (KGP) started reporting diagnostics to Gradle's Problems API to provide a consistent experience both in Gradle's CLI and in IntelliJ IDEA.

In Kotlin 2.4.0, the plugin also writes compiler messages to the Problems API for Kotlin/JVM, bringing the API closer to becoming a single source for all logs and messages.

Maven

Kotlin 2.4.0 makes project configuration even easier with support for Maven Toolchains and automatic alignment between Java and JVM target versions.

Automatic alignment between Java and JVM target versions

To simplify project configuration and prevent compatibility issues, the Kotlin Maven plugin now automatically aligns the JVM target version with the Java compiler version configured in the project.

This ensures that the Kotlin and Maven compilers target the same bytecode version, avoiding issues where Kotlin-generated bytecode is incompatible with the rest of the project or the intended deployment environment.

With the `<extensions>` option enabled, you don't need to set the `kotlin.compiler.jvmTarget` or `kotlin.compiler.jdkRelease` options. If neither of them is defined, the Kotlin Maven plugin automatically resolves the JVM target version in the following order:

1. As the `maven.compiler.release` version defined either as a project property or within the `maven-compiler-plugin` configuration.

   In this case, both `jvmTarget` and `jdkRelease` compiler options are set for the Kotlin compiler, limiting the API to a specific JDK version.

2. As the `maven.compiler.target` version in case the Maven release version is not set. The compiler target can be defined either as a project property or within the `maven-compiler-plugin` configuration.

   In this case, only Kotlin's `jvmTarget` is set, and the API is not limited to a specific JDK version.

This greatly simplifies your Kotlin project configuration, so your `pom.xml` file can look like this:

```
<properties>
    <maven.compiler.release>17</maven.compiler.release>
    <kotlin.version>2.4.20</kotlin.version>
</properties>

<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <extensions>true</extensions>
        </plugin>
    </plugins>
</build>
```

During the build, the plugin outputs a similar message:

```
[INFO] Using jvmTarget=17 (derived from maven.compiler.release=17)
```

The `<extensions>` option only checks project-level properties and the global `maven-compiler-plugin` configuration. It doesn't check the configurations defined in the plugin's `<executions>` section.

For more information about automatic project configuration, see our documentation.

Support for Maven Toolchains

Kotlin 2.4.0 introduces support for Maven Toolchains to the Kotlin Maven plugin.

The feature helps manage the JDK version in your build. With Maven Toolchains, you can specify the JDK version used for Kotlin compilation, independent of the JVM version running Maven (set in `JAVA_HOME`). When the `maven-toolchains-plugin` is configured in the build, the Kotlin Maven plugin automatically picks up the selected JDK toolchain, in the same way the Maven compiler plugin and other Maven plugins do. This allows you to configure a single toolchain to control the JDK used across all plugins in the build, including Kotlin compilation:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-toolchains-plugin</artifactId>
    <version>3.2.0</version>
    <executions>
        <execution>
            <goals>
                <goal>toolchain</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <toolchains>
            <jdk>
                <version>21</version>
            </jdk>
        </toolchains>
    </configuration>
</plugin>
```

Keep in mind the priority of different ways to set up the JDK version:

1. `jdkHome` in the `kotlin-maven-plugin` configuration. An explicitly set `jdkHome` option always takes precedence over the toolchain version.
2. JDK version in `maven-toolchains-plugin`. The JDK version set through Maven Toolchains overrides the JDK version set in the `JAVA_HOME` path.
3. The `JAVA_HOME` path.

You can also use a plugin-specific `<jdkToolchain>` option to directly set the JDK version in the toolchain of `kotlin-maven-plugin`. Compared to using `maven-toolchains-plugin`, this parameter only affects Kotlin compilation and has no impact on other plugins in the build.

Currently, setting `maven-toolchains-plugin` to use a specific JDK version does not affect the `kapt` and `test-kapt` goals of `kotlin-maven-plugin`. To work around this, set the necessary version in the `JAVA_HOME` path. For more details, see KT-79897.

For more information on configuring Kotlin Maven projects, see our documentation.

Build tools API

Kotlin 2.4.0 brings a number of improvements to the build tools API (BTA). The BTA:

- Introduces new type-safe abstractions for most JVM and common compiler options. The BTA now handles their format instead of the client, reducing the risk of errors and providing an additional layer of assistance. This change is backwards-compatible at runtime, but it may break source compatibility.
- Can now track non-source changes in incremental compilation, such as configuring a different Kotlin version or changing compiler options. Build systems can control this behavior through the `BaseIncrementalCompilationConfiguration.TRACK_CONFIGURATION_INPUTS` option.
- Supports binary compatibility validation through the `AbiValidationToolchain`, making it easier for other build systems to add this functionality.
- Introduces a new feature so that build systems can customize how compiler messages are displayed through the `CompilerMessageRenderer` interface and the `JvmCompilationOperation` builder.

- Introduces new options for configuring Kotlin daemon logging:
  - `LOGS_PATH` — the directory for daemon log files.
  - `LOGS_FILE_SIZE_LIMIT` — the maximum log file size in bytes.
  - `LOGS_FILE_COUNT_LIMIT` — the maximum number of retained log files.

  By default, limits are set to a value specific to the Kotlin compiler version. To have no limit, build tools must set the option to `null`.

  Build systems can set the option when configuring the execution policy:

```
val executionPolicy = kotlinToolchains.daemonExecutionPolicy {
    set(ExecutionPolicy.WithDaemon.LOGS_PATH, Paths("/var/log/kotlin-daemon"))
    set(ExecutionPolicy.WithDaemon.LOGS_FILE_SIZE_LIMIT, 10_485_760L)
    set(ExecutionPolicy.WithDaemon.LOGS_FILE_COUNT_LIMIT, 10)
}
```

Kotlin compiler

Kotlin 2.4.0 includes more consistent behavior for inline functions declared in the same module during `.klib` compilation.

Consistent intra-module function inlining during klib compilation

Previously, function inlining behaved inconsistently on different Kotlin platforms. The JetBrains team is working to unify it across all supported platforms to ensure the same compatibility guarantees.

On the Kotlin/JVM, function inlining happens at compile time. So, when Kotlin sources are compiled with the Kotlin/JVM compiler, the resulting class files have no inline function calls in the bytecode because the bodies of inline functions are inlined into their call sites, so their behavior is fixed during compilation.

On the contrary, on Kotlin/Native, Kotlin/JS, and Kotlin/Wasm, function inlining did not happen during source-to-klib compilation, only during binary generation. As a result, the behavior of inline functions wasn't fixed during `.klib` compilation, and `.klib` libraries didn't provide the same compatibility guarantees for inline functions as Kotlin/JVM does.

Kotlin 2.4.0 takes the first step in unifying the behavior of inline functions by enabling intra-module inlining when generating `.klib` artifacts:

```
// Existing logging.klib library
inline fun logDebug(message: String) {
    println("[DEBUG] $message")
}
```

```
// Currently compiled App module
inline fun greetUser(name: String) {
    println("Hello, $name!")
}

fun main() {
    logDebug("App started") // Not inlined: declared in another module
    greetUser("Alice")      // Inlined: declared in the same module
}
```

When compiled to a `.klib`, the code looks something like:

```
// Pseudocode
fun main() {
    logDebug("App started")  // Not inlined, declared in another module
    val tmp0 = "Alice"
    println("Hello, $tmp0!") // Inlined from greetUser()
}
```

This means only inline functions declared in the same module are inlined during `.klib` compilation. Other functions, in this case, are inlined during the generation of platform-specific binaries.

How to enable

Starting with 2.4.0, the intra-module inlining is enabled by default for Kotlin/Native, Kotlin/JS, and Kotlin/Wasm.

If you face unexpected problems with this feature, you can disable it using the following compiler option in the command line:

```
-Xklib-ir-inliner=disabled
```

The next step is to enable cross-module inlining to ensure all inline functions in the project are consistently inlined. This change is planned for future Kotlin releases, but you can already try it out using the following compiler option in the command line:

```
-Xklib-ir-inliner=full
```

Please share your feedback and report any problems in YouTrack.

Consistent partial library linkage across Kotlin compilers

In Kotlin 1.9.0, partial library linkage was enabled by default for both the Kotlin/Native and Kotlin/JS compilers, with Kotlin/Wasm following in Kotlin 2.0.0. This feature effectively makes compilers treat linkage issues in Kotlin libraries consistently with Kotlin/JVM.

Since then, we haven't received negative feedback and haven't noticed users disabling the partial linkage in their projects. That's why starting with Kotlin 2.4.0, the partial linkage is always enabled, and the `-Xpartial-linkage` compiler option is now deprecated.

The default log level for all Kotlin compilers is `SILENT`. Linkage issues are not reported during compilation. To change this behavior in your projects, set the `-Xpartial-linkage-loglevel` compiler option in your build file:

```
// build.gradle.kts
kotlin {
    macosX64("native") {
        binaries.executable()

        compilations.configureEach {
            compilerOptions.configure {
                // To report linkage issues with the “info” log level:
                freeCompilerArgs.add("-Xpartial-linkage-loglevel=INFO")

                // To report issues as errors:
                freeCompilerArgs.add("-Xpartial-linkage-loglevel=ERROR")
            }
        }
    }
}
```

- `INFO` reports linkage issues with the "info" log level.
- `WARNING` reports warnings at compile time and records them in compilation logs.
- `ERROR` allows compilation to fail in case of linkage issues and reports errors in compilation logs. Use this option to examine the linkage issues more closely.

If you encounter issues with this feature, please report them in our issue tracker.

Kotlin compiler plugins

In Kotlin 2.4.0, Kotlin's compiler plugins received notable updates, too. The kapt plugin can now exclude unnecessary annotation processors from the compile classpath, and the Power-assert plugin offers simplified configuration through the new runtime library.

kapt: Exclude annotation processors from compile classpath

Kotlin 2.4.0 adds support for the `includeCompileClasspath` configuration option for annotation processor discovery, similar to the Kotlin Gradle plugin. The new option allows you to exclude unnecessary annotation processors from the compile classpath.

To configure this in your build file, set the `includeCompileClasspath` option to `false` in the `<execution>` section of the kapt plugin:

```
<execution>
    <id>kapt</id>
        <goals><goal>kapt</goal></goals>
        <configuration>
            <!-- Add new option -->
            <includeCompileClasspath>false</includeCompileClasspath>
            <sourceDirs>...</sourceDirs>
            <annotationProcessorPaths>...</annotationProcessorPaths>
        </configuration>
</execution>
```

Alternatively, you can do the same with the `kapt.include.compile.classpath` in the `<properties>` section:

```
<properties>
    <kapt.include.compile.classpath>false</kapt.include.compile.classpath>
</properties>
```

With the option set to `false`, annotation processors not included in the `<annotationProcessorPaths>` section of the kapt configuration are excluded from the kapt processing.

If `includeCompileClasspath` is not set and kapt detects an annotation processor on the compile classpath that is not explicitly defined in the `<annotationProcessorPaths>` section, you'll see the following deprecation warning:

```
[WARNING] Annotation processors discovery from compile classpath is deprecated. Set
'kapt.include.compile.classpath=false' to disable discovery.
```

For more information on kapt configuration, see our documentation.

Power-assert: New runtime library

Kotlin 2.4.0 makes Power-assert capable functions more discoverable and easier to configure with the new runtime library.

Previously, adopting Power-assert required complex build configurations and function parameter conventions. Starting with this release, Power-assert capable functions can use the new runtime library to integrate directly with the compiler plugin transformations.

This brings major improvements for both plugin users and library authors:

- The new `CallExplanation` data structure provides detailed information about the call site. This enables more dynamic diagram rendering for assertion failures and better integration with external tools.
- The new `@PowerAssert` annotation makes assertion functions instantly discoverable by the compiler plugin. That way, you can now add out-of-the-box support for Power-assert into your libraries.

> Use our example collection as a playground for experimenting with the new features.

For more information, see our documentation.

Compose compiler

With Kotlin 2.4.0, the Compose compiler offers more consistent incremental compilation and advances the deprecation cycle of several feature flags.

Consistent incremental compilation for internal declarations

Starting from Kotlin 2.4.0, the Compose compiler offers more consistent incremental compilation. Stability of internal types across different files is now inferred during runtime. This allows Compose to update inferred stability values even when class usages are not recompiled.

As a side effect, the size of your artifacts may increase whenever a `@Composable` function uses an `internal` class from a different file as a parameter. This is caused by the compiler encoding the execution paths for both stable and unstable cases, since stability has to be decided during runtime. This overhead of runtime stability is removed by minifiers that perform full-app optimizations (such as R8) as they are able to infer the unnecessary execution path and eliminate it.

This update does not change the final stability value, so the behavior of `@Composable` functions remains unchanged.

Feature flag deprecations

Kotlin 2.4.0 advances the deprecation cycle of experimental feature flags that graduated to stable and are now enabled by default:

- `StrongSkipping`, `IntrinsicRemember`, and associated DSL properties are advanced to `DeprecationLevel.ERROR`. They will be removed in Kotlin 2.5.0.
- `OptimizeNonSkippingGroups` and `PausableComposition` are now deprecated. They are scheduled to be removed in Kotlin 2.6.0.

Breaking changes and deprecations

This section highlights important breaking changes and deprecations. For a complete overview, see our Compatibility guide.

- Starting with Kotlin 2.4.0, the compiler no longer supports `-language-version=1.9`. As a result, the K1 compiler is no longer supported.
- Kotlin 2.4.0 streamlines the DSL for binary compatibility validation in the Kotlin Gradle plugin and deprecates some parts. For the latest DSL, see Binary compatibility validation in the Kotlin Gradle plugin.
- Support for Kotlin script execution through the `KotlinScriptMojo` Maven plugin has been removed.

Documentation updates

We made the following documentation changes in the Kotlin ecosystem:

- Liquid Glass in a Compose Multiplatform app – Migrate an iOS app from fully Compose-driven navigation to native SwiftUI navigation with iOS 26 Liquid Glass styling.
- Adding Swift packages as dependencies to KMP modules – Learn how to set up a SwiftPM dependency in your KMP project.
- Switch Kotlin Multiplatform project from CocoaPods to SwiftPM dependencies manually or with Junie – Learn how you can use Junie and Kotlin AI skills to make migration easier.
- Configure TeamCity for a KMP app – Use TeamCity to build, test, and deploy your KMP applications.
- Recommended serialization approaches for Navigation 3 – Find the best way to use serialization with Navigation 3 in your CMP application.
- Multiplatform ViewModel – Learn how to set up and work with ViewModels in a multiplatform project.
- Backend development with Kotlin – Explore the different frameworks you can use for backend development.
- Create a task manager app with Spring Boot and Claude – Learn how Claude can help you create an app with Spring Boot from scratch.
- Configure a Maven project – Set up Kotlin compilation in your existing Java Maven project or in a new Kotlin Maven project.
- Test Kotlin projects with Maven – Learn how to create tests with JUnit and use Maven plugins to run unit and integration tests.
- Use annotation processors in Kotlin projects – Choose between kapt and KSP to process annotations in your backend project.
- Kotlin AI skills – Use agent skills to help you perform Kotlin-specific tasks.
- Kotlin Language Server – Read about JetBrains' official implementation of the Language Server Protocol (LSP) for Kotlin.
- Numbers – Explore Kotlin's number types and how to work with them.
- Getting started with KSP – Learn how to add a KSP-based processor to your project or create your own.
- Migrate from kapt to KSP – Migrate your annotation processors to get the best out of Kotlin's features.
- Lincheck overview – Understand how Lincheck works behind the scenes to test concurrent code on the JVM.
- Getting started with Lincheck – Create a project and run tests with Lincheck.

- Testing arbitrary code with Lincheck – Learn how to test concurrent code with Lincheck.
- How to test data structures with Lincheck – Dive into Lincheck's data structure testing process.
- Testing strategies with Lincheck – Learn about Lincheck's testing strategies: model checking and stress testing.
- Configuring a testing strategy with Lincheck – Explore the different options for Lincheck's testing strategies.
- Deploy a Ktor application with Dokku – Learn about the deployment workflow with Dokku.

Compatibility guide for Kotlin 2.4.x

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 2.3 to Kotlin 2.4. This document also includes information about tool-related changes.

Basic terms

In this document, we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Drop support for `-language-version=1.9` and the K1 compiler

Issue: KT-80590

Component: Compiler

Incompatible change type: source

Short summary: Starting with Kotlin 2.4, the compiler no longer supports `-language-version=1.9`. As a result, the K1 compiler is no longer supported.

Deprecation cycle:

- 2.2.0: report a warning when using `-language-version` with version 1.9
- 2.4.0: raise the warning to an error

Prohibit flexible explicit nullable type arguments for Java types

Issue: KTLC-284

Component: Core language

Incompatible change type: source

Short summary: Previously, when calling Java APIs from Kotlin, the compiler could treat explicitly specified nullable type arguments as flexible type arguments. Kotlin 2.4.0 no longer applies this behavior for nullable type arguments, so the compiler now reports errors for code that could break type safety or fail at runtime.

Deprecation cycle:

- 2.2.0: report a warning for explicitly specified nullable type arguments that are treated as flexible types
- 2.4.0: raise the warning to an error

Prohibit always-false `is` checks for definitely incompatible types

Issue: KTLC-365

Component: Core language

Incompatible change type: source

Short summary: The compiler now prevents meaningless `is` checks that are always false because the checked types are definitely incompatible. This keeps the behavior consistent with other operations involving incompatible types.

Deprecation cycle:

- 2.0.0: report a warning for `is` checks with definitely incompatible types
- 2.4.0: raise the warning to an error

Prohibit exposing types and declarations with lower visibility in inline functions

Issue: KTLC-283

Component: Core language

Incompatible change type: source

Short summary: The compiler now prevents inline functions from exposing types and declarations that have lower visibility than the inline function itself.

Deprecation cycle:

- 2.3.0: report a warning for exposing types and declarations with lower visibility in inline functions
- 2.4.0: raise the warning to an error

Change default use-site target selection for annotations

Issue: KTLC-391

Component: Core language

Incompatible change type: binary

Short summary: Kotlin 2.4.0 updates the defaulting rules for propagating annotations to parameters, properties, and fields. This can affect annotation processing, reflection, and binary metadata after recompilation. When you don't specify a use-site target, the compiler now uses `param` and `property` if they apply, and uses `field` only if `property` doesn't apply.

You can specify a use-site target explicitly, such as `@param:Annotation` instead of `@Annotation`. To use the previous defaulting rule for your whole project, add `-Xannotation-default-target=first-only` to your build file.

Deprecation cycle:

- 2.2.0: report a warning when the new defaulting rule changes the chosen use-site targets
- 2.4.0: enable the new defaulting rule

Forbid implicit references to inaccessible types

Issue: KTLC-384

Component: Core language

Incompatible change type: source

Short summary: Using declarations that implicitly reference inaccessible types from indirect dependencies now results in an error.

To migrate, add an explicit dependency on the module that declares the inaccessible type, or update the intermediate API so it doesn't expose that type.

Deprecation cycle:

- 2.3.0: report a warning for implicit references to inaccessible types
- 2.4.0: raise the warning to an error

Enforce Jakarta nullability annotations

Issue: KTLC-285

Component: Core language

Incompatible change type: source

Short summary: The compiler now enforces declared nullability in Kotlin for Java declarations that use `jakarta.annotation.Nullable` or `jakarta.annotation.Nonnull`. If you assign a Java declaration marked as nullable by these annotations to a non-null Kotlin type, the compiler reports an error.

Deprecation cycle:

- 2.2.0: report a warning for nullability mismatches in Java declarations annotated with Jakarta nullability annotations
- 2.4.0: raise the warning to an error

Report misplaced type arguments in callable reference qualifiers

Issue: KTLC-388

Component: Core language

Incompatible change type: source

Short summary: The compiler now checks the left-hand side of callable references and reports a warning if an inner class contains type arguments in the wrong part of the qualifier.

To migrate, update the reference so that each type argument belongs to the class that declares it. For example, write the full type `Outer<Int>.Inner<String>::toString` instead of `Inner<String, Int>::toString`.

Deprecation cycle:

- 2.4.0: report a warning when type arguments in the left-hand side of a callable reference belong to another part of the qualifier

Report errors for class literals from reified type parameters with nullable upper bounds

Issue: KTLC-370

Component: Core language

Incompatible change type: source

Short summary: The compiler now reports an error when you use `::class` on an expression whose type comes from a reified type parameter with a nullable upper bound. If you use `::class` on such an expression, make the value non-null first with an explicit null check or the `!!` operator.

Deprecation cycle:

- 2.3.0: report a warning when `::class` is used on an expression whose type comes from a reified type parameter with a nullable upper bound
- 2.4.0: raise the warning to an error

Prohibit initialization before declarations in anonymous objects

Issue: KTLC-290

Component: Core language

Incompatible change type: source

Short summary: Kotlin now reports an error when you initialize a property in an `init` block of an anonymous object before declaring that property.

Deprecation cycle:

- 2.2.20: report a warning when an `init` block in an anonymous object initializes a property before the property declaration
- 2.4.0: raise the warning to an error

Enforce exhaustiveness for `when` expressions with non-abstract Java sealed classes

Issue: KTLC-366

Component: Core language

Incompatible change type: source

Short summary: Kotlin now checks exhaustiveness more strictly and requires an `else` branch or a branch that matches the sealed class itself when you use a `when` expression with a non-abstract Java sealed class. Previously, Kotlin could treat such `when` expressions as exhaustive even though the Java sealed class itself could be instantiated directly.

Deprecation cycle:

- 2.3.0: report a warning for non-exhaustive `when` expressions with non-abstract Java sealed classes
- 2.4.0: raise the warning to an error

Prohibit `operator` modifier on `getValue()` and `setValue()` functions with too many parameters

Issue: KTLC-289

Component: Core language

Incompatible change type: source

Short summary: When you mark the `getValue()` or `setValue()` functions with the `operator` modifier, the compiler now checks that they have the required number of value parameters. The `getValue()` function must have exactly two value parameters, and the `setValue()` function must have exactly three. To migrate, remove the `operator` modifier or change the function signature.

Deprecation cycle:

- 2.2.20: report a warning for `operator` `getValue()` and `setValue()` functions with too many value parameters
- 2.4.0: raise the warning to an error

Prohibit inconsistent type arguments in generic calls

Issue: KTLC-373

Component: Core language

Incompatible change type: source

Short summary: When you specify type arguments in a generic call, the compiler now reports an error if one type argument violates an upper-bound constraint that depends on another type argument. If type parameters depend on each other, use type arguments that match those constraints, for example `Container<Alpha, AlphaKey>()` instead of `Container<Alpha, BetaKey>()`.

Deprecation cycle:

- 2.3.0: report a warning when explicit type arguments in a generic call violate upper-bound constraints between type parameters
- 2.4.0: raise the warning to an error

Deprecate references to the `javaClass` property

Issue: KTLC-375

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 2.4.0 deprecates property references to the `javaClass` property to reduce confusion with `::class.java`. Use `.javaClass` to get the runtime Java class of an object, or `::class.java` to get a Java class reference.

Deprecation cycle:

- 2.4.0: report a warning for property references to the `javaClass` property

Report errors for implicit enum constructor calls that require opt-in

Issue: KTLC-359

Component: Core language

Incompatible change type: source

Short summary: Kotlin now reports an error when an enum entry implicitly calls an enum primary constructor that requires opt-in. To migrate, add `@OptIn` to the enum class or to each enum entry that calls the constructor.

Deprecation cycle:

- 2.2.20: report a warning when an enum entry implicitly calls an enum primary constructor that requires opt-in
- 2.4.0: raise the warning to an error

Forbid `inline` modifier on enum entries

Issue: KTLC-361

Component: Core language

Incompatible change type: source

Short summary: Kotlin now reports an error when you use the `inline` modifier on an enum entry.

Deprecation cycle:

- 2.3.0: report a warning when the `inline` modifier is used on an enum entry
- 2.4.0: raise the warning to an error

Prohibit array literals outside annotation calls and parameter defaults

Issue: KTLC-369

Component: Core language

Incompatible change type: source

Short summary: Using array literals outside annotation calls and default values for annotation parameters now results in an error. To migrate, use `arrayOf(...)`, for example `Roles(arrayOf("admin", "user"))` instead of `Roles(["admin", "user"])`.

Deprecation cycle:

- 2.3.0: report a warning for array literals outside annotation calls and default values for annotation parameters
- 2.4.0: raise the warning to an error

Prohibit `_root_ide_package_` in CLI compiler mode

Issue: KTLC-378

Component: Compiler

Incompatible change type: source

Short summary: Using the IDE-only `_root_ide_package_` qualifier in CLI compiler mode now results in an error.

Deprecation cycle:

- 2.3.20: report a warning for `_root_ide_package_` references in CLI compiler mode
- 2.4.0: raise the warning to an error

Correct equality for function references with vararg conversions

Issue: KTLC-385

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin/JVM now treats function references with different conversions as unequal. Previously, Kotlin/JVM ignored vararg conversion in equality checks when the same function reference also used another conversion, so `getDefault(::foo) == getDefaultAndVararg(::foo)` could return `true` even though only one side used vararg conversion.

Deprecation cycle:

- 2.4.0: introduce the new behavior

Enforce opt-in for companion object access

Issue: KTLC-386

Component: Core language

Incompatible change type: source

Short summary: Kotlin now reports an opt-in error when a class name reference resolves to a companion object that requires opt-in. For example, `val p = C` requires opt-in if `C` resolves to a companion object marked with an opt-in annotation.

Deprecation cycle:

- 2.3.20: report a warning when companion object access requires opt-in
- 2.4.0: raise the warning to an error for `ERROR`-level opt-in requirements

Report type mismatches from supertypes with nested generic arguments

Issue: KTLC-372

Component: Core language

Incompatible change type: source

Short summary: Kotlin now reports an error when the compiler detects a type mismatch involving a supertype with nested generic arguments. Previously, the compiler could miss this mismatch, which later failed with a `ClassCastException`. To migrate, use a type argument that matches the receiver's generic type, or remove the explicit type argument so the compiler can infer it.

Deprecation cycle:

- 2.4.0: report an error for type mismatches involving supertypes with nested generic arguments

Prohibit inferred types with inaccessible declarations

Issue: KTLC-363

Component: Core language

Incompatible change type: source

Short summary: Using an inferred type that contains a declaration inaccessible in the current scope now results in an error.

Deprecation cycle:

- 2.3.0: report a warning when an inferred type contains a declaration that isn't accessible in the current scope
- 2.4.0: raise the warning to an error

Standard library

Deprecate `kotlin.io.readLine()` function

Issue: KTLC-394

Component: kotlin-stdlib

Incompatible change type: source

Short summary: The `kotlin.io.readLine()` function is deprecated. Use the `readln()` function instead of `readLine()!!`, and the `readlnOrNull()` function instead of `readLine()`.

Deprecation cycle:

- 2.4.0: report a warning when using `kotlin.io.readLine()`

Deprecate `AbstractCoroutineContextKey` and related APIs

Issue: KT-84970

Component: kotlin-stdlib

Incompatible change type: source

Short summary: The `AbstractCoroutineContextKey` class and its related APIs were experimental since Kotlin 1.3 and proved to be error-prone. For this reason, this class and the related `getPolymorphicElement()` and `minusPolymorphicKey()` functions are deprecated.

Deprecation cycle:

- 2.4.0: report a warning when using the deprecated APIs

Change `Random.nextDouble()` contract for infinite bounds

Issue: KT-84368

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: The documented contract for `Random.nextDouble(until)` now requires the `until` bound to be finite. Use a finite bound instead.

Deprecation cycle:

- 2.4.0: enable the new behavior

Tools

Deprecate legacy Kotlin/JS compiler type selection APIs

Issue: KT-64275, KT-84753

Component: Gradle

Incompatible change type: source

Short summary: Kotlin 2.4.0 removes deprecated Gradle APIs related to selecting the legacy Kotlin/JS compiler type.

Additionally, the `KotlinJsCompilerType` enum and the `KotlinProjectExtension.js()` overloads with a compiler type parameter are deprecated. To migrate, remove the compiler type argument from the `js()` target declaration and use the `js {}` block instead.

Deprecation cycle:

- 1.8.0: deprecate legacy Kotlin/JS compiler type constants
- 2.4.0: remove the deprecated legacy compiler type APIs and report a warning when using `KotlinJsCompilerType` or `KotlinProjectExtension.js()` overloads with a compiler type parameter

Deprecate `sourceSets` in the Kotlin Android extension

Issue: KT-74451

Component: Gradle

Incompatible change type: source

Short summary: The `sourceSets` property in `KotlinAndroidProjectExtension` is deprecated. To migrate, configure source sets through the Android Gradle plugin's `android { sourceSets { ... } }` block instead.

Deprecation cycle:

- 2.4.0: report a warning when accessing `sourceSets` from `KotlinAndroidProjectExtension`

Remove consumable configurations for Kotlin/Native Apple frameworks

Issue: KT-74503, KT-82230

Component: Gradle

Incompatible change type: source

Short summary: Kotlin 2.4.0 removes generated consumable Gradle configurations that expose Kotlin/Native Apple frameworks as outgoing artifacts.

Deprecation cycle:

- 2.4.0: remove consumable configurations for Kotlin/Native Apple frameworks

Remove deprecated task, compilation, and DSL APIs from the Kotlin Gradle plugin

Issue: KT-85509

Component: Gradle

Incompatible change type: source

Short summary: Kotlin 2.4.0 removes the following deprecated Kotlin Gradle plugin APIs:

Compile task configuration APIs:

- `KotlinJvmCompile.parentKotlinOptions`
- `KotlinJvmCompile.moduleName`
- `KotlinJvmFactory.createKotlinJvmOptions()`
- `BaseKotlinCompile.moduleName` from `KotlinCompile` and `Kotlin2JsCompile` tasks

Kotlin Multiplatform hierarchy and target APIs:

- `DeprecatedKotlinTargetHierarchyDsl`
- `KotlinMultiplatformExtension.targetHierarchy`
- `KotlinTargetComponent.sourcesArtifacts`
- `KotlinTarget.sourceSets`
- `KotlinHierarchyBuilder.withoutCompilations()`
- `KotlinHierarchyBuilder.filterCompilations()`
- `KotlinHierarchyBuilder.withWasm()`
- `KotlinCompilation.defaultSourceSetName`

Kotlin compilation task APIs:

- `KotlinCompilation.compileKotlinTaskProvider`
- `KotlinCompilation.compileKotlinTask`

Kotlin dependency handler APIs:

- `KotlinDependencyHandler.enforcedPlatform()`
- `KotlinDependencyHandler.platform()` Other deprecated task and extension APIs:
- `KaptExtension.processors`
- `KotlinTest.excludes`
- `KotlinTest.fileResolver`
- `KotlinTest.execHandleFactory`
- `IncrementalSyncTask.destinationDir`

To migrate, remove usages of these APIs and use the replacements suggested by the deprecation diagnostics.

Deprecation cycle:

- 2.4.0: remove the deprecated APIs

Deprecate explicit shrunk classpath snapshot configuration

Issue: KT-75837

Component: Build tools API

Incompatible change type: source

Short summary: The `shrunkClasspathSnapshot` configuration parameter in `ClasspathSnapshotBasedIncrementalCompilationApproachParameters` is deprecated. The shrunk classpath snapshot is an internal incremental compilation cache, so the compiler now creates and manages it automatically under the incremental compiler metadata `workingDirectory`. To migrate, use the automatically managed snapshot file, instead of passing a value to `shrunkClasspathSnapshot`.

Deprecation cycle:

- 2.4.0: report a warning when using `shrunkClasspathSnapshot`

Remove redundant ABI validation Gradle DSL elements

Issue: KT-80685

Component: Gradle

Incompatible change type: source

Short summary: Kotlin 2.4.0 simplifies the ABI validation Gradle DSL and removes redundant configuration entries. To migrate, configure report settings directly in `abiValidation {}` instead of `abiValidation { legacyDump { ... } }`, remove `abiValidation { klib { enabled = ... } }`, and use `keepLocallyUnsupportedTargets` instead of `klib.keepUnsupportedTargets`.

Deprecation cycle:

- 2.4.0: remove redundant ABI validation DSL elements

Deprecate obsolete Compose compiler Gradle plugin options

Issue: KT-85343

Component: Gradle

Incompatible change type: source

Short summary: In Kotlin 2.4.0, the following deprecated Compose compiler Gradle plugin options now report an error when used:

- `generateFunctionKeyMetaClasses`
- `enableIntrinsicRemember`
- `enableNonSkippingGroupOptimization`
- `enableStrongSkippingMode`
- `stabilityConfigurationFile`
- `ComposeFeatureFlag.StrongSkipping`
- `ComposeFeatureFlag.IntrinsicRemember`

Use `featureFlags` instead of the deprecated feature options, and `stabilityConfigurationFiles` instead of `stabilityConfigurationFile`.

Deprecation cycle:

- 2.0.20: report warnings for `enableIntrinsicRemember`, `enableNonSkippingGroupOptimization`, and `enableStrongSkippingMode`
- 2.1.0: report a warning for `stabilityConfigurationFile`
- 2.4.0: raise the warnings to errors

Report errors for obsolete Kotlin/Native Gradle task APIs

Issue: KT-85510

Component: Gradle

Incompatible change type: source

Short summary: The following deprecated Kotlin/Native Gradle task APIs now report an error when used:

`AbstractKotlinNativeCompile` properties:

- `additionalCompilerOptions`
- `languageSettings`
- `progressiveMode`

`KotlinNativeCompile` properties:

- `moduleName`
- `konanDataDir`
- `konanHome`
- `languageVersion`
- `apiVersion`
- `enabledLanguageFeatures`
- `optInAnnotationsInUse`
- `additionalCompilerOptions`

`CInteropProcess` properties:

- `outputFile`
- `konanDataDir`
- `konanHome`
- `defFile`

`KotlinNativeLink` properties:

- `languageSettings`
- `additionalCompilerOptions`
- `konanDataDir`
- `konanHome`

Additionally, the `KotlinNativeLink.compilation` property is removed.

Deprecation cycle:

- 2.4.0: report an error for the deprecated Kotlin/Native Gradle task APIs, remove the `KotlinNativeLink.compilation` property

Report warnings for case mismatches in compiler argument values

Issue: KT-86059

Component: Build tools API

Incompatible change type: source

Short summary: Compiler arguments that accept a fixed set of values previously handled letter case inconsistently: some accepted any letter case, while others required an exact match. The Build tools API now accepts any letter case for these values but reports a warning, for example `Case mismatch for -module-kind: expected 'commonjs', got 'CommonJS'`. To migrate, use the letter case listed for the argument in the compiler reference.

Deprecation cycle:

- 2.4.20: report a warning when the letter case of a compiler argument value doesn't match the expected value

Basic syntax overview

This is a collection of basic syntax elements with examples. At the end of every section, you'll find a link to a detailed description of the related topic.

You can also learn all the Kotlin essentials with the free Kotlin Core track by JetBrains Academy.

Package definition and imports

Package specification should be at the top of the source file:

```
package my.demo

import kotlin.text.*

// ...
```

It is not required to match directories and packages: source files can be placed arbitrarily in the file system.

See Packages.

Program entry point

An entry point of a Kotlin application is the `main` function:

```
fun main() {
    println("Hello world!")
}
```

Another form of `main` accepts a variable number of `String` arguments:

```
fun main(args: Array<String>) {
    println(args.contentToString())
}
```

Print to the standard output

`print` prints its argument to the standard output:

```
fun main() {
    print("Hello ")
    print("world!")
}
```

`println` prints its arguments and adds a line break, so that the next thing you print appears on the next line:

```
fun main() {
    println("Hello world!")
    println(42)
}
```

Read from the standard input

The `readln()` function reads from the standard input. This function reads the entire line the user enters as a string.

You can use the `println()`, `readln()`, and `print()` functions together to print messages requesting and showing user input:

```
// Prints a message to request input
println("Enter any word: ")

// Reads and stores the user input. For example: Happiness
val yourWord = readln()

// Prints a message with the input
print("You entered the word: ")
print(yourWord)
// You entered the word: Happiness
```

For more information, see Read standard input.

Functions

A function with two `Int` parameters and `Int` return type:

```
fun sum(a: Int, b: Int): Int {
    return a + b
}

fun main() {
    print("sum of 3 and 5 is ")
    println(sum(3, 5))
}
```

A function body can be an expression. Its return type is inferred:

```
fun sum(a: Int, b: Int) = a + b

fun main() {
    println("sum of 19 and 23 is ${sum(19, 23)}")
}
```

A function that returns no meaningful value:

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}

fun main() {
    printSum(-1, 8)
}
```

`Unit` return type can be omitted:

```
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}

fun main() {
    printSum(-1, 8)
}
```

See Functions.

Variables

In Kotlin, you declare a variable starting with a keyword, `val` or `var`, followed by the name of the variable.

Use the `val` keyword to declare variables that are assigned a value only once. These are immutable, read-only local variables that can't be reassigned a different value after initialization:

```
fun main() {
    // Declares the variable x and initializes it with the value of 5
    val x: Int = 5
    // 5
    println(x)
}
```

Use the `var` keyword to declare variables that can be reassigned. These are mutable variables, and you can change their values after initialization:

```
fun main() {
    // Declares the variable x and initializes it with the value of 5
    var x: Int = 5
    // Reassigns a new value of 6 to the variable x
    x += 1
    // 6
    println(x)
}
```

Kotlin supports type inference and automatically identifies the data type of a declared variable. When declaring a variable, you can omit the type after the variable name:

```
fun main() {
    // Declares the variable x with the value of 5;`Int` type is inferred
    val x = 5
    // 5
    println(x)
}
```

You can use variables only after initializing them. You can either initialize a variable at the moment of declaration or declare a variable first and initialize it later. In the second case, you must specify the data type:

```
fun main() {
    // Initializes the variable x at the moment of declaration; type is not required
    val x = 5
    // Declares the variable c without initialization; type is required
    val c: Int
    // Initializes the variable c after declaration
    c = 3
    // 5
    // 3
    println(x)
    println(c)
}
```

You can declare variables at the top level:

```
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
// x = 0; PI = 3.14
// incrementX()
// x = 1; PI = 3.14

fun main() {
    println("x = $x; PI = $PI")
    incrementX()
    println("incrementX()")
    println("x = $x; PI = $PI")
}
```

For information about declaring properties, see Properties.

Creating classes and instances

To define a class, use the `class` keyword:

```
class Shape
```

Properties of a class can be listed in its declaration or body:

```
class Rectangle(val height: Double, val length: Double) {
    val perimeter = (height + length) * 2
}
```

The default constructor with parameters listed in the class declaration is available automatically:

```
class Rectangle(val height: Double, val length: Double) {
    val perimeter = (height + length) * 2
}
fun main() {
    val rectangle = Rectangle(5.0, 2.0)
    println("The perimeter is ${rectangle.perimeter}")
}
```

Inheritance between classes is declared by a colon (`:`). Classes are `final` by default; to make a class inheritable, mark it as `open` :

```
open class Shape

class Rectangle(val height: Double, val length: Double): Shape() {
    val perimeter = (height + length) * 2
}
```

For more information about constructors and inheritance, see Classes and Objects and instances.

Comments

Just like most modern languages, Kotlin supports single-line (or *end-of-line*) and multi-line (*block*) comments:

```
// This is an end-of-line comment

/* This is a block comment
   on multiple lines. */
```

Block comments in Kotlin can be nested:

```
/* The comment starts here
/* contains a nested comment */
and ends here. */
```

See Documenting Kotlin Code for information on the documentation comment syntax.

String templates

```
fun main() {
    var a = 1
    // simple name in template:
    val s1 = "a is $a"

    a = 2
    // arbitrary expression in template:
    val s2 = "${s1.replace("is", "was")}, but now is $a"
    println(s2)
}
```

See String templates for details.

Conditional expressions

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

In Kotlin, `if` can also be used as an expression:

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

See `if` -expressions.

for loop

```
fun main() {
    val items = listOf("apple", "banana", "kiwifruit")
    for (item in items) {
        println(item)
    }
}
```

or:

```
fun main() {
    val items = listOf("apple", "banana", "kiwifruit")
    for (index in items.indices) {
        println("item at $index is ${items[index]}")
    }
}
```

See for loop.

while loop

```
fun main() {
    val items = listOf("apple", "banana", "kiwifruit")
    var index = 0
    while (index < items.size) {
        println("item at $index is ${items[index]}")
        index++
    }
}
```

See while loop.

when expression

```
fun describe(obj: Any): String =
    when (obj) {
        1          → "One"
        "Hello"    → "Greeting"
        is Long    → "Long"
        !is String → "Not a string"
        else       → "Unknown"
    }

fun main() {
    println(describe(1))
    println(describe("Hello"))
    println(describe(1000L))
    println(describe(2))
    println(describe("other"))
}
```

See when expressions and statements.

Ranges

Check if a number is within a range using `in` operator:

```
fun main() {
    val x = 10
    val y = 9
    if (x in 1..y+1) {
        println("fits in range")
    }
}
```

Check if a number is out of range:

```
fun main() {
    val list = listOf("a", "b", "c")

    if (-1 !in 0..list.lastIndex) {
        println("-1 is out of range")
    }
    if (list.size !in list.indices) {
        println("list size is out of valid list indices range, too")
    }
}
```

Iterate over a range:

```
fun main() {
    for (x in 1..5) {
        print(x)
    }
}
```

Or over a progression:

```
fun main() {
    for (x in 1..10 step 2) {
        print(x)
    }
    println()
    for (x in 9 downTo 0 step 3) {
        print(x)
    }
}
```

See Ranges and progressions.

Collections

Iterate over a collection:

```
fun main() {
    val items = listOf("apple", "banana", "kiwifruit")
    for (item in items) {
        println(item)
    }
}
```

Check if a collection contains an object using `in` operator:

```
fun main() {
    val items = setOf("apple", "banana", "kiwifruit")
    when {
        "orange" in items → println("juicy")
        "apple" in items → println("apple is fine too")
    }
}
```

Use lambda expressions to filter and map collections:

```
fun main() {
    val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
    fruits
      .filter { it.startsWith("a") }
      .sortedBy { it }
      .map { it.uppercase() }
      .forEach { println(it) }
}
```

See Collections overview.

Nullable values and null checks

A reference must be explicitly marked as nullable when a `null` value is possible. Nullable type names have `?` at the end. For example, `Int?` .

Return `null` if `str` does not hold an integer:

```
fun parseInt(str: String): Int? {
    return str.toIntOrNull()
}
```

Use a function returning nullable value:

```
fun parseInt(str: String): Int? {
    return str.toIntOrNull()
}

fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // Using `x * y` yields error because they may hold nulls.
    if (x ≠ null && y ≠ null) {
        // x and y are automatically cast to non-nullable after null check
        println(x * y)
    }
    else {
        println("'$arg1' or '$arg2' is not a number")
    }
}

fun main() {
    printProduct("6", "7")
    printProduct("a", "7")
    printProduct("a", "b")
}
```

or:

```
fun parseInt(str: String): Int? {
    return str.toIntOrNull()
}

fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // ...
    if (x == null) {
        println("Wrong number format in arg1: '$arg1'")
        return
    }
    if (y == null) {
        println("Wrong number format in arg2: '$arg2'")
        return
    }

    // x and y are automatically cast to non-nullable after null check
    println(x * y)
}

fun main() {
    printProduct("6", "7")
    printProduct("a", "7")
    printProduct("99", "b")
}
```

See Null-safety.

Type checks and automatic casts

The `is` operator checks if an expression is an instance of a type. If an immutable local variable or property is checked for a specific type, there's no need to cast it explicitly:

```
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` is automatically cast to `String` in this branch
        return obj.length
    }

    // `obj` is still of type `Any` outside of the type-checked branch
    return null
}

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result: ${getStringLength(obj) ?: "Error: The object is
not a string"} ")
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf(Any()))
}
```

or:

```
fun getStringLength(obj: Any): Int? {
    if (obj !is String) return null

    // `obj` is automatically cast to `String` in this branch
    return obj.length
}

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result: ${getStringLength(obj) ?: "Error: The object is
not a string"} ")
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf(Any()))
}
```

or even:

```
fun getStringLength(obj: Any): Int? {
    // `obj` is automatically cast to `String` on the right-hand side of `&&`
    if (obj is String && obj.length >= 0) {
        return obj.length
    }

    return null
}

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result: ${getStringLength(obj) ?: "Error: The object is
not a string"} ")
    }
    printLength("Incomprehensibilities")
    printLength("")
    printLength(1000)
}
```

See Classes and Type casts.

Keywords and operators

Hard keywords

The following tokens are always interpreted as keywords and cannot be used as identifiers:

- `as`
  - is used for type casts.
  - specifies an alias for an import
- `as?` is used for safe type casts.
- `break` terminates the execution of a loop.
- `class` declares a class.
- `continue` proceeds to the next step of the nearest enclosing loop.
- `do` begins a do/while loop (a loop with a postcondition).
- `else` defines the branch of an if expression that is executed when the condition is false.
- `false` specifies the 'false' value of the Boolean type.
- `for` begins a for loop.
- `fun` declares a function.
- `if` begins an if expression.
- `in`
  - specifies the object being iterated in a for loop.
  - is used as an infix operator to check that a value belongs to a range, a collection, or another entity that defines a 'contains' method.

    - is used in when expressions for the same purpose.
    - marks a type parameter as contravariant.
- `!in`
    - is used as an operator to check that a value does NOT belong to a range, a collection, or another entity that defines a 'contains' method.
    - is used in when expressions for the same purpose.
- `interface` declares an interface.
- `is`
    - checks that a value has a certain type.
    - is used in when expressions for the same purpose.
- `!is`
    - checks that a value does NOT have a certain type.
    - is used in when expressions for the same purpose.
- `null` is a constant representing an object reference that doesn't point to any object.
- `object` declares a class and its instance at the same time.
- `package` specifies the package for the current file.
- `return` returns from the nearest enclosing function or anonymous function.
- `super`
    - refers to the superclass implementation of a method or property.
    - calls the superclass constructor from a secondary constructor.
- `this`
    - refers to the current receiver.
    - calls another constructor of the same class from a secondary constructor.
- `throw` throws an exception.
- `true` specifies the 'true' value of the Boolean type.
- `try` begins an exception-handling block.
- `typealias` declares a type alias.
- `typeof` is reserved for future use.
- `val` declares a read-only property or local variable.
- `var` declares a mutable property or local variable.
- `when` begins a when expression (executes one of the given branches).
- `while` begins a while loop (a loop with a precondition).

Soft keywords

The following tokens act as keywords in the context in which they are applicable, and they can be used as identifiers in other contexts:

- `by`
  - delegates the implementation of an interface to another object.
  - delegates the implementation of the accessors for a property to another object.
- `catch` begins a block that handles a specific exception type.
- `constructor` declares a primary or secondary constructor.
- `delegate` is used as an annotation use-site target.
- `dynamic` references a dynamic type in Kotlin/JS code.
- `field`
  - declares an explicit backing field.
  - is used as an annotation use-site target.
- `file` is used as an annotation use-site target.
- `finally` begins a block that is always executed when a try block exits.
- `get`
  - declares the getter of a property.
  - is used as an annotation use-site target.
- `import` imports a declaration from another package into the current file.
- `init` begins an initializer block.
- `param` is used as an annotation use-site target.
- `property` is used as an annotation use-site target.
- `receiver` is used as an annotation use-site target.
- `set`
  - declares the setter of a property.
  - is used as an annotation use-site target.
- `setparam` is used as an annotation use-site target.
- `value` with the `class` keyword declares an inline class.
- `where` specifies the constraints for a generic type parameter.

Modifier keywords

The following tokens act as keywords in modifier lists of declarations, and they can be used as identifiers in other contexts:

- `abstract` marks a class or member as abstract.

- `actual` denotes a platform-specific implementation in multiplatform projects.
- `annotation` declares an annotation class.
- `companion` declares a companion object.
- `const` marks a property as a compile-time constant.
- `crossinline` forbids non-local returns in a lambda passed to an inline function.
- `data` instructs the compiler to generate canonical members for a class.
- `enum` declares an enumeration.
- `expect` marks a declaration as platform-specific, expecting an implementation in platform modules.
- `external` marks a declaration as implemented outside of Kotlin (accessible through JNI or in JavaScript).
- `final` forbids overriding a member.
- `infix` allows calling a function using infix notation.
- `inline` tells the compiler to inline a function and the lambdas passed to it at the call site.
- `inner` allows referring to an outer class instance from a nested class.
- `internal` marks a declaration as visible in the current module.
- `lateinit` allows initializing a non-nullable property outside of a constructor.
- `noinline` turns off inlining of a lambda passed to an inline function.
- `open` allows subclassing a class or overriding a member.
- `operator` marks a function as overloading an operator or implementing a convention.
- `out` marks a type parameter as covariant.
- `override` marks a member as an override of a superclass member.
- `private` marks a declaration as visible in the current class or file.
- `protected` marks a declaration as visible in the current class and its subclasses.
- `public` marks a declaration as visible anywhere.
- `reified` marks a type parameter of an inline function as accessible at runtime.
- `sealed` declares a sealed class (a class with restricted subclassing).
- `suspend` marks a function or lambda as suspending (usable as a coroutine).
- `tailrec` marks a function as tail-recursive (allowing the compiler to replace recursion with iteration).
- `vararg` allows passing a variable number of arguments for a parameter.

Special identifiers

The following identifiers are defined by the compiler in specific contexts, and they can be used as regular identifiers in other contexts:

- `field` is used inside a property accessor to refer to the backing field of the property.
- `it` is used inside a lambda to refer to its parameter implicitly.

Operators and special symbols

Kotlin supports the following operators and special symbols:

- `+`, `-`, `*`, `/`, `%` - mathematical operators
  - `*` is also used to pass an array to a vararg parameter.
- `=`
  - assignment operator.
  - is used to specify default values for parameters.
- `+=`, `-=`, `*=`, `/=`, `%=` - augmented assignment operators.
- `++`, `--` - increment and decrement operators.
- `&&`, `||`, `!` - logical 'and', 'or', 'not' operators (for bitwise operations, use the corresponding infix functions instead).
- `==`, `!=` - equality operators (translated to calls of `equals()` for non-primitive types).
- `===`, `!==` - referential equality operators.
- `<`, `>`, `<=`, `>=` - comparison operators (translated to calls of `compareTo()` for non-primitive types).
- `[`, `]` - indexed access operator (translated to calls of `get` and `set`).
- `!!` asserts that an expression is non-nullable.
- `?.` performs a safe call (calls a method or accesses a property if the receiver is non-nullable).
- `?:` takes the right-hand value if the left-hand value is null (the elvis operator).
- `::` creates a member reference or a class reference.
- `.`
  - accesses members, including nested classes and enum entries.
  - defines and calls extensions.
  - qualifies names and packages.
  - separates the integer and fractional parts of a floating-point literal.
- `..`, `..<` create ranges.
- `:` separates a name from a type in a declaration.
- `?` marks a type as nullable.
- `->`
  - separates the parameters and body of a lambda expression.
  - separates the parameters and return type declaration in a function type.
  - separates the condition and body of a when expression branch.
- `@`
  - introduces an annotation.
  - introduces or references a loop label.

- introduces or references a lambda label.
- references a 'this' expression from an outer scope.
- references an outer superclass.

- `;` separates multiple statements on the same line.
- `$` references a variable or expression in a string template.
- `_`
  - substitutes an unused parameter in a lambda expression.
  - substitutes an unused parameter in a destructuring declaration.

For operator precedence, see this reference in Kotlin grammar.

Packages and imports

In a Kotlin project, code is organized using packages and imports:

- A **package** is a container for one or more Kotlin files. Files are linked to a package using a `package` header.
- An **import** is a directive that makes entities from other packages available in the current file.

Package headers

A source file may start with a package header:

```
package org.example

fun printMessage() { /*...*/ }
class Message(val text: String) { /*...*/ }
```

All contents of the source file, such as classes and functions, belong to this package. Their fully qualified name combines the package name with the entity's name. In this example:

- The fully qualified name of `printMessage()` is `org.example.printMessage`.
- The fully qualified name of `Message` is `org.example.Message`.

If a file has no package header, its contents belong to the root package.

Imports

To use an entity from a file in a different package, use an `import` directive. In addition to the default imports, each file may declare its own imports.

Import a single entity

Import a specific entity so you can use it without qualification:

```
// Message is accessible without qualification
import org.example.Message

fun main() {
    val message = Message("Hello")
    println(message.text)
}
```

Import the contents of a scope

Star imports, ending in an asterisk `*`, import all named entities inside the corresponding scope:

```
// Everything in org.example is accessible
import org.example.*

fun main() {
    printMessage()
    val message = Message("Hi")
}
```

If you import an entity with both a star import and an explicit import, the explicit import takes priority during overload resolution.

Resolve name clashes with aliases

If two imported entities have the same name, use the `as` keyword to locally rename one of them:

```
// Message refers to org.example.Message
import org.example.Message

// TestMessage refers to org.test.Message
import org.test.Message as TestMessage

fun main() {
    val a = Message("from example")
    val b = TestMessage("from test")
}
```

What you can import

The `import` keyword is not limited to classes. You can import any of the following entities, whether they come from a package, a class, an object, or an enum:

- Top-level functions and properties declared directly inside a package:

  ```
  import org.example.printMessage // Top-level function
  import org.example.VERSION      // Top-level property
  ```

- Functions and properties from object declarations:

  ```
  import org.example.Config.DEFAULT_TIMEOUT // Property from an object
  import org.example.Config.loadSettings    // Function from an object
  ```

- Members of a companion object, referenced through the enclosing class name:

```
import org.example.MyClass.create // Refers to MyClass.Companion.create
```

- Enum constants:

```
import org.example.Color.RED
import org.example.Color.GREEN
```

- Nested classes:

```
import org.example.Outer.Nested
```

Default imports

Kotlin includes the following imports by default:

- kotlin.*
- kotlin.annotation.*
- kotlin.collections.*
- kotlin.comparisons.*
- kotlin.io.*
- kotlin.ranges.*
- kotlin.sequences.*
- kotlin.text.*
- kotlin.math.*

Kotlin imports additional packages depending on the target platform:

- JVM:
  - java.lang.*
  - kotlin.jvm.*
- JS:
  - kotlin.js.*

Visibility and imports

The ability to import an entity depends on its visibility modifiers:

- `public` entities can be imported anywhere.
- `internal` entities can be imported only within the same module.
- `protected` entities cannot be imported.
- Top-level `private` entities are only accessible within their declaring file.
- Other `private` entities cannot be imported.

Annotations

Annotations are tags that you can use to attach metadata to elements in your code. Tools and frameworks process this metadata during compilation and runtime, and perform different actions based on it.

You can annotate your code to simplify and automate common tasks, such as generating boilerplate code, enforcing coding standards or writing documentation.

> If you want to develop your own annotation processors, you can use the Kotlin Symbol Processing (KSP) API.

Declaration

Annotations are a special type of class. To declare an annotation, use the `annotation` keyword before the class declaration:

```
annotation class Fancy
```

Additional attributes of the annotation can be specified by annotating the annotation class with meta-annotations:

- `@Target` specifies the possible kinds of elements which can be annotated with the annotation (such as classes, functions, properties, and expressions);
- `@Retention` specifies whether the annotation is stored in the compiled class files and whether it's visible through reflection at runtime (by default, both are true);
- `@Repeatable` allows using the same annotation on a single element multiple times;
- `@MustBeDocumented` specifies that the annotation is part of the public API and should be included in the class or method signature shown in the generated API documentation.

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.TYPE_PARAMETER, AnnotationTarget.VALUE_PARAMETER,
        AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Fancy
```

Usage

```
@Fancy class Foo {
    @Fancy fun baz(@Fancy foo: Int): Int {
        return (@Fancy 1)
    }
}
```

If you need to annotate the primary constructor of a class, you need to add the `constructor` keyword to the constructor declaration, and add the annotations before it:

```
class Foo @Inject constructor(dependency: MyDependency) { ... }
```

You can also annotate property accessors:

```
class Foo {
    var x: MyDependency? = null
        @Inject set
}
```

Constructors

Annotations can have constructors that take parameters.

```
annotation class Special(val why: String)

@Special("example") class Foo {}
```

Allowed parameter types are:

- Types that correspond to Java primitive types (Int, Long etc.)
- Strings
- Classes (`Foo::class`)
- Enums
- Other annotations
- Arrays of the types listed above

Annotation parameters cannot have nullable types, because the JVM does not support storing `null` as a value of an annotation attribute.

If an annotation is used as a parameter of another annotation, its name is not prefixed with the `@` character:

```
annotation class ReplaceWith(val expression: String)

annotation class Deprecated(
        val message: String,
        val replaceWith: ReplaceWith = ReplaceWith(""))

@Deprecated("This function is deprecated, use === instead", ReplaceWith("this === other"))
```

If you need to specify a class as an argument of an annotation, use a Kotlin class (KClass). The Kotlin compiler will automatically convert it to a Java class, so that the Java code can access the annotations and arguments normally.

```
import kotlin.reflect.KClass

annotation class Ann(val arg1: KClass<*>, val arg2: KClass<out Any>)

@Ann(String::class, Int::class) class MyClass
```

Instantiation

In Java, an annotation type is a form of an interface, so you can implement it and use an instance. As an alternative to this mechanism, Kotlin lets you call a constructor of an annotation class in arbitrary code and similarly use the resulting instance.

```
annotation class InfoMarker(val info: String)

fun processInfo(marker: InfoMarker): Unit = TODO()

fun main(args: Array<String>) {
    if (args.isNotEmpty())
        processInfo(getAnnotationReflective(args))
    else
        processInfo(InfoMarker("default"))
}
```

Learn more about instantiation of annotation classes in this KEEP.

Lambdas

Annotations can also be used on lambdas. They will be applied to the `invoke()` method into which the body of the lambda is generated. This is useful for frameworks like Quasar, which uses annotations for concurrency control.

```
annotation class Suspendable

val f = @Suspendable { Fiber.sleep(10) }
```

Annotation use-site targets

When you're annotating a property or a primary constructor parameter, there are multiple Java elements that are generated from the corresponding Kotlin element, and therefore multiple possible locations for the annotation in the generated Java bytecode. To specify how exactly the annotation should be generated, use the following syntax:

```
class Example(@field:Ann val foo,    // annotate only the Java field
              @get:Ann val bar,      // annotate only the Java getter
              @param:Ann val quux)   // annotate only the Java constructor parameter
```

The same syntax can be used to annotate the entire file. To do this, put an annotation with the target `file` at the top level of a file, before the package directive or before all imports if the file is in the default package:

```
@file:JvmName("Foo")

package org.jetbrains.demo
```

If you have multiple annotations with the same target, you can avoid repeating the target by adding brackets after the target and putting all the annotations inside the brackets (except for the `all` meta-target):

```
class Example {
     @set:[Inject VisibleForTesting]
     var collaborator: Collaborator
}
```

The full list of supported use-site targets is:

- `file`
- `field`

- `property` (annotations with this target are not visible to Java)
- `get` (property getter)
- `set` (property setter)
- `all` (a meta-target for properties, see the `all` meta-target section for more information)
- `receiver` (receiver parameter of an extension function or property)

  To annotate the receiver parameter of an extension function, use the following syntax:

  ```
  fun @receiver:Fancy String.myExtension() { ... }
  ```

- `param` (constructor parameter)
- `setparam` (property setter parameter)
- `delegate` (the field storing the delegate instance for a delegated property)

Defaults when no use-site targets are specified

If you don't specify a use-site target, the compiler chooses the target according to the `@Target` annotation of the annotation you use. If there are multiple applicable targets, the compiler chooses one or more of them in the following order:

- The constructor parameter target (`param`).
- The property target (`property`).
- The field target (`field`), if it's applicable and the property target (`property`) isn't.

If none of `param` , `property` , or `field` are applicable, the annotation is invalid, and you need to specify a use-site target explicitly.

Let's use the `@Email` annotation from Jakarta Bean Validation:

```
@Target(value={METHOD,FIELD,ANNOTATION_TYPE,CONSTRUCTOR,PARAMETER,TYPE_USE})
public @interface Email { }
```

With this annotation, consider the following example:

```
data class User(val username: String,
                // @Email is now equivalent to @param:Email @field:Email
                @Email val email: String) {
    // @Email is still equivalent to @field:Email
    @Email val secondaryEmail: String? = null
}
```

In this example, the `@Email` annotation applies to both the constructor parameter and the field targets for the `email` property because the property:

- Is declared in the primary constructor.
- Has no custom getter or setter, so the compiler generates a backing field.

The `@Email` annotation only applies to the field target for the `secondaryEmail` property because the property:

- Isn't declared in the primary constructor.
- Has no custom getter or setter, so the compiler generates a backing field.

`all` meta-target

The `all` target makes it easier to apply the same annotation not only to the parameter and the property or field, but also to the corresponding getter and setter.

Specifically, the annotation marked with `all` is propagated, if applicable:

- To the constructor parameter (`param`) if the property is defined in the primary constructor.
- To the property itself (`property`).
- To the backing field (`field`) if the property has one.
- To the getter (`get`).
- To the setter parameter (`setparam`) if the property is defined as `var` .
- To the Java-only target `RECORD_COMPONENT` if the class has the `@JvmRecord` annotation.

Let's use the `@Email` annotation from Jakarta Bean Validation, which is defined as follows:

```
@Target(value={METHOD,FIELD,ANNOTATION_TYPE,CONSTRUCTOR,PARAMETER,TYPE_USE})
public @interface Email { }
```

In the example below, this `@Email` annotation is applied to all relevant targets:

```
data class User(
    val username: String,
    // Applies @Email to param, field, and get
    @all:Email val email: String,
    // Applies @Email to param, field, get, and setparam
    @all:Email var name: String,
) {
    // Applies @Email to field and getter (not param since it's not in the constructor)
    @all:Email val secondaryEmail: String? = null
}
```

You can use the `all` meta-target with any property, both inside and outside the primary constructor.

Limitations

The `all` target comes with some limitations:

- It does not propagate an annotation to types, potential extension receivers, or context receivers or parameters.
- It cannot be used with multiple annotations:

  ```
  @all:[A B] // forbidden, use @all:A @all:B
  val x: Int = 5
  ```

- It cannot be used with delegated properties.

Java annotations

Java annotations are 100% compatible with Kotlin:

```
import org.junit.Test
import org.junit.Assert.*
import org.junit.Rule
import org.junit.rules.*

class Tests {
    // apply @Rule annotation to property getter
    @get:Rule val tempFolder = TemporaryFolder()

    @Test fun simple() {
        val f = tempFolder.newFile()
        assertEquals(42, getTheAnswer())
    }
}
```

Since the order of parameters for an annotation written in Java is not defined, you can't use a regular function call syntax for passing the arguments. Instead, you need to use the named argument syntax:

```
// Java
public @interface Ann {
    int intValue();
    String stringValue();
}
```

```
// Kotlin
@Ann(intValue = 1, stringValue = "abc") class C
```

Just like in Java, a special case is the `value` parameter; its value can be specified without an explicit name:

```
// Java
public @interface AnnWithValue {
    String value();
}
```

```
// Kotlin
@AnnWithValue("abc") class C
```

Arrays as annotation parameters

If the `value` argument in Java has an array type, it becomes a `vararg` parameter in Kotlin:

```
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
```

```
// Kotlin
@AnnWithArrayValue("abc", "foo", "bar") class C
```

For other arguments that have an array type, you need to use the array literal syntax or `arrayOf(...)`:

```
// Java
public @interface AnnWithArrayMethod {
    String[] names();
}
```

```
@AnnWithArrayMethod(names = ["abc", "foo", "bar"])
class C
```

Accessing properties of an annotation instance

Values of an annotation instance are exposed as properties to Kotlin code:

```
// Java
public @interface Ann {
    int value();
}
```

```
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```

Ability to not generate JVM 1.8+ annotation targets

If a Kotlin annotation has `TYPE` among its Kotlin targets, the annotation maps to `java.lang.annotation.ElementType.TYPE_USE` in its list of Java annotation targets. This is just like how the `TYPE_PARAMETER` Kotlin target maps to the `java.lang.annotation.ElementType.TYPE_PARAMETER` Java target. This is an issue for Android clients with API levels less than 26, which don't have these targets in the API.

To avoid generating the `TYPE_USE` and `TYPE_PARAMETER` annotation targets, use the new compiler argument `-Xno-new-java-annotation-targets`.

Repeatable annotations

Just like in Java, Kotlin has repeatable annotations, which can be applied to a single code element multiple times. To make your annotation repeatable, mark its declaration with the `@kotlin.annotation.Repeatable` meta-annotation. This will make it repeatable both in Kotlin and Java. Java repeatable annotations are also supported from the Kotlin side.

The main difference with the scheme used in Java is the absence of a *containing annotation*, which the Kotlin compiler generates automatically with a predefined name. For an annotation in the example below, it will generate the containing annotation `@Tag.Container`:

```
@Repeatable
annotation class Tag(val name: String)

// The compiler generates the @Tag.Container containing annotation
```

You can set a custom name for a containing annotation by applying the `@kotlin.jvm.JvmRepeatable` meta-annotation and passing an explicitly declared containing annotation class as an argument:

```
@JvmRepeatable(Tags::class)
annotation class Tag(val name: String)

annotation class Tags(val value: Array<Tag>)
```

To extract Kotlin or Java repeatable annotations via reflection, use the `KAnnotatedElement.findAnnotations()` function.

Learn more about Kotlin repeatable annotations in this KEEP.

Visibility modifiers

Classes, objects, interfaces, constructors, and functions, as well as properties and their setters, can have *visibility modifiers*. Getters always have the same visibility as their properties.

There are four visibility modifiers in Kotlin: `private`, `protected`, `internal`, and `public`. The default visibility is `public`.

On this page, you'll learn how the modifiers apply to different types of declaring scopes.

Packages

Functions, properties, classes, objects, and interfaces can be declared at the "top-level" directly inside a package:

```
// file name: example.kt
package foo

fun baz() { ... }
class Bar { ... }
```

- If you don't use a visibility modifier, `public` is used by default, which means that your declarations will be visible everywhere.
- If you mark a declaration as `private`, it will only be visible inside the file that contains the declaration.
- If you mark it as `internal`, it will be visible everywhere in the same module.
- The `protected` modifier is not available for top-level declarations.

To use a visible top-level declaration from another package, you should import it.

Examples:

```
// file name: example.kt
package foo

private fun foo() { ... } // visible inside example.kt

public var bar: Int = 5 // property is visible everywhere
    private set         // setter is visible only in example.kt

internal val baz = 6    // visible inside the same module
```

Class members

For members declared inside a class:

- `private` means that the member is visible inside this class only (including all its members).
- `protected` means that the member has the same visibility as one marked as `private`, but that it is also visible in subclasses.

- `internal` means that any client *inside this module* who sees the declaring class sees its `internal` members.
- `public` means that any client who sees the declaring class sees its `public` members.

> In Kotlin, an outer class does not see private members of its inner classes.

If you override a `protected` or an `internal` member and do not specify the visibility explicitly, the overriding member will also have the same visibility as the original.

Examples:

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal open val c = 3
    val d = 4  // public by default

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
    // a is not visible
    // b, c and d are visible
    // Nested and e are visible

    override val b = 5   // 'b' is protected
    override val c = 7   // 'c' is internal
}

class Unrelated(o: Outer) {
    // o.a, o.b are not visible
    // o.c and o.d are visible (same module)
    // Outer.Nested is not visible, and Nested::e is not visible either
}
```

Constructors

Use the following syntax to specify the visibility of the primary constructor of a class:

> You need to add an explicit `constructor` keyword.

```
class C private constructor(a: Int) { ... }
```

Here the constructor is `private`. By default, all constructors are `public`, which effectively amounts to them being visible everywhere the class is visible (this means that a constructor of an `internal` class is only visible within the same module).

For sealed classes, constructors are `protected` by default. For more information, see Sealed classes.

Local declarations

Local variables, functions, and classes can't have visibility modifiers.

Modules

The `internal` visibility modifier means that the member is visible within the same module. More specifically, a module is a set of Kotlin files compiled together, for example:

- An IntelliJ IDEA module.
- A Maven project.
- A Gradle source set (with the exception that the `test` source set can access the internal declarations of `main`).

Coding conventions

Commonly known and easy-to-follow coding conventions are vital for any programming language. Here we provide guidelines on the code style and code organization for projects that use Kotlin.

Configure style in IDE

Two most popular IDEs for Kotlin - IntelliJ IDEA and Android Studio provide powerful support for code styling. You can configure them to automatically format your code in consistence with the given code style.

Apply the style guide

1. Go to **Settings/Preferences | Editor | Code Style | Kotlin**.
2. Click **Set from...**.
3. Select **Kotlin style guide**.

Verify that your code follows the style guide

1. Go to **Settings/Preferences | Editor | Inspections | General**.
2. Switch on **Incorrect formatting** inspection. Additional inspections that verify other issues described in the style guide (such as naming conventions) are enabled by default.

For more information, see the Migrate to Kotlin code style with IntelliJ IDEA guide.

Source code organization

Directory structure

In pure Kotlin projects, the recommended directory structure follows the package structure with the common root package omitted. For example, if all the code in the project is in the `org.example.kotlin` package and its subpackages, files with the `org.example.kotlin` package should be placed directly under the source root, and files in `org.example.kotlin.network.socket` should be in the `network/socket` subdirectory of the source root.

> On JVM: In projects where Kotlin is used together with Java, Kotlin source files should reside in the same source root as the Java source files, and follow the same directory structure: each file should be stored in the directory corresponding to each package statement.

Source file names

If a Kotlin file contains a single class or interface (potentially with related top-level declarations), its name should be the same as the name of the class, with the `.kt` extension appended. It applies to all types of classes and interfaces. If a file contains multiple classes, or only top-level declarations, choose a name describing what the file contains, and name the file accordingly. Use upper camel case, where the first letter of each word is capitalized. For example, `ProcessDeclarations.kt`.

The name of the file should describe what the code in the file does. Therefore, you should avoid using meaningless words such as `Util` in file names.

Multiplatform projects

In multiplatform projects, files with top-level declarations in platform-specific source sets should have a suffix associated with the name of the source set. For example:

- **jvm**Main/kotlin/Platform. **jvm** .kt
- **android**Main/kotlin/Platform. **android** .kt
- **ios**Main/kotlin/Platform. **ios** .kt

As for the common source set, files with top-level declarations should not have a suffix. For example, `commonMain/kotlin/Platform.kt`.

Technical details

We recommend following this file naming scheme in multiplatform projects due to JVM limitations: it doesn't allow top-level members (functions, properties).

To work around this, the Kotlin JVM compiler creates wrapper classes (so-called "file facades") that contain top-level member declarations. File facades have an internal name derived from the file name.

In turn, JVM doesn't allow several classes with the same fully qualified name (FQN). This might lead to situations when a Kotlin project cannot be compiled to JVM:

```
root
├─ commonMain/kotlin/myPackage/Platform.kt // contains 'fun count() { }'
├─ jvmMain/kotlin/myPackage/Platform.kt // contains 'fun multiply() { }'
```

Here both `Platform.kt` files are in the same package, so the Kotlin JVM compiler produces two file facades, both of which have FQN `myPackage.PlatformKt`. This produces the "Duplicate JVM classes" error.

The simplest way to avoid that is renaming one of the files according to the guideline above. This naming scheme helps avoid clashes while retaining code readability.

There are two scenarios where these recommendations may seem redundant, but we still advise to follow them:

- Non-JVM platforms don't have issues with duplicating file facades. However, this naming scheme can help you keep file naming consistent.
- On JVM, if source files don't have top-level declarations, the file facades aren't generated, and you won't face naming clashes.

  However, this naming scheme can help you avoid situations when a simple refactoring or an addition could include a top-level function and result in the same "Duplicate JVM classes" error.

Source file organization

Placing multiple declarations (classes, top-level functions or properties) in the same Kotlin source file is encouraged as long as these declarations are closely related to each other semantically, and the file size remains reasonable (not exceeding a few hundred lines).

In particular, when defining extension functions for a class which are relevant for all clients of this class, put them in the same file with the class itself. When defining extension functions that make sense only for a specific client, put them next to the code of that client. Avoid creating files just to hold all extensions of some class.

Class layout

The contents of a class should go in the following order:

1. Property declarations and initializer blocks
2. Secondary constructors
3. Method declarations
4. Companion object

Do not sort the method declarations alphabetically or by visibility, and do not separate regular methods from extension methods. Instead, put related stuff together, so that someone reading the class from top to bottom can follow the logic of what's happening. Choose an order (either higher-level stuff first, or vice versa) and stick to it.

Put nested classes next to the code that uses those classes. If the classes are intended to be used externally and aren't referenced inside the class, put them in the end, after the companion object.

Interface implementation layout

When implementing an interface, keep the implementing members in the same order as members of the interface (if necessary, interspersed with additional private methods used for the implementation).

Overload layout

Always put overloads next to each other in a class.

Naming rules

Package and class naming rules in Kotlin are quite simple:

- Names of packages are always lowercase and do not use underscores (`org.example.project`). Using multi-word names is generally discouraged, but if you do need to use multiple words, you can either just concatenate them together or use camel case (`org.example.myProject`).
- Names of classes and objects use upper camel case:

```
open class DeclarationProcessor { /*...*/ }

object EmptyDeclarationProcessor : DeclarationProcessor() { /*...*/ }
```

Function names

Names of functions, properties, and local variables start with a lowercase letter and use camel case without underscores:

```
fun processDeclarations() { /*...*/ }
var declarationCount = 1
```

Names for class-like functions

There are two exceptions where function names should follow class-naming convention instead. Functions of this kind are usually defined at the top level.

- Factory functions that create class instances can have the same name as the abstract return type:

```
interface Foo { /*...*/ }

class FooImpl : Foo { /*...*/ }

fun Foo(): Foo { return FooImpl() }
```

- `@Composable` functions that return `Unit`:

```
@Composable fun TabHeader { /*...*/ }
```

Names for test methods

In tests (and **only** in tests), you can use method names with spaces enclosed in backticks. Note that such method names are only supported by Android runtime from API level 30. Underscores in method names are also allowed in test code.

```
class MyTestCase {
    @Test fun `ensure everything works`() { /*...*/ }

    @Test fun ensureEverythingWorks_onAndroid() { /*...*/ }
}
```

Property names

Names of constants (properties marked with `const`, or top-level or object `val` properties with no custom `get` function that hold deeply immutable data) should use all uppercase, underscore-separated names following the screaming snake case convention:

```
const val MAX_COUNT = 8
val USER_NAME_FIELD = "UserName"
```

Names of top-level or object properties which hold objects with behavior or mutable data should use camel case names:

```
val mutableCollection: MutableSet<String> = HashSet()
```

Names of properties holding references to singleton objects can use the same naming style as `object` declarations:

```
val PersonComparator: Comparator<Person> = /* ... */
```

For enum constants, it's OK to use either all uppercase, underscore-separated (screaming snake case) names (`enum class Color { RED, GREEN }`) or upper camel case names, depending on the usage.

Names for backing properties

If a class has two properties which are conceptually the same but one is part of a public API and another is an implementation detail, use an underscore as the prefix for the name of the private property:

```
class C {
    private val _elementList = mutableListOf<Element>()

    val elementList: List<Element>
        get() = _elementList
}
```

Choose good names

The name of a class is usually a noun or a noun phrase explaining what the class *is*: `List` , `PersonReader` .

The name of a method is usually a verb or a verb phrase saying what the method *does*: `close` , `readPersons` . The name should also suggest if the method is mutating the object or returning a new one. For instance, `sort` is sorting a collection in place, while `sorted` is returning a sorted copy of the collection.

The names should make it clear what the purpose of the entity is, so it's best to avoid using meaningless words (`Manager` , `Wrapper`) in names.

When using an acronym as part of a declaration name, follow these rules:

- For two-letter acronyms, use uppercase for both letters. For example, `IOStream` .
- For acronyms longer than two letters, capitalize only the first letter. For example, `XmlFormatter` or `HttpInputStream` .

Formatting

Indentation

Use four spaces for indentation. Do not use tabs.

For curly braces, put the opening brace at the end of the line where the construct begins, and the closing brace on a separate line aligned horizontally with the opening construct.

```
if (elements != null) {
    for (element in elements) {
        // ...
    }
}
```

In Kotlin, semicolons are optional, and therefore line breaks are significant. The language design assumes Java-style braces, and you may encounter surprising behavior if you try to use a different formatting style.

Horizontal whitespace

- Put spaces around binary operators (`a + b`). Exception: don't put spaces around the "range to" operator (`0..i`).
- Do not put spaces around unary operators (`a++`).
- Put spaces between control flow keywords (`if` , `when` , `for` , and `while`) and the corresponding opening parenthesis.
- Do not put a space before an opening parenthesis in a primary constructor declaration, method declaration or method call.

```
class A(val x: Int)

fun foo(x: Int) { ... }

fun bar() {
    foo(1)
}
```

- Never put a space after `(` , `[` , or before `]` , `)` .
- Never put a space around `.` or `?.` : `foo.bar().filter { it > 2 }.joinToString()` , `foo?.bar()` .
- Put a space after `//` : `// This is a comment` .
- Do not put spaces around angle brackets used to specify type parameters: `class Map<K, V> { ... }` .
- Do not put spaces around `::` : `Foo::class` , `String::length` .
- Do not put a space before `?` used to mark a nullable type: `String?` .

As a general rule, avoid horizontal alignment of any kind. Renaming an identifier to a name with a different length should not affect the formatting of either the declaration or any of the usages.

Colon

Put a space before `:` in the following scenarios:

- When it's used to separate a type and a supertype.
- When delegating to a superclass constructor or a different constructor of the same class.
- After the `object` keyword.

Don't put a space before `:` when it separates a declaration and its type.

Always put a space after `:` .

```
abstract class Foo<out T : Any> : IFoo {
    abstract fun foo(a: Int): T
}

class FooImpl : Foo() {
    constructor(x: String) : this(x) { /*...*/ }

    val x = object : IFoo { /*...*/ }
}
```

Class headers

Classes with a few primary constructor parameters can be written in a single line:

```
class Person(id: Int, name: String)
```

Classes with longer headers should be formatted so that each primary constructor parameter is in a separate line with indentation. Also, the closing parenthesis should be on a new line. If you use inheritance, the superclass constructor call, or the list of implemented interfaces should be located on the same line as the parenthesis:

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name) { /*...*/ }
```

For multiple interfaces, the superclass constructor call should be located first and then each interface should be located in a different line:

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name),
    KotlinMaker { /*...*/ }
```

For classes with a long supertype list, put a line break after the colon and align all supertype names horizontally:

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne {

    fun foo() { /*...*/ }
}
```

To clearly separate the class header and body when the class header is long, either put a blank line following the class header (as in the example above), or put the opening curly brace on a separate line:

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne
{
    fun foo() { /*...*/ }
}
```

Use regular indent (four spaces) for constructor parameters. This ensures that properties declared in the primary constructor have the same indentation as properties declared in the body of a class.

Modifiers order

If a declaration has multiple modifiers, always put them in the following order:

```
public / protected / private / internal
expect / actual
final / open / abstract / sealed / const
external
override
lateinit
tailrec
vararg
suspend
inner
enum / annotation / fun // as a modifier in `fun interface`
companion
inline / value
infix
operator
data
```

Place all annotations before modifiers:

```
@Named("Foo")
private val foo: Foo
```

Unless you're working on a library, omit redundant modifiers (for example, `public`).

Annotations

Place annotations on separate lines before the declaration to which they are attached, and with the same indentation:

```
@Target(AnnotationTarget.PROPERTY)
annotation class JsonExclude
```

Annotations without arguments may be placed on the same line:

```
@JsonExclude @JvmField
var x: String
```

A single annotation without arguments may be placed on the same line as the corresponding declaration:

```
@Test fun foo() { /*...*/ }
```

File annotations

File annotations are placed after the file comment (if any), before the `package` statement, and are separated from `package` with a blank line (to emphasize the fact that they target the file and not the package).

```
/** License, copyright and whatever */
@file:JvmName("FooBar")

package foo.bar
```

Functions

If the function signature doesn't fit on a single line, use the following syntax:

```
fun longMethodName(
    argument: ArgumentType = defaultValue,
    argument2: AnotherArgumentType,
): ReturnType {
    // body
}
```

Use regular indent (four spaces) for function parameters. It helps ensure consistency with constructor parameters.

Prefer using an expression body for functions with the body consisting of a single expression.

```
fun foo(): Int {     // bad
    return 1
}

fun foo() = 1        // good
```

Expression bodies

If the function has an expression body whose first line doesn't fit on the same line as the declaration, put the `=` sign on the first line and indent the expression body by four spaces.

```
fun f(x: String, y: String, z: String) =
    veryLongFunctionCallWithManyWords(andLongParametersToo(), x, y, z)
```

Properties

For very simple read-only properties, consider one-line formatting:

```
val isEmpty: Boolean get() = size == 0
```

For more complex properties, always put `get` and `set` keywords on separate lines:

```
val foo: String
    get() { /*...*/ }
```

For properties with an initializer, if the initializer is long, add a line break after the `=` sign and indent the initializer by four spaces:

```
private val defaultCharset: Charset? =
    EncodingRegistry.getInstance().getDefaultCharsetForPropertiesFiles(file)
```

Control flow statements

If the condition of an `if` or `when` statement is multiline, always use curly braces around the body of the statement. Indent each subsequent line of the condition by four spaces relative to the statement start. Put the closing parentheses of the condition together with the opening curly brace on a separate line:

```
if (!component.isSyncing &&
    !hasAnyKotlinRuntimeInScope(module)
) {
    return createKotlinNotConfiguredPanel(module)
}
```

This helps align the condition and statement bodies.

Put the `else`, `catch`, `finally` keywords, as well as the `while` keyword of a `do-while` loop, on the same line as the preceding curly brace:

```
if (condition) {
    // body
} else {
    // else part
}

try {
    // body
} finally {
    // cleanup
}
```

In a `when` statement, if a branch is more than a single line, consider separating it from adjacent case blocks with a blank line:

```
private fun parsePropertyValue(propName: String, token: Token) {
    when (token) {
        is Token.ValueToken ->
            callback.visitValue(propName, token.value)

        Token.LBRACE -> { // ...
        }
    }
}
```

Put short branches on the same line as the condition, without braces.

```
when (foo) {
    true -> bar() // good
    false -> { baz() } // bad
}
```

Method calls

In long argument lists, put a line break after the opening parenthesis. Indent arguments by four spaces. Group multiple closely related arguments on the same line.

```
drawSquare(
    x = 10, y = 10,
    width = 100, height = 100,
    fill = true
)
```

Put spaces around the `=` sign separating the argument name and value.

Wrap chained calls

When wrapping chained calls, put the `.` character or the `?.` operator on the next line, with a single indent:

```
val anchor = owner
    ?.firstChild!!
    .siblings(forward = true)
    .dropWhile { it is PsiComment || it is PsiWhiteSpace }
```

The first call in the chain should usually have a line break before it, but it's OK to omit it if the code makes more sense that way.

Lambdas

In lambda expressions, spaces should be used around the curly braces, as well as around the arrow which separates the parameters from the body. If a call takes a single lambda, pass it outside parentheses whenever possible.

```
list.filter { it > 10 }
```

If assigning a label for a lambda, do not put a space between the label and the opening curly brace:

```
fun foo() {
    ints.forEach lit@{
        // ...
    }
}
```

When declaring parameter names in a multiline lambda, put the names on the first line, followed by the arrow and the newline:

```
appendCommaSeparated(properties) { prop ->
    val propertyValue = prop.get(obj)  // ...
}
```

If the parameter list is too long to fit on a line, put the arrow on a separate line:

```
foo {
    context: Context,
    environment: Env
    ->
    context.configureEnv(environment)
}
```

Trailing commas

A trailing comma is a comma symbol after the last item in a series of elements:

```
class Person(
    val firstName: String,
    val lastName: String,
    val age: Int, // trailing comma
)
```

Using trailing commas has several benefits:

- It makes version-control diffs cleaner – as all the focus is on the changed value.
- It makes it easy to add and reorder elements – there is no need to add or delete the comma if you manipulate elements.

- It simplifies code generation, for example, for object initializers. The last element can also have a comma.

Trailing commas are entirely optional – your code will still work without them. The Kotlin style guide encourages the use of trailing commas at the declaration site and leaves it at your discretion for the call site.

To enable trailing commas in the IntelliJ IDEA formatter, go to **Settings/Preferences | Editor | Code Style | Kotlin**, open the **Other** tab and select the **Use trailing comma** option.

Enumerations

```
enum class Direction {
    NORTH,
    SOUTH,
    WEST,
    EAST, // trailing comma
}
```

Value arguments

```
fun shift(x: Int, y: Int) { /*...*/ }
shift(
    25,
    20, // trailing comma
)
val colors = listOf(
    "red",
    "green",
    "blue", // trailing comma
)
```

Class properties and parameters

```
class Customer(
    val name: String,
    val lastName: String, // trailing comma
)
class Customer(
    val name: String,
    lastName: String, // trailing comma
)
```

Function value parameters

```
fun powerOf(
    number: Int,
    exponent: Int, // trailing comma
) { /*...*/ }
constructor(
    x: Comparable<Number>,
    y: Iterable<Number>, // trailing comma
) {}
fun print(
    vararg quantity: Int,
    description: String, // trailing comma
) {}
```

Parameters with optional type (including setters)

```
val sum: (Int, Int, Int) -> Int = fun(
    x,
    y,
    z, // trailing comma
): Int {
    return x + y + x
}
println(sum(8, 8, 8))
```

Indexing suffix

```
class Surface {
    operator fun get(x: Int, y: Int) = 2 * x + 4 * y - 10
}
fun getZValue(mySurface: Surface, xValue: Int, yValue: Int) =
    mySurface[
        xValue,
        yValue, // trailing comma
    ]
```

Parameters in lambdas

```
fun main() {
    val x = {
            x: Comparable<Number>,
            y: Iterable<Number>, // trailing comma
        ->
        println("1")
    }
    println(x)
}
```

when entry

```
fun isReferenceApplicable(myReference: KClass<*>) = when (myReference) {
    Comparable::class,
    Iterable::class,
    String::class, // trailing comma
        -> true
    else -> false
}
```

Collection literals (in annotations)

```
annotation class ApplicableFor(val services: Array<String>)
@ApplicableFor([
    "serializer",
    "balancer",
    "database",
    "inMemoryCache", // trailing comma
])
fun run() {}
```

Type arguments

```
fun <T1, T2> foo() {}
fun main() {
    foo<
            Comparable<Number>,
            Iterable<Number>, // trailing comma
            >()
}
```

Type parameters

```
class MyMap<
        MyKey,
        MyValue, // trailing comma
        > {}
```

Destructuring declarations

```
data class Car(val manufacturer: String, val model: String, val year: Int)
val myCar = Car("Tesla", "Y", 2019)
val (
    manufacturer,
    model,
    year, // trailing comma
) = myCar
val cars = listOf<Car>()
fun printMeanValue() {
    var meanValue: Int = 0
    for ((
        _,
        _,
        year, // trailing comma
    ) in cars) {
        meanValue += year
    }
    println(meanValue/cars.size)
}
printMeanValue()
```

Documentation comments

For longer documentation comments, place the opening `/**` on a separate line and begin each subsequent line with an asterisk:

```
/**
 * This is a documentation comment
 * on multiple lines.
 */
```

Short comments can be placed on a single line:

```
/** This is a short documentation comment. */
```

Generally, avoid using `@param` and `@return` tags. Instead, incorporate the description of parameters and return values directly into the documentation comment, and add links to parameters wherever they are mentioned. Use `@param` and `@return` only when a lengthy description is required which doesn't fit into the flow of the main text.

```
// Avoid doing this:

/**
 * Returns the absolute value of the given number.
 * @param number The number to return the absolute value for.
 * @return The absolute value.
 */
fun abs(number: Int): Int { /*...*/ }

// Do this instead:

/**
 * Returns the absolute value of the given [number].
 */
fun abs(number: Int): Int { /*...*/ }
```

Avoid redundant constructs

In general, if a certain syntactic construction in Kotlin is optional and highlighted by the IDE as redundant, you should omit it in your code. Do not leave unnecessary syntactic elements in code just "for clarity".

Unit return type

If a function returns Unit, the return type should be omitted:

```
fun foo() { // ": Unit" is omitted here

}
```

Semicolons

Omit semicolons whenever possible.

String templates

Don't use curly braces when inserting a simple variable into a string template. Use curly braces only for longer expressions:

```
println("$name has ${children.size} children")
```

Use multi-dollar string interpolation to treat the dollar sign chars `$` as string literals:

```
val KClass<*>.jsonSchema : String
    get() = $$"""
        {
            "$schema": "https://json-schema.org/draft/2020-12/schema",
            "$id": "https://example.com/product.schema.json",
            "$dynamicAnchor": "meta",
            "title": "$${simpleName ?: qualifiedName ?: "unknown"}",
            "type": "object"
        }
        """
```

Idiomatic use of language features

Immutability

Prefer using immutable data to mutable. Always declare local variables and properties as `val` rather than `var` if they are not modified after initialization.

Always use immutable collection interfaces (`Collection` , `List` , `Set` , `Map`) to declare collections which are not mutated. When using factory functions to create collection instances, always use functions that return immutable collection types when possible:

```
// Bad: use of a mutable collection type for value which will not be mutated
fun validateValue(actualValue: String, allowedValues: HashSet<String>) { ... }

// Good: immutable collection type used instead
fun validateValue(actualValue: String, allowedValues: Set<String>) { ... }

// Bad: arrayListOf() returns ArrayList<T>, which is a mutable collection type
val allowedValues = arrayListOf("a", "b", "c")

// Good: listOf() returns List<T>
val allowedValues = listOf("a", "b", "c")
```

Default parameter values

Prefer declaring functions with default parameter values to declaring overloaded functions.

```
// Bad
fun foo() = foo("a")
fun foo(a: String) { /*...*/ }

// Good
fun foo(a: String = "a") { /*...*/ }
```

Type aliases

If you have a functional type or a type with type parameters which is used multiple times in a codebase, prefer defining a type alias for it:

```
typealias MouseClickHandler = (Any, MouseEvent) -> Unit
typealias PersonIndex = Map<String, Person>
```

If you use a private or internal type alias for avoiding name collision, prefer the `import ... as ...` mentioned in Packages and Imports.

Lambda parameters

In lambdas which are short and not nested, it's recommended to use the `it` convention instead of declaring the parameter explicitly. In nested lambdas with parameters, always declare parameters explicitly.

Returns in a lambda

Avoid using multiple labeled returns in a lambda. Consider restructuring the lambda so that it will have a single exit point. If that's not possible or not clear enough, consider converting the lambda into an anonymous function.

Do not use a labeled return for the last statement in a lambda.

Named arguments

Use the named argument syntax when a method takes multiple parameters of the same primitive type, or for parameters of `Boolean` type, unless the meaning of all parameters is absolutely clear from context.

```
drawSquare(x = 10, y = 10, width = 100, height = 100, fill = true)
```

Conditional statements

Prefer using the expression form of `try`, `if`, and `when`.

```
return if (x) foo() else bar()
```

```
return when(x) {
    0 -> "zero"
    else -> "nonzero"
}
```

The above is preferable to:

```
if (x)
    return foo()
else
    return bar()
```

```
when(x) {
    0 -> return "zero"
    else -> return "nonzero"
}
```

if versus when

Prefer using `if` for binary conditions instead of `when`. For example, use this syntax with `if`:

```
if (x == null) ... else ...
```

Instead of this one with `when`:

```
when (x) {
    null -> // ...
    else -> // ...
}
```

Prefer using `when` if there are three or more options.

Guard conditions in when expression

Use parentheses when combining multiple boolean expressions in `when` expressions or statements with guard conditions:

```
when (status) {
    is Status.Ok if (status.info.isEmpty() || status.info.id == null) -> "no information"
}
```

Instead of:

```
when (status) {
    is Status.Ok if status.info.isEmpty() || status.info.id == null -> "no information"
}
```

Nullable Boolean values in conditions

If you need to use a nullable `Boolean` in a conditional statement, use `if (value == true)` or `if (value == false)` checks.

Loops

Prefer using higher-order functions (`filter` , `map` etc.) to loops. Exception: `forEach` (prefer using a regular `for` loop instead, unless the receiver of `forEach` is nullable or `forEach` is used as part of a longer call chain).

When making a choice between a complex expression using multiple higher-order functions and a loop, understand the cost of the operations being performed in each case and keep performance considerations in mind.

Loops on ranges

Use the `..<` operator to loop over an open-ended range:

```
for (i in 0..n - 1) { /*...*/ }  // bad
for (i in 0..<n) { /*...*/ }  // good
```

Strings

Prefer string templates to string concatenation.

Prefer multiline strings to embedding `\n` escape sequences into regular string literals.

To maintain indentation in multiline strings, use `trimIndent` when the resulting string does not require any internal indentation, or `trimMargin` when internal indentation is required:

```
fun main() {
    println("""
     Not
     trimmed
     text
     """
    )

    println("""
     Trimmed
     text
     """.trimIndent()
    )

    println()

    val a = """Trimmed to margin text:
            |if(a > 1) {
            |    return a
            |}""".trimMargin()

   println(a)
}
```

Learn the difference between Java and Kotlin multiline strings.

Functions vs properties

In some scenarios, functions with no arguments might be interchangeable with read-only properties. Although the semantics are similar, there are some stylistic conventions on when to prefer one to another.

Prefer a property over a function when the underlying algorithm:

- Does not throw.
- Is cheap to calculate (or cached on the first run).
- Returns the same result over invocations if the object state hasn't changed.

Extension functions

Use extension functions liberally. Every time you have a function that works primarily on an object, consider making it an extension function accepting that object as a receiver. To minimize API pollution, restrict the visibility of extension functions as much as it makes sense. As necessary, use local extension functions, member extension functions, or top-level extension functions with private visibility.

Infix functions

Declare a function as `infix` only when it works on two objects which play a similar role. Good examples: `and`, `to`, `zip`. Bad example: `add`.

Do not declare a method as `infix` if it mutates the receiver object.

Factory functions

If you declare a factory function for a class, avoid giving it the same name as the class itself. Prefer using a distinct name, making it clear why the behavior of the factory function is special. Only if there is really no special semantics, you can use the same name as the class.

```
class Point(val x: Double, val y: Double) {
    companion object {
        fun fromPolar(angle: Double, radius: Double) = Point(...)
    }
}
```

If you have an object with multiple overloaded constructors that don't call different superclass constructors and can't be reduced to a single constructor including parameters with default values, prefer to replace the overloaded constructors with factory functions.

Platform types

A public function/method returning an expression of a platform type must declare its Kotlin type explicitly:

```
fun apiCall(): String = MyJavaApi.getProperty("name")
```

Any property (package-level or class-level) initialized with an expression of a platform type must declare its Kotlin type explicitly:

```
class Person {
    val name: String = MyJavaApi.getProperty("name")
}
```

A local value initialized with an expression of a platform type may or may not have a type declaration:

```
fun main() {
    val name = MyJavaApi.getProperty("name")
    println(name)
}
```

Scope functions apply/with/run/also/let

Kotlin provides a set of functions to execute a block of code in the context of a given object: `let`, `run`, `with`, `apply`, and `also`. For the guidance on choosing the right scope function for your case, refer to Scope Functions.

Coding conventions for libraries

When writing libraries, it's recommended to follow an additional set of rules to ensure API stability:

- Always explicitly specify member visibility (to avoid accidentally exposing declarations as public API).
- Always explicitly specify function return types and property types (to avoid accidentally changing the return type when the implementation changes).
- Provide KDoc comments for all public members, except for overrides that do not require any new documentation (to support generating documentation for the library).

Learn more about best practices and ideas to consider when writing an API for your library in the Library authors' guidelines.

Idioms

A collection of random and frequently used idioms in Kotlin. If you have a favorite idiom, contribute it by sending a pull request.

Create DTOs (POJOs/POCOs)

```
data class Customer(val name: String, val email: String)
```

provides a `Customer` class with the following functionality:

- getters (and setters in case of `var`s) for all properties
- `equals()`
- `hashCode()`
- `toString()`
- `copy()`
- `component1()`, `component2()`, ..., for all properties (see Data classes)

Default values for function parameters

```
fun foo(a: Int = 0, b: String = "") { ... }
```

Filter a list

```
val positives = list.filter { x -> x > 0 }
```

Or alternatively, even shorter:

```
val positives = list.filter { it > 0 }
```

Learn the difference between Java and Kotlin filtering.

Check the presence of an element in a collection

```
if ("john@example.com" in emailsList) { ... }

if ("jane@example.com" !in emailsList) { ... }
```

String interpolation

```
println("Name $name")
```

Learn the difference between Java and Kotlin string concatenation.

Read standard input safely

```
// Reads a string and returns null if the input can't be converted into an integer. For example: Hi
there!
val wrongInt = readln().toIntOrNull()
println(wrongInt)
// null

// Reads a string that can be converted into an integer and returns an integer. For example: 13
val correctInt = readln().toIntOrNull()
println(correctInt)
// 13
```

For more information, see Read standard input.

Instance checks

```
when (x) {
    is Foo → ...
    is Bar → ...
    else   → ...
}
```

Read-only list

```
val list = listOf("a", "b", "c")
```

Read-only map

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

Access a map entry

```
println(map["key"])
map["key"] = value
```

Traverse a map or a list of pairs

```
for ((k, v) in map) {
    println("$k → $v")
}
```

`k` and `v` can be any convenient names, such as `name` and `age`.

Iterate over a range

```
for (i in 1..100) { ... }  // closed-ended range: includes 100
for (i in 1..<100) { ... } // open-ended range: does not include 100
for (x in 2..10 step 2) { ... }
for (x in 10 downTo 1) { ... }
(1..10).forEach { ... }
```

Lazy property

```
val p: String by lazy { // the value is computed only on first access
    // compute the string
}
```

Extension functions

```
fun String.spaceToCamelCase() { ... }

"Convert this to camelcase".spaceToCamelCase()
```

Create a singleton

```
object Resource {
    val name = "Name"
}
```

Use inline value classes for type-safe values

```
@JvmInline
value class EmployeeId(private val id: String)

@JvmInline
value class CustomerId(private val id: String)
```

If you accidentally mix up `EmployeeId` and `CustomerId`, a compilation error is triggered.



The `@JvmInline` annotation is only needed for JVM backends.

Instantiate an abstract class

```
abstract class MyAbstractClass {
    abstract fun doSomething()
    abstract fun sleep()
}

fun main() {
    val myObject = object : MyAbstractClass() {
        override fun doSomething() {
            // ...
        }

        override fun sleep() { // ...
        }
    }
    myObject.doSomething()
}
```

If-not-null shorthand

```
val files = File("Test").listFiles()

println(files?.size) // size is printed if files is not null
```

If-not-null-else shorthand

```
val files = File("Test").listFiles()

// For simple fallback values:
println(files?.size ?: "empty") // if files is null, this prints "empty"

// To calculate a more complicated fallback value in a code block, use `run`
val filesSize = files?.size ?: run {
    val someSize = getSomeSize()
    someSize * 2
}
println(filesSize)
```

Execute an expression if null

```
val values = ...
val email = values["email"] ?: throw IllegalStateException("Email is missing!")
```

Get first item of a possibly empty collection

```
val emails = ... // might be empty
val mainEmail = emails.firstOrNull() ?: ""
```

Learn the difference between Java and Kotlin first item getting.

Execute if not null

```
val value = ...

value?.let {
    ... // execute this block if not null
}
```

Map nullable value if not null

```
val value = ...

val mapped = value?.let { transformValue(it) } ?: defaultValue
// defaultValue is returned if the value or the transform result is null.
```

Return on when statement

```
fun transform(color: String): Int {
    return when (color) {
        "Red" -> 0
        "Green" -> 1
        "Blue" -> 2
        else -> throw IllegalArgumentException("Invalid color param value")
    }
}
```

try-catch expression

```
fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
        throw IllegalStateException(e)
    }

    // Working with result
}
```

if expression

```
val y = if (x == 1) {
    "one"
} else if (x == 2) {
    "two"
} else {
    "other"
}
```

Builder-style usage of methods that return Unit

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

Single-expression functions

```
fun theAnswer() = 42
```

This is equivalent to

```
fun theAnswer(): Int {
    return 42
}
```

This can be effectively combined with other idioms, leading to shorter code. For example, with the `when` expression:

```
fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param value")
}
```

Call multiple methods on an object instance (with)

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
    fun forward(pixels: Double)
}

val myTurtle = Turtle()
with(myTurtle) { //draw a 100 pix square
    penDown()
    for (i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

Configure properties of an object (apply)

```
val myRectangle = Rectangle().apply {
    length = 4
    breadth = 5
    color = 0xFAFAFA
}
```

This is useful for configuring properties that aren't present in the object constructor.

Java 7's try-with-resources

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader →
    println(reader.readText())
}
```

Generic function that requires the generic type information

```
//  public final class Gson {
//      ...
//      public <T> T fromJson(JsonElement json, Class<T> classOfT) throws JsonSyntaxException {
//      ...

inline fun <reified T: Any> Gson.fromJson(json: JsonElement): T = this.fromJson(json, T::class.java)
```

Swap two variables

```
var a = 1
var b = 2
a = b.also { b = a }
```

Mark code as incomplete (TODO)

Kotlin's standard library has a `TODO()` function that will always throw a `NotImplementedError`. Its return type is `Nothing` so it can be used regardless of expected type. There's also an overload that accepts a reason parameter:

```
fun calcTaxes(): BigDecimal = TODO("Waiting for feedback from accounting")
```

IntelliJ IDEA's kotlin plugin understands the semantics of `TODO()` and automatically adds a code pointer in the TODO tool window.

What's next?

- Solve Advent of Code puzzles using the idiomatic Kotlin style.
- Learn how to perform typical tasks with strings in Java and Kotlin.
- Learn how to perform typical tasks with collections in Java and Kotlin.
- Learn how to handle nullability in Java and Kotlin.

Types overview

In Kotlin, everything is an object in the sense that you can call member functions and properties on any variable. While certain types have an optimized internal representation as primitive values at runtime (such as numbers, characters, and booleans), they appear and behave like regular classes to you.

This section describes the basic types used in Kotlin:

- Numbers and their unsigned counterparts
- Booleans
- Characters
- Strings
- Arrays

To learn about other Kotlin types, such as `Nothing`, `Any`, and `Unit`, look through the Kotlin API reference:

- `Any`
- `Nothing`
- `Unit`

Kotlin also has non-denotable types. They are the types that you can't write directly in the Kotlin code. Instead, the compiler uses them internally, for example, for interoperability with other languages. Kotlin creates non-denotable types to represent type information that is more precise than what Kotlin source syntax allows.

Even though you can't declare non-denotable types yourself, you may encounter them in compiler diagnostics, IDE tooltips, or inferred type displays. Learn more about non-denotable types in:

- Platform types
- Intersection types
- Integer literal types
- Captured types
- Kotlin language specification: Type system

Learn how to perform type checks and casts in Kotlin.

Numbers

The Kotlin number types represent:

- Integer values (Byte, Short, Int, and Long)
- Floating-point values (Float and Double)

Use number types to store and process numeric data, for example, in arithmetic, counters, measurements, and other calculations.

Choose a number type

In most cases, you can refer to the following rules to determine the correct number type for your task:

- Use `Int` for whole numbers.
- Use `Long` for whole numbers outside the `Int` range.
- Use `Double` for decimal numbers.
- Use `Float` when lower precision is acceptable or required.
- Use `Byte` and `Short` when an API or data format requires them.

Kotlin also provides Unsigned integer types as a Beta feature.

Integer types

Kotlin provides four integer types with different sizes and value ranges:

Type	Size (bits)	Min value	Max value
`Byte`	8	-128	127
`Short`	16	-32768	32767
`Int`	32	-2,147,483,648 (-231)	2,147,483,647 (231 - 1)
`Long`	64	-9,223,372,036,854,775,808 (-263)	9,223,372,036,854,775,807 (263 - 1)

Declare integer values

Kotlin supports the following literal forms for integer values:

- Decimals: `123`
- Hexadecimals: `0x0F`
- Binaries: `0b00001011`

Kotlin does not support octal literals.

To declare a numeric value, specify the type explicitly:

```
val one: Int = 1

// Use underscores to improve readability
val oneBillion: Long = 1_000_000_000
val hexBytes: Int = 0x7F_EC_DE_5E
val bytes: Int = 0b01010010_01101001_10010100_10010010

val oneByte: Byte = 1
val oneShort: Short = 1
```

You can also append the `L` suffix, to declare a `Long` value:

```
val oneLong = 1L
```

When you declare a numeric type explicitly, the compiler checks that the value fits in the range of that type:

```
// Value fits in Byte
val oneByte: Byte = 1

// Error: the value does not fit in Byte
val tooBig: Byte = 128
```

When you do not specify a numeric type, Kotlin infers `Int` if the value fits in the `Int` range. Otherwise, Kotlin infers `Long`:

```
val million = 1_000_000 // Int
val threeBillion = 3_000_000_000 // Long
```

If a value can be absent, use nullable types:

```
val maybeAbsent: Int? = null
```

Floating-point types

For numbers with a fractional part, Kotlin provides `Float` and `Double`.

Floating-point types follow the IEEE 754 standard. `Float` reflects the *single precision*. `Double` reflects the *double precision*.

Floating-point types differ in size and precision:

Type	Size (bits)	Significant bits	Exponent bits	Decimal digits
`Float`	32	24	8	6-7
`Double`	64	53	11	15-16

Declare floating-point values

To declare a floating-point literal, include a decimal point (`.`) or use exponent notation:

```
val pi = 3.14
val avogadro = 6.02214076e23
```

By default, Kotlin infers floating-point literals as `Double`. To declare a `Float`, add the `f` or `F` suffix:

```
val pi = 3.14 // Double
val eFloat = 2.7182817f // Float
```

Kotlin rounds a `Float` literal that contains more precision than `Float` can store.

If a value can be absent, use nullable types:

```
val maybeAbsent: Double? = null
```

Arithmetic operations

Kotlin supports the standard arithmetic operations on numbers: `+`, `-`, `*`, `/`, and `%`.

Use these operators to perform common calculations:

```
fun main() {
    println(1 + 2) // 3
    println(2_500_000_000L - 1L) // 2499999999
    println(3.14 * 2.71) // 8.5094
    println(10.0 / 3) // 3.3333333333333335
}
```

The result type depends on the types of the operands. Learn more in Mixed numeric expressions.

> You can override these operators in custom number classes. For more information, see Operator overloading.

Integer division

Division between integer values always returns an integer result. The compiler discards the fractional part:

```
fun main() {
    val intValue = 5 / 2
    println(intValue) // 2

    val longValue = 5L / 2
    println(longValue) // 2
}
```

To return a floating-point result, make at least one operand a `Float` or `Double`:

```
fun main() {
    val a = 5 / 2.0
    println(a) // 2.5

    val b = 5 / 2.toDouble()
    println(b) // 2.5
}
```

Type conversion

Numeric types are not subtypes of one another. Kotlin requires explicit conversions to avoid silent data loss and unexpected behavior.

For example, a function that expects `Double` cannot accept an `Int` or a `Float` value without conversion:

```
fun main() {
    fun printDouble(x: Double) {
        print(x)
    }

    val x = 1.0
    val xInt = 1
    val xFloat = 1.0f
    val one: Double = 1 // Error: initializer type mismatch

    printDouble(x) // OK
    printDouble(xInt) // Error: argument type mismatch
    printDouble(xFloat) // Error: argument type mismatch
}
```

All number types support conversions to other number types. To convert a number to another type, use an explicit conversion function:

- `toByte()`
- `toShort()`
- `toInt()`
- `toLong()`
- `toFloat()`
- `toDouble()`

For example, the following code converts an `Int` value to `Double`:

```
fun main() {
    val intValue: Int = 1
    val doubleValue = intValue.toDouble()

    println(doubleValue) // 1.0
}
```

When you convert a floating-point value to an integer type, the compiler discards the fractional part:

```
fun main() {
    val d: Double = 1.5
    val l: Long = d.toLong()

    println(l) // 1
}
```

Mixed numeric expressions

Kotlin does not support implicit conversion for assignments or function arguments. However, you can combine different numeric types in arithmetic expressions. In such cases, Kotlin determines a result type based on the operand types, and arithmetic operators handle the conversion automatically:

```
val intNumber: Int = 1
val longNumber: Long = 1000
val result = intNumber + longNumber // 1001, Long
```

If you try to assign the result to a smaller type, the compiler reports an error:

```
val intNumber: Int = 1
val longNumber: Long = 1000
val result: Int = intNumber + longNumber
// Error: Initializer type mismatch
```

Integer literal types

During type inference, Kotlin treats unsuffixed integer literals as a special Integer Literal Type (ILT) until the surrounding context determines a specific type:

```
fun List<Any>.log() {
    println(joinToString(" | ") { it::class.simpleName ?: "Unknown" })
}

fun main() {
    listOf(1, 2).log()
    // Int | Int

    listOf(1L, 2L).log()
    // Long | Long

    // Compiler interprets 1 as an ILT and resolves it to Long
    listOf(1, 2L).log()
    // Long | Long

    // .toInt() converts the literal to Int
    listOf(1.toInt(), 2L).log()
    // Int | Long
}
```

It's especially easy to miss with the `Int` and `Long` values because they have the same string representation at runtime. To avoid this, specify the expected type or convert values explicitly:

```
fun List<Any>.log() {
    println(joinToString(" | ") { it::class.simpleName ?: "Unknown" })
}

fun main() {
    val longValues: List<Long> = listOf(1, 2L)
    longValues.log()
    // Long | Long

    val numberValues: List<Number> = listOf(1.toInt(), 2L)
    numberValues.log()
    // Int | Long
}
```

You can also use an explicit type to catch unintended type inference:

```
fun main() {
    val intValues: List<Int> = listOf(1, 2L)
    // Error: initializer type mismatch
}
```

Learn more about Integer literal types.

Data overflow

Numeric types can represent only values within their defined ranges.

If the result of an operation falls outside that range, overflow occurs. If you convert a value to a smaller numeric type, the converted value may not preserve the original numeric value.

This behavior can affect the result of your code even when the compiler accepts it.

Overflow in operations

Each integer type can store only values within its defined range. When the result of an arithmetic operation exceeds that range, *data overflow* occurs:

```
fun main(){
    val intNumber: Int = 2147483647
    // Max Int value is 2147483647
    println(intNumber + 1) // -2147483648
}
```

Here, the result wraps around because the value no longer fits in `Int`.

The compiler does not automatically produce an error when integer overflow occurs.

Overflow in negation

Overflow can also occur during negation. For example, you cannot represent the positive counterpart of `Int.MIN_VALUE` as an `Int`.

```
fun main(){
    val min = Int.MIN_VALUE
    println(-min) // -2147483648
}
```

Narrowing conversions

When you convert a value to a smaller integer type, the result may not preserve the original numeric value:

```
fun main() {
    val large: Int = 130
    val narrowed: Byte = large.toByte()

    println(narrowed) // -126
}
```

However, since floating-point types follow the IEEE 754 Standard, very large results can become `Infinity`:

```
fun main() {
    println(Double.MAX_VALUE * 2) // Infinity
}
```

Bitwise operations

Kotlin provides *bitwise operations* for `Int` and `Long`. These operations are represented by a set of infix functions and `inv()`.

```
fun main() {
    val x = 1

    println(x shl 2) // 4
    println(x and 0x000FF000) // 0
}
```

Bitwise operations include:

- `shl()` – signed shift left
- `shr()` – signed shift right
- `ushr()` – unsigned shift right
- `and()` – bitwise AND
- `or()` – bitwise OR
- `xor()` – bitwise XOR
- `inv()` – bitwise inversion

Floating-point number comparison

In Kotlin, floating-point comparison depends on the static type of the operands.

When the operands are statically known to be `Float` or `Double`, operations on the numbers and the range that they form follow the IEEE 754 Standard for Floating-Point Arithmetic.

However, in generic use cases (such as `Any`, `Comparable<...>`, or `Collection<T>`), behavior differs for operands that are not statically typed as floating-point numbers. In these cases, Kotlin uses the `equals()` and `compareTo()` implementations for `Float` and `Double`.

As a result:

- `NaN` is considered equal to itself
- `NaN` is considered greater than any other element including `POSITIVE_INFINITY`
- `-0.0` is considered less than `0.0`

The following example shows the difference between operands statically typed as floating-point numbers and operands used through generic types:

```
//sampleStart
fun generalizedEquals(a: Any, b: Any): Boolean {
    return a == b
}

fun main() {
    // Operands statically typed as floating-point numbers
    println(Double.NaN == Double.NaN) // false
    println(0.0 == -0.0) // true

    // Operands used through a non-floating-point static type
    println(generalizedEquals(Double.NaN, Double.NaN)) // true
    println(generalizedEquals(0.0, -0.0)) // false
}
```

Boxing and caching numbers on the JVM

On the JVM, non-nullable numeric values are usually stored using primitive types, such as `int`, `long`, or `double`. However, when you use generic types or nullable numeric types like `Int?`, the value is boxed and represented as an object.

The JVM applies a memory optimization technique to small numbers by caching their boxed representations. As a result, boxed numbers with the same value can be referentially equal.

For example, the JVM caches boxed `Integer` values in the range `-128` to `127`. Therefore, the following code returns `true`:

```
fun main() {
    val score: Int = 100
    val savedScore: Int? = score
    val displayedScore: Int? = score

    println(savedScore === displayedScore) // true
}
```

For values outside the cached range, boxed values are separate objects. In that case, they are not referentially equal, even if their values are structurally equal. For this reason, use `==` to compare numeric values:

```
fun main() {
    val score: Int = 10000
    val savedScore: Int? = score
    val displayedScore: Int? = score

    println(savedScore === displayedScore) // false
    println(savedScore == displayedScore) // true
}
```

Unsigned integer types

In addition to integer types, Kotlin provides the following types for unsigned integer numbers:

Type	Size (bits)	Min value	Max value
`UByte`	8	0	255
`UShort`	16	0	65,535
`UInt`	32	0	4,294,967,295 (232 - 1)
`ULong`	64	0	18,446,744,073,709,551,615 (264 - 1)

Unsigned types support most of the operations of their signed counterparts.

Unsigned numbers are implemented as inline classes with a single storage property that contains the corresponding signed counterpart type of the same width. If you want to convert between unsigned and signed integer types, make sure you update your code so that any function calls and operations support the new type.

Unsigned arrays and ranges

Unsigned arrays and operations on them are in Beta. They can be changed incompatibly at any time. Opt-in is required (see the details below).

Same as for primitives, each unsigned type has a corresponding type that represents arrays of that type:

- `UByteArray`: an array of unsigned bytes.
- `UShortArray`: an array of unsigned shorts.
- `UIntArray`: an array of unsigned ints.
- `ULongArray`: an array of unsigned longs.

Same as for signed integer arrays, they provide a similar API to the `Array` class without boxing overhead.

When you use unsigned arrays, you receive a warning that indicates that this feature is not stable yet. To remove the warning, opt-in with the `@ExperimentalUnsignedTypes` annotation. It's up to you to decide if your clients have to explicitly opt-in into usage of your API, but keep in mind that unsigned arrays are not a stable feature, so an API that uses them can be broken by changes in the language. Learn more about opt-in requirements.

Ranges and progressions are supported for `UInt` and `ULong` by classes `UIntRange`, `UIntProgression`, `ULongRange`, and `ULongProgression`. Together with the unsigned integer types, these classes are stable.

Unsigned integers literals

To make unsigned integers easier to use, you can append a suffix to an integer literal indicating a specific unsigned type (similarly to `F` for `Float` or `L` for `Long`):

- `u` and `U` letters signify unsigned literals without specifying the exact type. If no expected type is provided, the compiler uses `UInt` or `ULong` depending on the size of the literal:

```
val b: UByte = 1u  // UByte, expected type provided
val s: UShort = 1u // UShort, expected type provided
val l: ULong = 1u  // ULong, expected type provided

val a1 = 42u // UInt: no expected type provided, constant fits in UInt
val a2 = 0xFFFF_FFFF_FFFFu // ULong: no expected type provided, constant doesn't fit in UInt
```

- `uL` and `UL` explicitly specify that literal should be an unsigned long:

```
val a = 1UL // ULong, even though no expected type provided and the constant fits into UInt
```

Use cases

The main use case of unsigned numbers is utilizing the full bit range of an integer to represent positive values. For example, to represent hexadecimal constants that do not fit in signed types such as color in 32-bit `AARRGGBB` format:

```
data class Color(val representation: UInt)

val yellow = Color(0xFFCC00CCu)
```

You can use unsigned numbers to initialize byte arrays without explicit `toByte()` literal casts:

```
val byteOrderMarkUtf8 = ubyteArrayOf(0xEFu, 0xBBu, 0xBFu)
```

Another use case is interoperability with native APIs. Kotlin allows representing native declarations that contain unsigned types in the signature. The mapping won't substitute unsigned integers with signed ones keeping the semantics unaltered.

Non-goals

While unsigned integers can only represent positive numbers and zero, it's not a goal to use them where application domain requires non-negative integers. For example, as a type of collection size or collection index value.

There are a couple of reasons:

- Using signed integers can help to detect accidental overflows and signal error conditions, such as `List.lastIndex` being -1 for an empty list.
- Unsigned integers cannot be treated as a range-limited version of signed ones because their range of values is not a subset of the signed integers range. Neither signed nor unsigned integers are subtypes of each other.

Booleans

The `Boolean` type represents logical values: `true` and `false`.

Use `Boolean` values in functions that answer yes-or-no questions, and in the `while`, `if`, and `when` conditions.

Declare a `Boolean` variable

To declare a `Boolean` variable, assign it `true` or `false`.

You can specify the `Boolean` type explicitly or let Kotlin infer it from the value:

```
val isTrue: Boolean = true
val isFalse = false // Kotlin infers Boolean
```

If a value can be `null`, use `Boolean?`:

```
val isEnabled: Boolean? = null
```


You cannot assign an integer value to a `Boolean` variable. In Kotlin, `0` and `1` are not `Boolean` values.

Produce `Boolean` values

You can use comparison expressions and functions to produce `Boolean` values:

```
fun main() {
    val number = 10
    val isPositive = number > 0
    println(isPositive) // true

    val language = "Kotlin"
    val isEmpty = language.isEmpty()
    println(isEmpty) // false
}
```

You can use the results in conditions and other expressions as well:

```
fun main() {
    val number = 10
    val isPositive = number > 0 // true

    if (isPositive) {
        println("The number is positive.")
    }
}
```

`Boolean` operations

Kotlin provides operators and infix functions for working with `Boolean` values. You can use them to invert a `Boolean` value or combine multiple `Boolean` values into a single result.

Negation (NOT)

The NOT operator inverts a `Boolean` value.

To use NOT, place the `!` operator before a `Boolean` value:

```
val isOn = true
val isOff = !isOn // isOff is false
```

Logical AND

The AND operator returns `true` only if both operands are `true`.

To use logical AND, place the `&&` operator between operands:

```
val a = false && false // false
val b = false && true // false
val c = true && false // false
val d = true && true  // true
```

If the first operand is `false`, the `&&` operator skips the second operand. To evaluate both operands, use the `and` infix function instead.

Logical OR

The OR operator returns `true` if at least one operand is `true`.

To use logical OR, place the `||` operator between operands:

```
val a = false || false // false
val b = false || true  // true
val c = true || false  // true
val d = true || true   // true
```

If the first operand is `true`, the `||` operator skips the second operand. To evaluate both operands, use the `or` infix function instead.

Exclusive OR (XOR)

The exclusive OR (XOR) operation returns `true` if the operands have different values.

To use XOR, write `xor` between operands:

```
val a = false xor false // false
val b = false xor true  // true
val c = true xor false  // true
val d = true xor true   // false
```

`xor` is an infix function, not an operator.

Learn more about `Boolean` functions in the API Reference.

Operator precedence

If an expression contains multiple logical operations and no parentheses to specify the evaluation order, Kotlin applies precedence rules. Operations with higher precedence are evaluated before operations with lower precedence.

For the `Boolean` operations described in this section, the precedence order is as follows:

1. `!`

2. `xor` (and other infix functions)
3. `&&`
4. `||`

In the following example, the compiler evaluates `&&` before `||`:

```
fun main() {
    val result = true || false && false
    println(result) // true
}
```

To make evaluation order explicit, use parentheses:

```
fun main() {
    val result = (true || false) && false
    println(result) // false
}
```

`Boolean` in conditions

`if`, `when`, and `while` evaluate `Boolean` expressions to direct program flow.

`if` expressions

```
fun main() {
    val number = 4
    val isEven = number % 2 == 0

    // Condition already has the `Boolean` type
    // You do not need to compare it to `true` or `false`
    if (isEven) {
        println("The number is even.")
    } else {
        println("The number is odd.")
    }
}
```

`when` expressions

```
fun main() {
    val number = 3

    when {
        number > 0 -> println("The number is positive.")
        number < 0 -> println("The number is negative.")
        else -> println("The number is zero.")
    }
}
```

`while` loops

```
fun main() {
    var isCalculating = true

    while (isCalculating) {
        println("Calculating...")
        isCalculating = false
    }
}
```

Characters

The `Char` type represents a single character as a UTF-16 code unit.

Use `Char` for individual character values, such as letters, digits, punctuation marks, or whitespace. For sequences of characters, use `String`.

> `Char` is not a numeric type, but each character has a numeric Unicode value that you can access. See Character conversion.

Syntax

To declare a character, enclose the value in single quotes (`' '`). You can specify the `Char` type explicitly or let Kotlin infer it from the value:

```
val letter: Char = 'a'

// Kotlin infers Char because the values are written in single quotes
val digit = '1'
val symbol = '!'
val space = ' '
val separator = ':'
```

A character literal must contain exactly one character. Otherwise, the Kotlin compiler reports an error:

```
val invalid = 'AB' // Error
val invalidEmpty = '' // Error
```

Nullable values

To store a nullable value, use `Char?`:

```
val maybeAbsent: Char? = null
```

> On the JVM, nullable `Char` values are boxed when needed. The same applies to numeric types.

Unicode support

Kotlin represents `Char` values as UTF-16 code units. This means that a single `Char` stores one UTF-16 code unit, not necessarily one complete Unicode character.

Basic Multilingual Plane

A single `Char` can store values in the range from `\u0000` to `\uFFFF`. This range covers the Basic Multilingual Plane (BMP) that includes characters for almost all modern languages and a large number of symbols.

To specify a character by the Unicode value, use `\u` followed by four-digit hexadecimal value from the Unicode table:

```
val unicodeNumber = '\u0031' // Equals '1'
```

Supplementary characters

Unicode characters outside the BMP, such as emojis and some historic scripts, cannot be represented by a single `Char`. In UTF-16, they are encoded as a *surrogate pair*, where two `Char` values together represent one Unicode character in a `String`:

```
fun main() {
    val emoji = "🥦"

    println(emoji.length) // 2
    println(emoji[0])     // First surrogate
    println(emoji[1])     // Second surrogate
}
```

To handle 32-bit symbols individually, use Unicode code points stored as `Int` values.

Escape sequences

Use escape sequences for special characters that are difficult to write directly in source code or have a special meaning.

Every escape sequence begins with a backslash (`\`).

Supported sequence	Description
`\t`	Tab
`\b`	Backspace
`\n`	New Line (LF)
`\r`	Carriage Return (CR)
`\'`	Single quotation mark

Supported sequence	Description
`\"`	Double quotation mark
`\\`	Backslash
`\$`	Dollar sign

For example:

```
val newLine = '\n'
val dollar = '\$'
val backslash = '\\'
```

Operations

`Char` supports comparison, inspection, case conversion, and explicit numeric conversion.

Character comparison

To compare `Char` values, use standard comparison operators such as `==`, `!=`, `<`, `>`, `<=`, and `>=`.

Kotlin compares `Char` values by their numeric Unicode values and returns a `Boolean` value:

```
val before = 'a' < 'b' // true
val after = 'c' > 'd' // false
val different = 'A' == 'a' // false
val equal = 'A' == 'A' // true
```

Character processing

Kotlin provides functions for inspection and case conversion of character values. For example:

```
fun main() {
    val myChar = 'A'
    // Checks if the character represents a digit
    println(myChar.isDigit()) // false
    // Checks if the character represents an uppercase letter
    println(myChar.isUpperCase()) // true
    // Returns a lowercase version
    println(myChar.lowercaseChar()) // 'a'
}
```

Learn more about available functions in the API Reference.

Character arithmetic

You can create another character value by adding or subtracting an integer:

```
fun main() {
    val a = 'a'

    println(a + 1)  // b
    println(a + 2)  // c
    println(a - 32) // A
}
```

These operations follow Unicode values, not language-specific alphabet rules.

You can also use the increment (`++`) and decrement (`--`) operators in the prefix and postfix forms with mutable variables:

```
fun main() {
    var a = 'A'

    a += 10
    println(a)   // 'K'

    println(++a) // 'L'  prefix increment
    println(a++) // 'L'  postfix increment
    println(a)   // 'M'

    println(--a) // 'L'  prefix decrement
    println(a--) // 'L'  postfix decrement
    println(a)   // 'K'
}
```

Character conversion

To convert `Char` to a numeric type, use explicit conversion:

- Use `.code` to get the numeric Unicode value of a character:

```
fun main() {
    val letter = 'A'
    println(letter.code) // 65
}
```

- If a character represents a decimal digit, use `digitToInt()` :

```
fun main() {
    val digit = '7'
    println(digit.digitToInt()) // 7
}
```

If the character may not be a valid digit, use `digitToIntOrNull()` .

Strings

The `String` type represents a sequence of characters. You can use it for text values, such as words, sentences, messages, or structured text.

The `String` type is immutable. After you create a `String` object, its contents stay the same for the rest of its lifetime. Any operation that appears to modify the string actually creates a new string.

Declare strings

To declare a `String` literal, enclose the value in double quotes (`""`). You can specify the `String` type explicitly or let Kotlin infer it from the value:

```
val name: String = "Kotlin"
val message = "Hello, world!" // Kotlin infers String
```

Double-quoted string literals support escape sequences such as `\n` or `\t` :

```
val message = "Hello,\nworld!"
val quote = "Kotlin says, \"Hi\"."
```

Multiline strings

To store text that consists of multiple lines or contains quotes that you don't want to escape, use a multiline string enclosed in triple quotes (`""" """`):

```
val text = """
Hello,
Kotlin
"""

val quote = """Kotlin says, "Hi"."""
```

Multiline strings don't support escape sequences. Kotlin treats these characters as regular text.

Multiline strings preserve line breaks and indentations as written in the source code. This behavior is useful when you want the runtime value to match the text layout in your file.

In the following example, the spaces before each line are part of the resulting string:

```
val text = """
    Hello,
    Kotlin
"""
```

To remove common leading indentation, use the `trimIndent()` function. It detects the common minimal indent of non-empty lines and removes it:

```
fun main() {
    val text = """
        Hello,
        Kotlin
    """.trimIndent()

    println(text)
}
```

To control indentation removal more explicitly, use the `trimMargin()` function. It removes everything before and including the margin prefix on each line:

```
fun main() {
    val text = """
        |Hello,
        |Kotlin
    """.trimMargin()

    println(text)
}
```

By default, the `trimMargin()` function uses a pipe symbol (`|`) as the margin prefix, but you can pass another character as a parameter. For example: `trimMargin(">")` .

When you process a string with functions like `trimIndent()` or `trimMargin()` , the resulting string uses only newline (`\n`) separators, regardless of the platform.

String templates

String templates let you embed variables and expressions directly inside a `String` literal. This process is called *interpolation*. You can use string templates in both regular and multiline strings.

To insert a variable into a string, use the `$` symbol:

```
fun main() {
    val name = "Kotlin"
    println("Hello, $name")
    // Hello, Kotlin
}
```

To insert an expression into a string or to place a variable directly next to other text, use `${}` :

```
fun main() {
    val text = "abc"
    println("The length of $text is ${text.length}")
    // The length of abc is 3

    val language = "Kotlin"
    println("${language}Lang")
    // KotlinLang
}
```

You can also combine strings with the `+` operator. However, string templates are usually easier to read and more idiomatic.

Template expressions can also contain double-quoted strings without escaping:

```
// Double-quoted string
val test = "${"test".uppercase()}"

// Multiline string
val result = """
Result: ${"OK".lowercase()}
"""
```

Nullable values in string templates

If an interpolated expression or variable evaluates to `null`, the Kotlin compiler inserts the text `null` into the resulting string. To replace `null` with another value, use the Elvis operator (`?:`):

```
fun main(){
    val text: String? = null

    println("Hello, $text")
    // Hello, null

    println("Hello, ${text ?: "Kotlin"}")
    // Hello, Kotlin
}
```

Multi-dollar string interpolation

In regular string templates, a single dollar sign (`$`) starts interpolation. If you need to include literal dollar signs in a string, use **multi-dollar string interpolation**.

Multi-dollar string interpolation allows you to specify how many consecutive dollar signs are required to trigger interpolation. Dollar signs below that number are treated as literal characters.

For example, when you use `$$` before a string literal, interpolation begins only with two consecutive dollar signs:

```
val KClass<*>.jsonSchema : String
    get() = $$"""
    {
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "$id": "https://example.com/product.schema.json",
      "$dynamicAnchor": "meta",
      "title": "$${simpleName ?: qualifiedName ?: "unknown"}",
      "type": "object"
    }
    """
```

If you use single-dollar string interpolation, multi-dollar string interpolation doesn't affect your code. You can continue using a single `$` and apply multi-dollar signs when needed.

Basic string operations

Kotlin provides a range of operations for working with strings. This section introduces some of the most commonly used operations.

Learn more about all available functions in the API Reference.

Get string length

To get the number of characters in a string, use the `length` property:

```
fun main (){
    val language = "Kotlin"
    println(language.length)
    // 6
}
```

Access characters

You can access an individual character in a string with the indexing operator (`[]`):

```
fun main (){
    val language = "Kotlin"

    println(language[0])
    // K
    println(language[5])
    // n
}
```

A string index starts at zero. If you try to access an index outside the valid range, Kotlin throws an exception.

You can also iterate over the characters in a string:

```
fun main(){
    for (char in "Kotlin") {
      println(char)
    }
}
```

Extract parts of a string

To extract parts of a string, use one of the following functions:

- `substring()` to return a new string with the selected part of the original text.
- `subSequence()` to return a `CharSequence` with the selected part of the original text.

For example:

```
fun main() {
//sampleStart
    val text = "Kotlin"
    println(text.substring(1))
    // otlin
    println(text.substring(1, 5))
    // otli
    println(text.subSequence(1, 5))
    // otli
}
```

Since the `String` type is immutable, these functions don't modify the original string.

Compare strings

You can check whether two strings have the same content with the `==` operator:

```
fun main(){
    println("kotlin" == "kotlin")
    // true

    println("kotlin" == "Kotlin")
    // false
}
```

You can also compare strings lexicographically (character by character) with the `compareTo()` function. It scans both strings until it finds the first differing pair of characters and returns:

- `0` when the strings are equal.
- A value less than `0` when the receiver is smaller than the argument.
- A value greater than `0` when the receiver is greater than the argument.

```
fun main() {
//sampleStart
    println("abc".compareTo("abd") < 0)
    // true

    println("abc".compareTo("ABC") > 0)
    // true

    // Pass true to ignore case differences
    println("abc".compareTo("ABC", ignoreCase = true) == 0)
    // true
}
```

Work with string content

If you want to change the content of a string, create a modified copy of it with functions like `.trim()`, `.replace()`, `.uppercase()`, and `.lowercase()`:

```
fun main() {
    val text = "  Hello, Kotlin  "

    println(text.trim())
    // Hello, Kotlin

    println(text.replace("Kotlin", "world"))
    //   Hello, world

    println(text.uppercase())
    //   HELLO, KOTLIN

    println(text.lowercase())
    //   hello, kotlin
}
```

You can also inspect the string content with the `contains()`, `startsWith()`, and `endsWith()` functions:

```
fun main() {
    val domain = "kotlinlang.org"

    // Checks if the string contains "."
    println(domain.contains("."))
    // true

    // Checks if the string starts with "kotlin"
    println(domain.startsWith("kotlin"))
    // true

    // Checks if the string ends with ".org"
    println(domain.endsWith(".org"))
    // true
}
```

Split strings

You can divide a string into parts around a delimiter with the `split()` function:

```
fun main() {
    val numbers = "one, two, three"
    println(numbers.split(", "))
    // [one, two, three]
}
```

If you want to split a string into individual lines, use the `lines()` function:

```
fun main() {
    val numbers = "one\ntwo\nthree"
    println(numbers.lines())
    // [one, two, three]
}
```

Build and format strings

For most formatting tasks in Kotlin, use string templates.

When you concatenate strings with the `+` operator, Kotlin creates a new `String` object for each operation. However, this approach may not be beneficial in loops or when you assemble many pieces. To avoid such issues, you can use the `buildString()` function or `StringBuilder`. They collect all pieces in a single mutable buffer and produce only one string at the end.

Use the `buildString()` function when the logic that determines what to append is complex. For example, when you have multiple conditions that contribute a different fragment. With `buildString()`, you don't handle the buffer directly. The function creates a `StringBuilder` internally, runs your block, and returns the resulting string.

```
fun main() {

    val hasErrors = true
    val hasWarnings = true
    val isComplete = false

    // buildString creates an empty buffer
    val status = buildString {
        // Appends "Errors found" to the buffer
        if (hasErrors) append("Errors found")
        if (hasWarnings) {
            // The buffer is not empty, appends "; "
            if (isNotEmpty()) append("; ")
            // Appends "Warnings found"
            append("Warnings found")
        }
        // isComplete = false, nothing to append
        if (isComplete) {
            if (isNotEmpty()) append("; ")
            append("Completed")
        }
        // The buffer is not empty, skips the fallback
        if (isEmpty()) append("OK")
    }

    println(status)
    // Errors found; Warnings found
}
```

Use `StringBuilder` when you need the buffer as an explicit value. For example, to change the existing text:

```
fun main() {
    val text = "Hello, Kotlin"
    val builder = StringBuilder(text)

    builder.replace(7, 13, "world")
    println(builder.toString())
    // Hello, world
}
```

On the JVM, you can also format a string with the `String.format()` function:

```
val text = String.format("Hello, %s", "Kotlin")
```

Use the `String.format()` function only when you specifically need formatter-style specifiers on the JVM. Learn more about format specifiers in the Java Class Formatter documentation.

String conversion

Often you may use strings to represent values of other types, such as numbers, `Boolean` values, or identifiers from the input. Kotlin provides functions for converting values to strings and for parsing strings into other types.

To return a string representation of a value, use the `toString()` function:

```
val number = 10
val text = number.toString()
```

In string templates and string concatenation, Kotlin converts values to strings automatically.

To convert a string to another type, use the corresponding parsing functions:

- For integer values: `toByte()`, `toShort()`, `toInt()`, `toLong()`
- For floating-point values: `toDouble()`, `toFloat()`
- For booleans: `toBoolean()`, `toBooleanStrict()`

These functions return a value of the requested type if the string has a valid format. If the input may be invalid, use the `OrNull` variants. These functions return `null` instead of throwing an exception making them safe for user input or data that you don't fully control:

```
val toInt = "10".toInt() // 10

// 1000000000000 exceeds maximum value of Int
val toIntInvalid = "1000000000000".toIntOrNull()

val toBoolean = "true".toBooleanStrict() // true
val toBooleanInvalid = "yes".toBooleanStrictOrNull() // null
```

Arrays

An array is a data structure that holds a fixed number of values of the same type or its subtypes. Array elements are ordered and accessed by index.

Kotlin provides the `Array<T>` class and primitive-type arrays.

When to use arrays

Use arrays for interoperability with Java APIs or low-level requirements. For example, if you have performance requirements beyond what is needed for regular applications, or you need to build custom data structures.

For most use cases, prefer collections instead.

Functionality	Arrays	Collections
Size	Fixed	Depends on the type
Read-only variant	No, always mutable	Yes (`List` and `Set`)
Adding and removing elements	No native support. Allocate and copy to a new array	Yes (mutable collections)
Structural equality with `==`	No, compares references. Use `.contentEquals()` instead	Yes
Primitive values	Primitive-type arrays store values without boxing	Usually boxed

Functionality	Arrays	Collections
Java interoperability	Maps to `T[]`	Maps to `java.util.List` and `java.util.Set`
Functional-style filtering and transformations	Limited	Extensive

Learn how to convert arrays to collections.

Create arrays

To create arrays, you can use:

- The `arrayOf()`, `arrayOfNulls()`, or `emptyArray()` functions.
- The `Array` constructor.

> Declaring an array with `val` only prevents reassigning the variable but doesn't make the contents read-only. Elements are mutable in both `val` and `var` arrays. The distinction only affects the reference, not the contents. For a read-only view, use collections instead.

Array with values

To create a typed array from a known set of values, use the `arrayOf()` function. Kotlin infers the type automatically:

```
fun main() {
    val simpleArray = arrayOf(1, 2, 3) // Array<Int>
    println(simpleArray.joinToString())
    // 1, 2, 3
}
```

Empty array

To create an array with no elements, use the `emptyArray()` function. You can specify the type of elements on the left-hand or right-hand side of the assignment:

```
val emptyArrayRight = emptyArray<String>()
val emptyArrayLeft: Array<String> = emptyArray()
```

Learn how to add elements to an array.

Array with nulls

To create an array of a given size filled with `null` elements, use the `arrayOfNulls()` function:

```
fun main() {
    val nullArray: Array<Int?> = arrayOfNulls(3)
    println(nullArray.joinToString())
```

```
    // null, null, null
}
```

Array constructor

The `Array` constructor takes the array size and a function that returns values of array elements:

```
fun main() {
    val zeroes = Array<Int>(3) { 0 }
    println(zeroes.joinToString())
    // 0, 0, 0

    val squares = Array(5) { i -> i * i }
    println(squares.joinToString())
    // 0, 1, 4, 9, 16
}
```

Nested arrays

To create a nested or multidimensional array, use an array of arrays. Nested arrays don't have to be the same type or the same size.

```
fun main() {
    // Creates a two-dimensional array
    val twoDArray = Array(2) { Array<Int>(2) { 0 } }
    println(twoDArray.contentDeepToString())
    // [[0, 0], [0, 0]]

    // Creates a three-dimensional array
    val threeDArray = Array(3) { Array(3) { Array<Int>(3) { 0 } } }
    println(threeDArray.contentDeepToString())
    // [[[0, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 0, 0], [0, 0, 0],
[0, 0, 0]]]
}
```

Primitive-type arrays

If you use the `Array` class with primitive values, the compiler boxes these values into objects. To avoid boxing overhead, you can use dedicated primitive-type arrays. They are not subclasses of the `Array<T>` class, but they provide a similar set of functions and properties.

Kotlin type	Java equivalent
`BooleanArray`	`boolean[]`
`ByteArray`	`byte[]`
`CharArray`	`char[]`
`DoubleArray`	`double[]`

Kotlin type	Java equivalent
`FloatArray`	`float[]`
`IntArray`	`int[]`
`LongArray`	`long[]`
`ShortArray`	`short[]`

Kotlin doesn't have a dedicated `StringArray` type. `String` is not a primitive, so use the `arrayOf()` or `arrayOf<String>()` functions with inferred type instead.

To create a primitive-type array, use one of the following options:

- Constructor functions:

```
fun main() {
    // Creates an Int array of size 5 with the values initialized to zero
    val primitiveTypeArray = IntArray(5)
    println(primitiveTypeArray.joinToString())
    // 0, 0, 0, 0, 0

    // Creates an Int array and takes an initializer function
    val squares = IntArray(5) { i -> i * i }
    println(squares.joinToString())
    // 0, 1, 4, 9, 16
}
```

- Factory functions:

```
fun main() {
    // Creates an Int array with 5 elements
    val numbers = intArrayOf(1, 2, 3, 4, 5)
    println(numbers.joinToString())
    // 1, 2, 3, 4, 5

    // Creates a Char array with 3 elements
    val characters = charArrayOf('K', 't', 'l')
    println(characters.joinToString())
    // K, t, l

    // Creates a Double array with 3 elements
    val doubles = doubleArrayOf(0.22, 4.16, 0.5)
    println(doubles.joinToString())
    // 0.22, 4.16, 0.5
}
```

To convert primitive-type arrays to object-type arrays, use the `.toTypedArray()` function.

To convert object-type arrays to primitive-type arrays, use `.toBooleanArray()`, `.toByteArray()`, `.toCharArray()`, and so on.

Work with arrays

Arrays support many of the same operations as collections, including iteration, searching, sorting, and transformations. In Kotlin, you can work with arrays by using them to pass a variable number of arguments to a function or perform operations on the arrays themselves. Find the most common properties and functions in the following table:

Member	Returns
`size`	The number of elements
`indices`	The range of valid indices
`lastIndex`	The last valid index
`first()` and `last()`	First and last element
`isEmpty()` and `isNotEmpty()`	`true` if array is empty or not empty
`contains()`	`true` if array has the element

Learn more about array properties and functions in the API reference.

This section introduces some of the most commonly used operations.

Access and modify elements

To access and modify elements in an array, use the indexed access operator (`[]`):

```
fun main() {
    val simpleArray = arrayOf(1, 2, 3)
    val twoDArray = Array(2) { Array<Int>(2) { 0 } }

    // Accesses the element and modifies it
    simpleArray[0] = 10
    twoDArray[0][0] = 2

    // Prints the modified element
    println(simpleArray[0])
    // 10
    println(twoDArray[0][0])
    // 2
}
```

If you try to access an index outside the bounds of an array, Kotlin throws `ArrayIndexOutOfBoundsException` at runtime.

You can also use the `fill(element, fromIndex, toIndex)` function to replace elements in a range in place. `fromIndex` is inclusive, and `toIndex` is exclusive:

```
fun main() {
    val arr = IntArray(3)
    println(arr.joinToString())
    // 0, 0, 0

   arr.fill(1)
   println(arr.joinToString())
   // 1, 1, 1

   arr.fill(0, 0, 2)
   println(arr.joinToString())
   // 0, 0, 1
}
```

In Kotlin, arrays are *invariant*. This means that `Array<String>` is not a subtype of `Array<Any>`. This prevents possible runtime type failures. To express covariance, use the `Array<out Any>` type projection:

```
fun main() {
    fun printArr(arr: Array<out Any>) {
        for (item in arr) print("$item, ")
    }

    printArr(arrayOf("k", "t", "n"))
    // k, t, n,
}
```

Add and remove elements

Since arrays have a fixed size, they don't support the `.add()` and `.remove()` functions. To perform these operations, you need to create a new array. For that, you can use one of the following options:

- Use the `.copyOf` function:

```
fun main() {
    var arr = intArrayOf(0, 1, 2)

    arr = arr.copyOf(arr.size + 1)
    println(arr.joinToString())
    // 0, 1, 2, 0

    arr[arr.lastIndex] = 3
    println(arr.joinToString())
    // 0, 1, 2, 3
}
```

- Use the `+` or `+=` operators:

```
fun main() {
    var arr = intArrayOf(0, 1, 2)

    arr += 3
    println(arr.joinToString())
    // 0, 1, 2, 3

    arr = arr + intArrayOf(4, 5)
    println(arr.joinToString())
    // 0, 1, 2, 3, 4, 5
}
```

If you need to frequently add or remove elements, use mutable collections instead.

Compare arrays

To compare whether two arrays have the same elements in the same order, use the `.contentEquals()` and `.contentDeepEquals()` functions:

```
fun main() {
    val simpleArray = arrayOf(1, 2, 3)
    val anotherArray = arrayOf(1, 2, 3)

    // Compares contents of arrays
    println(simpleArray.contentEquals(anotherArray))
    // true

    // Using infix notation, compares contents of arrays after an element
    // is changed
    simpleArray[0] = 10
    println(simpleArray contentEquals anotherArray)
    // false
}
```

Don't use equality (`==`) and inequality (`!=`) operators to compare the contents of arrays. These operators check whether the assigned variables point to the same object.

To learn more about why arrays in Kotlin behave this way, see our blog post.

Transform arrays

Kotlin has many useful functions to transform arrays. This section highlights some of them. See the complete list in our API reference.

Sum

To return the sum of all elements in an array, use the `.sum()` function:

```
fun main() {
    val sumArray = arrayOf(1, 2, 3)
    println(sumArray.sum())
    // 6
}
```

The `.sum()` function can only be used with arrays of numeric data types, such as `Int` .

Sort and shuffle

You can sort the elements in the array according to their natural order with the `.sort()` function or randomly shuffle them with the `.shuffle()` function:

```
fun main() {
    val simpleArray = arrayOf(1, 2, 3)

    // Randomly shuffles elements
    simpleArray.shuffle()
    println(simpleArray.joinToString())

    // Sorts elements
    simpleArray.sort()
    println(simpleArray.joinToString())
    // 1, 2, 3
}
```

To get a new sorted array without modifying the original, use the `.sortedArray()` function instead.

Pass variable number of arguments to a function

In Kotlin, you can pass a variable number of arguments to a function via the `vararg` parameter. This is useful when you don't know the number of arguments in advance, like when formatting a message or creating an SQL query.

To pass an array containing a variable number of arguments to a function, use the *spread operator* (`*`). The spread operator passes each element of the array as individual arguments to your chosen function:

```
fun main() {
    val lettersArray = arrayOf("c", "d")
    printAllStrings("a", "b", *lettersArray)
    // abcd
}

fun printAllStrings(vararg strings: String) {
    for (string in strings) {
        print(string)
    }
}
```

For more information, see Variable number of arguments (varargs).

Convert to collections

If you work with different APIs where some use arrays and some use collections, you can convert your arrays to collections and vice versa. For that, use the `.toList()` , `.toSet()` , and `.toMap()` functions. These functions copy the content from your array to the independent copy. They don't reflect subsequent changes to the array.

Convert to List or Set

To convert an array to a `List` or `Set` , use the `.toList()` and `.toSet()` functions:

```
fun main() {
    val simpleArray = arrayOf("a", "b", "c", "c")

    // Converts to a Set
    println(simpleArray.toSet())
    // [a, b, c]

    // Converts to a List
    println(simpleArray.toList())
    // [a, b, c, c]
}
```

Unless you are completely sure that the original array isn't changed or shared elsewhere, don't use `.asList()` and related `as*` functions. These functions wrap the original array instead of copying it. Therefore, changes to the array are reflected in the list and vice versa.

```
fun main() {
    val simpleArray = arrayOf("a", "b", "c")

    val list = simpleArray.asList()
    simpleArray[0] = "d"
    println(list)
    // [d, b, c]
}
```

Convert to Map

To convert an array to a `Map`, use the `.toMap()` function.

You can convert only an array of `Pair<K,V>` to a `Map`. The first value of a `Pair` instance becomes a key, and the second becomes a value. If the same key appears more than once, the last value is used.

This example uses the infix notation to call the `.to` function to create tuples of `Pair`:

```
fun main() {
    val pairArray = arrayOf("apple" to 120, "banana" to 150, "cherry" to 90, "apple" to 140)

    // Converts to a Map
    // Fruits are keys, calorie numbers are values
    // The latest "apple" value overwrites the first one
    println(pairArray.toMap())
    // {apple=140, banana=150, cherry=90}
}
```

What's next?

- Learn more about why we recommend using collections for most use cases in the Collections overview.
- Learn about other basic types.
- If you are a Java developer, read our Java to Kotlin migration guide for Collections.

Type checks and casts

In Kotlin, you can do two things with types at runtime: check whether an object is a specific type, or convert it to another type. Type **checks** help you confirm the kind of object you're dealing with, while type **casts** attempt to convert the object to another type.

> To learn specifically about **generics** type checks and casts, for example `List<T>`, `Map<K,V>`, see Generics type checks and casts.

Checks with `is` and `!is` operators

Use the `is` operator (or `!is` to negate it) to check if an object matches a type at runtime:

```
fun main() {
    val input: Any = "Hello, Kotlin"

    if (input is String) {
        println("Message length: ${input.length}")
        // Message length: 13
    }

    if (input !is String) { // Same as !(input is String)
        println("Input is not a valid message")
    } else {
        println("Processing message: ${input.length} characters")
        // Processing message: 13 characters
    }
}
```

You can also use `is` and `!is` operators to check if an object matches a subtype:

```
interface Animal {
    val name: String
    fun speak()
}

class Dog(override val name: String) : Animal {
    override fun speak() = println("$name says: Woof!")
}

class Cat(override val name: String) : Animal {
    override fun speak() = println("$name says: Meow!")
}
fun handleAnimal(animal: Animal) {
    println("Handling animal: ${animal.name}")
    animal.speak()

    // Use is operator to check for subtypes
    if (animal is Dog) {
        println("Special care instructions: This is a dog.")
    } else if (animal is Cat) {
        println("Special care instructions: This is a cat.")
    }
}
fun main() {
    val pets: List<Animal> = listOf(
        Dog("Buddy"),
        Cat("Whiskers"),
        Dog("Rex")
    )

    for (pet in pets) {
        handleAnimal(pet)
        println("---")
    }
    // Handling animal: Buddy
    // Buddy says: Woof!
    // Special care instructions: This is a dog.
    // ---
    // Handling animal: Whiskers
    // Whiskers says: Meow!
    // Special care instructions: This is a cat.
    // ---
    // Handling animal: Rex
    // Rex says: Woof!
    // Special care instructions: This is a dog.
    // ---
}
```

This example uses the `is` operator to check if the `Animal` class instance has subtype `Dog` or `Cat` to print the relevant care instructions.

You can check if an object is a supertype of its declared type, but it's not worthwhile because the answer is always true. Every class instance is already an instance of its supertypes.



To identify the type of an object at runtime, see Reflection.

Type casts

To convert the type of an object in Kotlin to another type is called **casting**.

In some cases, the compiler automatically casts objects for you. This is called smart-casting.

If you need to explicitly cast a type, use `as?` or `as` cast operators.

Smart casts

The compiler tracks the type checks and explicit casts for immutable values and inserts implicit (safe) casts automatically:

```
fun logMessage(data: Any) {
    // data is automatically cast to String
    if (data is String) {
        println("Received text: ${data.length} characters")
    }
}

fun main() {
    logMessage("Server started")
    // Received text: 14 characters
    logMessage(404)
}
```

The compiler is even smart enough to know that a cast is safe if a negative check leads to a return:

```
fun logMessage(data: Any) {
    // data is automatically cast to String
    if (data !is String) return

    println("Received text: ${data.length} characters")
}

fun main() {
    logMessage("User signed in")
    // Received text: 14 characters
    logMessage(true)
}
```

Control flow

Smart casts work not only for `if` conditional expressions, but also for `when` expressions:

```
fun processInput(data: Any) {
    when (data) {
        // data is automatically cast to Int
        is Int -> println("Log: Assigned new ID ${data + 1}")
        // data is automatically cast to String
        is String -> println("Log: Received message \"$data\"")
        // data is automatically cast to IntArray
        is IntArray -> println("Log: Processed scores, total = ${data.sum()}")
    }
}

fun main() {
    processInput(1001)
    // Log: Assigned new ID 1002
    processInput("System rebooted")
    // Log: Received message "System rebooted"
    processInput(intArrayOf(10, 20, 30))
    // Log: Processed scores, total = 60
}
```

And for `while` loops:

```
sealed interface Status
data class Ok(val currentRoom: String) : Status
data object Error : Status

class RobotVacuum(val rooms: List<String>) {
    var index = 0

    fun status(): Status =
        if (index < rooms.size) Ok(rooms[index])
        else Error

    fun clean(): Status {
        println("Finished cleaning ${rooms[index]}")
        index++
        return status()
    }
}

fun main() {
        val robo = RobotVacuum(listOf("Living Room", "Kitchen", "Hallway"))

    var status: Status = robo.status()
    while (status is Ok) {
        // The compiler smart casts status to OK type, so the currentRoom
        // property is accessible.
        println("Cleaning ${status.currentRoom}...")
        status = robo.clean()
    }
    // Cleaning Living Room...
    // Finished cleaning Living Room
    // Cleaning Kitchen...
    // Finished cleaning Kitchen
    // Cleaning Hallway...
    // Finished cleaning Hallway
    //sampleEnd
}
```

In this example, the sealed interface `Status` has two implementations: the data class `Ok` and the data object `Error`. Only the `Ok` data class has the `currentRoom` property. When the `while` loop condition evaluates to

true, the compiler smart casts the `status` variable to `Ok` type, making the `currentRoom` property accessible within the loop body.

If you declare a variable of `Boolean` type before using it in your `if`, `when`, or `while` condition, any information collected by the compiler about the variable is accessible in the corresponding block for smart-casting.

This can be useful when you want to do things like extract boolean conditions into variables. Then, you can give the variable a meaningful name, which improves your code readability and makes it possible to reuse the variable later in your code. For example:

```
class Cat {
    fun purr() {
        println("Purr purr")
    }
}
fun petAnimal(animal: Any) {
    val isCat = animal is Cat
    if (isCat) {
        // The compiler can access information about
        // isCat, so it knows that animal was smart-cast
        // to the type Cat.
        // Therefore, the purr() function can be called.
        animal.purr()
    }
}

fun main(){
    val kitty = Cat()
    petAnimal(kitty)
    // Purr purr
}
```

Logical operators

The compiler can perform smart casts on the right-hand side of `&&` or `||` operators if there is a type check (regular or negative) on the left-hand side:

```
// x is automatically cast to String on the right-hand side of `||`
if (x !is String || x.length == 0) return

// x is automatically cast to String on the right-hand side of `&&`
if (x is String && x.length > 0) {
    print(x.length) // x is automatically cast to String
}
```

If you combine type checks for objects with an `and` operator (`&&`), the compiler smart-casts the object to all checked types simultaneously. Learn more in Intersection types.

If you combine type checks for objects with an `or` operator (`||`), a smart cast is made to their closest common supertype:

```
interface Status {
    fun signal() {}
}

interface Ok : Status
interface Postponed : Status
interface Declined : Status

fun signalCheck(signalStatus: Any) {
    if (signalStatus is Postponed || signalStatus is Declined) {
        // signalStatus is smart-cast to a common supertype Status
        signalStatus.signal()
    }
}
```

The common supertype is an **approximation** of a union type. Union types are not currently supported in Kotlin.

Intersection types

When the compiler smart-casts an object through multiple `&&` checks, it infers an *intersection type*. This is an internal type that simultaneously satisfies all the checked constraints:

```
interface Bird {
    fun fly()
}

interface Fish {
    fun swim()
}

fun describe(animal: Any) {
    // Infers the Bird and Fish types
    if (animal is Bird && animal is Fish) {
        // Accesses fly() and swim() without additional checks or casts
        animal.fly()
        animal.swim()
    }
}
```

Intersection types are non-denotable. They only exist in the compiler's internal type system to preserve type information during type checking. You can't write them directly in Kotlin code. You may encounter intersection types in compiler error messages and IDE tooltips, typically displayed as `A & B`. The one exception is `T & Any` that declares a definitely non-nullable type. This syntax is reserved specifically to combine a type parameter with `Any`.

```
fun <T> T.assertNotNull(): T & Any = this ?: throw IllegalStateException("null value")
```

Inline functions

The compiler can smart-cast variables captured within lambda functions that are passed to inline functions.

Inline functions are treated as having an implicit `callsInPlace` contract. This means that any lambda functions passed to an inline function are called in place. Since lambda functions are called in place, the compiler knows that a lambda function can't leak references to any variables contained within its function body.

The compiler uses this knowledge, along with other analyses, to decide whether it's safe to smart-cast any of the captured variables. For example:

```
interface Processor {
    fun process()
}

inline fun inlineAction(f: () -> Unit) = f()

fun nextProcessor(): Processor? = null

fun runProcessor(): Processor? {
    var processor: Processor? = null
    inlineAction {
        // The compiler knows that processor is a local variable and inlineAction()
        // is an inline function, so references to processor can't be leaked.
        // Therefore, it's safe to smart-cast processor.

        // If processor isn't null, processor is smart-cast
        if (processor ≠ null) {
            // The compiler knows that processor isn't null, so no safe call
            // is needed
            processor.process()
        }

        processor = nextProcessor()
    }

    return processor
}
```

Exception handling

Smart cast information is passed on to `catch` and `finally` blocks. This makes your code safer as the compiler tracks whether your object has a nullable type. For example:

```
fun testString() {
    var stringInput: String? = null
    // stringInput is smart-cast to String type
    stringInput = ""
    try {
        // The compiler knows that stringInput isn't null
        println(stringInput.length)
        // 0

        // The compiler rejects previous smart cast information for
        // stringInput. Now stringInput has the String? type.
        stringInput = null

        // Trigger an exception
        if (2 > 1) throw Exception()
        stringInput = ""
    } catch (exception: Exception) {
        // The compiler knows stringInput can be null
        // so stringInput stays nullable.
        println(stringInput?.length)
        // null
    }
}
fun main() {
    testString()
}
```

Smart cast prerequisites

Smart casts work only when the compiler can guarantee that the variable won't change between the check and its usage. They can be used in the following conditions:

`val` local variables	Always, except local delegated properties.
`val` properties	If the property is `private`, `internal`, or if the check is performed in the same module where the property is declared. Smart casts can't be used on `open` properties or properties that have custom getters.
`var` local variables	If the variable is not modified between the check and its usage, is not captured in a lambda that modifies it, and is not a local delegated property.
`var` properties	Never, because the variable can be modified at any time by other code.

`as` and `as?` cast operators

Kotlin has two cast operators: `as` and `as?`. You can use both to cast, but they have different behaviors.

If a cast fails with the `as` operator, a `ClassCastException` is thrown at runtime. That's why it's also called the **unsafe** operator. You can use `as` when casting to a non-null type:

```
fun main() {
    val rawInput: Any = "user-1234"

    // Casts to String successfully
    val userId = rawInput as String
    println("Logging in user with ID: $userId")
    // Logging in user with ID: user-1234

    // Triggers ClassCastException
    val wrongCast = rawInput as Int
    println("wrongCast contains: $wrongCast")
    // Exception in thread "main" java.lang.ClassCastException
}
```

If you use the `as?` operator instead, and the cast fails, the operator returns `null`. That's why it's also called the **safe** operator:

```
fun main() {
    val rawInput: Any = "user-1234"

    // Casts to String successfully
    val userId = rawInput as? String
    println("Logging in user with ID: $userId")
    // Logging in user with ID: user-1234

    // Assigns a null value to wrongCast
    val wrongCast = rawInput as? Int
    println("wrongCast contains: $wrongCast")
    // wrongCast contains: null
}
```

To cast a nullable type safely, use the `as?` operator to prevent triggering a `ClassCastException` if the cast fails.

You *can* use `as` with a nullable type. This allows the result to be `null`, but it still throws a `ClassCastException` if the cast is unsuccessful. For this reason, `as?` is the safer option:

```
fun main() {
    val config: Map<String, Any?> = mapOf(
        "username" to "kodee",
        "alias" to null,
        "loginAttempts" to 3
    )

    // Unsafely casts to a nullable String
    val username: String? = config["username"] as String?
    println("Username: $username")
    // Username: kodee

    // Unsafely casts a null value to a nullable String
    val alias: String? = config["alias"] as String?
    println("Alias: $alias")
    // Alias: null

    // Fails to cast to nullable String and throws ClassCastException
    // val unsafeAttempts: String? = config["loginAttempts"] as String?
    // println("Login attempts (unsafe): $unsafeAttempts")
    // Exception in thread "main" java.lang.ClassCastException

    // Fails to cast to nullable String and returns null
    val safeAttempts: String? = config["loginAttempts"] as? String
    println("Login attempts (safe): $safeAttempts")
    // Login attempts (safe): null
}
```

Up and downcasting

In Kotlin, you can cast objects to supertypes and subtypes.

Casting an object to an instance of its superclass is called **upcasting**. Upcasting doesn't need any special syntax or cast operators. For example:

```
interface Animal {
    fun makeSound()
}

class Dog : Animal {
    // Implements behavior for makeSound()
    override fun makeSound() {
        println("Dog says woof!")
    }
}

fun printAnimalInfo(animal: Animal) {
    animal.makeSound()
}

fun main() {
    val dog = Dog()
    // Upcasts Dog instance to Animal
    printAnimalInfo(dog)
    // Dog says woof!
}
```

In this example, when the `printAnimalInfo()` function is called with a `Dog` instance, the compiler upcasts it to `Animal` because that's the expected parameter type. Since the actual object is still a `Dog` instance, the compiler dynamically resolves the `makeSound()` function from the `Dog` class, printing `"Dog says woof!"`.

You'll often see explicit upcasting in Kotlin APIs where behavior depends on an abstract type. It's also common in Jetpack Compose and UI toolkits, which typically treat all UI elements as supertypes and later operate on specific subclasses:

```
val textView = TextView(this)
    textView.text = "Hello, View!"

    // Upcasts from TextView to View
    val view: View = textView

    // Use View functions
    view.setPadding(20, 20, 20, 20)
    // Activity expects a View type
    setContentView(view)
```

Casting an object to an instance of a subclass is called **downcasting**. Because downcasting can be unsafe, you need to use explicit cast operators. To avoid throwing exceptions on failed casts, we recommend using the safe cast operator `as?`, to return `null` if the cast fails:

```
interface Animal {
    fun makeSound()
}

class Dog : Animal {
    override fun makeSound() {
        println("Dog says woof!")
    }

    fun bark() {
        println("BARK!")
    }
}

fun main() {
    // Creates animal as a Dog instance with Animal
    // type
    val animal: Animal = Dog()

    // Safely downcasts animal to Dog type
    val dog: Dog? = animal as? Dog

    // Uses a safe call to call bark() if dog isn't null
    dog?.bark()
    // "BARK!"
}
```

In this example, `animal` is declared as type `Animal`, but it holds a `Dog` instance. The code safely casts `animal` to `Dog` type and uses a safe call (`?.`) to access the `bark()` function.

You'll use downcasting in serialization when deserializing a base class to a specific subtype. It's also common when working with Java libraries that return supertype objects, which you may need to downcast in Kotlin.

Type aliases

Type aliases provide alternative names for existing types. They can make long or frequently used type expressions shorter and easier to understand.

For example, you can create aliases for generic types, function types, and nested or inner classes:

```
// Generic types
typealias UserIndex = Map<Long, User>
typealias FileTable<K> = MutableMap<K, MutableList<File>>

// Function types
typealias RequestHandler = (Request) -> Response
typealias Predicate<T> = (T) -> Boolean

// Inner and nested classes
class Database {
    inner class Transaction
}
typealias DatabaseTransaction = Database.Transaction
```

A type alias doesn't create a new type. It introduces an alternative name for an existing type. The alias and its underlying type are interchangeable. For example, when you add `typealias Predicate<T>` and use `Predicate<Int>`, the compiler expands it to `(Int) -> Boolean`. You can use a value declared with the alias wherever the underlying type is expected, and the other way around:

```
typealias Predicate<T> = (T) -> Boolean

fun evaluate(predicate: Predicate<Int>) = predicate(42)

fun main() {
    val isPositive: (Int) -> Boolean = { it > 0 }
    println(evaluate(isPositive))
    // true

    val isValid: Predicate<Int> = { it > 0 }
    println(listOf(1, -2).filter(isValid))
    // [1]
}
```

Declare type aliases

You can declare a type alias:

- At the top level of a Kotlin file, as a top-level type alias.
- Inside a class, interface, or object, as a nested type alias.

You can't declare a type alias in a local scope, such as inside a function or lambda expression.

The declaration location determines the scope of a type alias, while its visibility determines which code can access it. By default, a type alias is `public`. A nested type alias is accessible only where its containing class, interface, or object is accessible. For example, a `public` alias inside an `internal` class isn't accessible from outside the module.

A type alias can't expose an underlying type with more restrictive visibility than its own. For example, a `public` type alias can't refer to a `private` class.

Top-level type aliases

A top-level type alias is a package-level declaration. Within the same package, you can refer to an alias by its unqualified name. To use the alias from another package, import the alias or refer to it by its qualified name:

```
// UserId.kt
package org.example.users

typealias UserId = Long

// Refers to the alias within the same package by its unqualified name
fun createUser(id: UserId) {
    // ...
}

// UserService.kt
package org.example.services

import org.example.users.UserId

// Uses the imported alias by its unqualified name
fun findUser(id: UserId) {
    // ...
}

// Uses the fully qualified name
fun deleteUser(id: org.example.users.UserId) {
    // ...
}
```

Nested type aliases

Nested type aliases allow for cleaner, more maintainable code by improving encapsulation, reducing package-level clutter, and simplifying internal implementations. Nested type aliases follow the same scope and name-resolution rules as nested classes.

Declare a type alias inside a class, interface, or object when the alternative name is relevant only in the context of that declaration. This keeps the alias close to the code that uses it and avoids adding another name to the package scope.

Within the containing declaration, you can refer to the alias by its unqualified name. Outside the declaration, qualify the alias with the name of its containing declaration:

```
class UserRepository {
    typealias UserIndex = Map<UserId, User>

    // Refers to the alias by its unqualified name inside UserRepository
    fun saveAll(users: UserIndex) {
        // ...
    }
}

// Refers to the alias by its qualified name outside UserRepository
fun synchronizeUsers(users: UserRepository.UserIndex) {
    // ...
}
```

Nested type aliases aren't supported in Kotlin Multiplatform `expect/actual` declarations.

Type parameters

To use type parameters in a nested type alias, add them to the alias declaration:

```
class Graph<Node> {
    typealias Path<T> = List<T>
}

val cityPath: Graph.Path<String> = listOf("London", "Berlin")
```

In this example, `Path` declares its type parameter `T`. In `Graph.Path<String>`, `String` is the type argument for `T` and is independent of the `Node` type parameter declared by `Graph`.

If you refer to a type parameter declared by its containing class or interface, the compiler reports an error:

```
class Graph<Node> {
    typealias Path = List<Node>
    // Unresolved reference 'Node'.
}
```

Here, `Path` refers to `Node` from `Graph` instead of declaring its own type parameter.

Conditions and loops

Kotlin gives you flexible tools to control your program's flow. Use `if`, `when`, and loops to define clear, expressive logic for your conditions.

If expression

To use `if` in Kotlin, add the condition to check within parentheses `()` and the action to take if the result is true within curly braces `{}`. You can use `else` and `else if` for additional branches and checks.

You can also write `if` as an expression, which lets you assign its returned value directly to a variable. In this form, an `else` branch is required. The `if` expression serves the same purpose as the ternary operator (`condition ? then : else`) found in other languages.

For example:

```
fun main() {
    val heightAlice = 160
    val heightBob = 175

        var taller = heightAlice
    if (heightAlice < heightBob) taller = heightBob

    // Uses an else branch
    if (heightAlice > heightBob) {
        taller = heightAlice
    } else {
        taller = heightBob
    }

    // Uses if as an expression
    taller = if (heightAlice > heightBob) heightAlice else heightBob

    // Uses else if as an expression:
    val heightLimit = 150
    val heightOrLimit = if (heightLimit > heightAlice) heightLimit else if (heightAlice > heightBob)
heightAlice else heightBob

    println("Taller height is $taller")
    // Taller height is 175
    println("Height or limit is $heightOrLimit")
    // Height or limit is 175
    //sampleEnd
}
```

Each branch in an `if` expression can be a block, where the value of the last expression becomes the result:

```
fun main() {
        val heightAlice = 160
    val heightBob = 175

    val taller = if (heightAlice > heightBob) {
        print("Choose Alice\n")
        heightAlice
    } else {
        print("Choose Bob\n")
        heightBob
    }

    println("Taller height is $taller")
    //sampleEnd
}
```

When expressions and statements

`when` is a conditional expression that runs code based on multiple possible values or conditions. It's similar to the `switch` statement in Java, C, and other languages. `when` evaluates its argument and compares the result against each branch in order until one branch condition is satisfied. For example:

```
fun main() {
        val userRole = "Editor"
    when (userRole) {
        "Viewer" -> print("User has read-only access")
        "Editor" -> print("User can edit content")
        else -> print("User role is not recognized")
    }
    // User can edit content
    //sampleEnd
}
```

You can use `when` either as an **expression** or a **statement**. As an expression, `when` returns a value you can use later in your code. As a statement, `when` completes an action without returning a result:

Expression	Statement

`// Returns a string assigned to the`
`// text variable`
`val text = when (x) {`
`    1 -> "x == 1"`
`    2 -> "x == 2"`
`    else -> "x is neither 1 nor 2"`
`}`
	`// Returns no result but triggers a`
	`// print statement`
	`when (x) {`
	`    1 -> print("x == 1")`
	`    2 -> print("x == 2")`
	`    else -> print("x is neither 1 nor 2")`
	`}`

Secondly, you can use `when` with or without a subject. The behavior stays the same either way. Using a subject usually makes your code more readable and maintainable because it clearly shows what you're checking.

With subject `x`	Without subject
`when(x) { ... }`	`when { ... }`

How you use `when` determines whether you need to cover all possible cases in your branches. Covering all possible cases is called being *exhaustive*.

Statements

If you use `when` as a statement, you don't need to cover all possible cases. In this example, some cases aren't covered, so no branch is triggered. However, no error occurs:

```
fun main() {
        val deliveryStatus = "OutForDelivery"
    when (deliveryStatus) {
        // Not all cases are covered
        "Pending" -> print("Your order is being prepared")
        "Shipped" -> print("Your order is on the way")
    }
    //sampleEnd
}
```

Just like with `if`, each branch can be a block, and its value is the value of the last expression in the block.

Expressions

If you use `when` as an expression, you **must** cover all possible cases. The value of the first matching branch becomes the value of the overall expression. If you don't cover all cases, the compiler throws an error.

If your `when` expression has a subject, you can use an `else` branch to make sure that all possible cases are covered, but it isn't mandatory. For example, if your subject is a `Boolean`, `enum class`, `sealed class`, or one of their nullable counterparts, you can cover all cases without an `else` branch:

```
import kotlin.random.Random
enum class Bit {
    ZERO, ONE
}

fun getRandomBit(): Bit {
    return if (Random.nextBoolean()) Bit.ONE else Bit.ZERO
}

fun main() {
    val numericValue = when (getRandomBit()) {
        // No else branch is needed because all cases are covered
        Bit.ZERO -> 0
        Bit.ONE -> 1
    }

    println("Random bit as number: $numericValue")
    // Random bit as number: 0
    //sampleEnd
}
```

> To simplify `when` expressions and reduce repetition, try out context-sensitive resolution (currently in preview). This feature allows you to omit the type name when using enum entries or sealed class members in `when` expressions if the expected type is known.
>
> For more information, see Preview of context-sensitive resolution or the related KEEP proposal.

If your `when` expression **doesn't** have a subject, you **must** have an `else` branch or the compiler throws an error. The `else` branch is evaluated when none of the other branch conditions are satisfied:

```
fun main() {
        val localFileSize = 1200
    val remoteFileSize = 1200

    val message = when {
        localFileSize > remoteFileSize -> "Local file is larger than remote file"
        localFileSize < remoteFileSize -> "Local file is smaller than remote file"
        else -> "Local and remote files are the same size"
    }

    println(message)
    // Local and remote files are the same size
    //sampleEnd
}
```

Other ways to use when

`when` expressions and statements offer different ways to simplify your code, handle multiple conditions, and perform type checks.

Group multiple conditions into a single branch using commas:

```
fun main() {
    val ticketPriority = "High"
        when (ticketPriority) {
        "Low", "Medium" -> print("Standard response time")
        else -> print("High-priority handling")
    }
    //sampleEnd
}
```

Use expressions that evaluate to `true` or `false` as branch conditions:

```
fun main() {
    val storedPin = "1234"
    val enteredPin = 1234

        when (enteredPin) {
        // Expression
        storedPin.toInt() -> print("PIN is correct")
        else -> print("Incorrect PIN")
    }
    //sampleEnd
}
```

Check whether a value is or isn't contained in a range or collection using the `in` or `!in` keywords:

```
fun main() {
    val x = 7
    val validNumbers = setOf(15, 16, 17)

        when (x) {
        in 1..10 -> print("x is in the range")
        in validNumbers -> print("x is valid")
        !in 10..20 -> print("x is outside the range")
        else -> print("none of the above")
    }
    //sampleEnd
}
```

Check a value's type using the `is` or `!is` keywords. Due to smart casts, you can access the member functions and properties of the type directly:

```
fun hasPrefix(input: Any): Boolean = when (input) {
    is String -> input.startsWith("ID-")
    else -> false
}

fun main() {
    val testInput = "ID-98345"
    println(hasPrefix(testInput))
    // true
}
```

Use `when` instead of a traditional `if` - `else` `if` chain. Without a subject, the branch conditions are simply boolean expressions. The first branch with a `true` condition runs:

```
fun Int.isOdd() = this % 2 != 0
fun Int.isEven() = this % 2 == 0

fun main() {
        val x = 5
    val y = 8

    when {
        x.isOdd() -> print("x is odd")
        y.isEven() -> print("y is even")
        else -> print("x+y is odd")
    }
    // x is odd
    //sampleEnd
}
```

Finally, capture the subject in a variable by using the following syntax:

```
fun main() {
    val message = when (val input = "yes") {
        "yes" -> "You said yes"
        "no" -> "You said no"
        else -> "Unrecognized input: $input"
    }

    println(message)
    // You said yes
}
```

The scope of a variable introduced as the subject is restricted to the body of the `when` expression or statement.

Guard conditions

Guard conditions allow you to include more than one condition to the branches of a `when` expression or statement, making complex control flow more explicit and concise. You can use guard conditions with `when` as long as it has a subject.

Place a guard condition after the primary condition in the same branch, separated by `if`:

```
sealed interface Animal {
    data class Cat(val mouseHunter: Boolean) : Animal
    data class Dog(val breed: String) : Animal
}

fun feedDog() = println("Feeding a dog")
fun feedCat() = println("Feeding a cat")

fun feedAnimal(animal: Animal) {
    when (animal) {
        // Branch with only primary condition
        // Calls feedDog() when animal is Dog
        is Animal.Dog -> feedDog()
        // Branch with both primary and guard conditions
        // Calls feedCat() when animal is Cat and not mouseHunter
        is Animal.Cat if !animal.mouseHunter -> feedCat()
        // Prints "Unknown animal" if none of the above conditions match
        else -> println("Unknown animal")
    }
}

fun main() {
    val animals = listOf(
        Animal.Dog("Beagle"),
        Animal.Cat(mouseHunter = false),
        Animal.Cat(mouseHunter = true)
    )

    animals.forEach { feedAnimal(it) }
    // Feeding a dog
    // Feeding a cat
    // Unknown animal
}
```

You can't use guard conditions when you have multiple conditions separated by a comma. For example:

```
0, 1 -> print("x == 0 or x == 1")
```

In a single `when` expression or statement, you can combine branches with and without guard conditions. The code in a branch with a guard condition runs only if both the primary condition and the guard condition evaluate to `true`. If the primary condition doesn't match, the guard condition isn't evaluated.

Since `when` statements don't need to cover all cases, using guard conditions in `when` statements without an `else` branch means that if no conditions match, no code is run.

Unlike statements, `when` expressions must cover all cases. If you use guard conditions in `when` expressions without an `else` branch, the compiler requires you to handle every possible case to avoid runtime errors.

Combine multiple guard conditions within a single branch using the boolean operators `&&` (AND) or `||` (OR). Use parentheses around the boolean expressions to avoid confusion:

```
when (animal) {
    is Animal.Cat if (!animal.mouseHunter && animal.hungry) -> feedCat()
}
```

Guard conditions also support `else if`:

```
when (animal) {
    // Checks if `animal` is `Dog`
    is Animal.Dog -> feedDog()
    // Guard condition that checks if `animal` is `Cat` and not `mouseHunter`
    is Animal.Cat if !animal.mouseHunter -> feedCat()
    // Calls giveLettuce() if none of the above conditions match and animal.eatsPlants is true
    else if animal.eatsPlants -> giveLettuce()
    // Prints "Unknown animal" if none of the above conditions match
    else -> println("Unknown animal")
}
```

Bytecode generation on the JVM

When you compile Kotlin code for JVM 21 or later, the compiler generates an `invokedynamic` instruction for eligible `when` expressions. This produces smaller bytecode, similar to the bytecode produced by Java `switch` statements.

The compiler uses `invokedynamic` with the `SwitchBootstraps.typeSwitch()` method when all the following conditions are met:

- All conditions except for `else` are `is` or `null` checks.
- The `when` expression doesn't contain guard conditions (`if`).
- The conditions don't include types that can't be type-checked directly, such as mutable Kotlin collections (`MutableList`) or function types (`kotlin.Function1` , `kotlin.Function2` , and so on).
- The `when` expression has at least two conditions besides `else` .
- All branches check the same subject of the `when` expression.

For example:

```
open class Shape

class Circle : Shape()
class Rectangle : Shape()
class Triangle : Shape()

fun countCorners(shape: Shape) = when (shape) {
    is Circle -> 0
    is Rectangle -> 4
    is Triangle -> 3
    else -> -1
}
```

The `when (shape)` expression here compiles to a single `invokedynamic` type switch instead of multiple `instanceof` checks in the bytecode.

For loops

Use the `for` loop to iterate through a collection, array, or range:

```
for (item in collection) print(item)
```

The body of a `for` loop can be a block with curly braces `{}` .

```
fun main() {
    val shoppingList = listOf("Milk", "Bananas", "Bread")
        println("Things to buy:")
    for (item in shoppingList) {
        println("- $item")
    }
    // Things to buy:
    // - Milk
    // - Bananas
    // - Bread
    //sampleEnd
}
```

Ranges

To iterate over a range of numbers, use a range expression with `..` and `..<` operators:

```
fun main() {
    println("Closed-ended range:")
    for (i in 1..6) {
        print(i)
    }
    // Closed-ended range:
    // 123456

    println("\nOpen-ended range:")
    for (i in 1..<6) {
        print(i)
    }
    // Open-ended range:
    // 12345

    println("\nReverse order in steps of 2:")
    for (i in 6 downTo 0 step 2) {
        print(i)
    }
    // Reverse order in steps of 2:
    // 6420
}
```

Arrays

If you want to iterate through an array or a list with an index, you can use the `indices` property:

```
fun main() {
    val routineSteps = arrayOf("Wake up", "Brush teeth", "Make coffee")
        for (i in routineSteps.indices) {
        println(routineSteps[i])
    }
    // Wake up
    // Brush teeth
    // Make coffee
    //sampleEnd
}
```

Alternatively, you can use the `.withIndex()` function from the standard library:

```
fun main() {
    val routineSteps = arrayOf("Wake up", "Brush teeth", "Make coffee")
        for ((index, value) in routineSteps.withIndex()) {
        println("The step at $index is \"$value\"")
    }
    // The step at 0 is "Wake up"
    // The step at 1 is "Brush teeth"
    // The step at 2 is "Make coffee"
    //sampleEnd
}
```

Iterators

The `for` loop iterates through anything that provides an iterator. Collections provide iterators by default, whereas ranges and arrays are compiled into index-based loops.

You can create your own iterators by providing a member or extension function called `iterator()` that returns an `Iterator<>`. The `iterator()` function must have a `next()` function and a `hasNext()` function that returns a `Boolean`.

The easiest way to create your own iterator for a class is to inherit from the `Iterable<T>` interface and override the `iterator()`, `next()`, and `hasNext()` functions that are already there. For example:

```
class Booklet(val totalPages: Int) : Iterable<Int> {
    override fun iterator(): Iterator<Int> {
        return object : Iterator<Int> {
            var current = 1
            override fun hasNext() = current <= totalPages
            override fun next() = current++
        }
    }
}

fun main() {
    val booklet = Booklet(3)
    for (page in booklet) {
        println("Reading page $page")
    }
    // Reading page 1
    // Reading page 2
    // Reading page 3
}
```

Learn more about interfaces and inheritance.

Alternatively, you can create the functions from scratch. In this case, add the `operator` keyword to the functions:

```
class Booklet(val totalPages: Int) {
    operator fun iterator(): Iterator<Int> {
        return object {
            var current = 1

            operator fun hasNext() = current <= totalPages
            operator fun next() = current++
        }.let {
            object : Iterator<Int> {
                override fun hasNext() = it.hasNext()
                override fun next() = it.next()
            }
        }
    }
}

fun main() {
    val booklet = Booklet(3)
    for (page in booklet) {
        println("Reading page $page")
    }
    // Reading page 1
    // Reading page 2
    // Reading page 3
}
```

While loops

`while` and `do-while` loops run the code in their body continuously while the condition is satisfied. The difference between them is the condition checking time:

- `while` checks the condition and, if it's satisfied, runs the code in its body and then returns to the condition check.
- `do-while` runs the code in its body and then checks the condition. If it's satisfied, the loop repeats. So, the body of `do-while` runs at least once, regardless of the condition.

For a `while` loop, place the condition to check in parentheses `()` and the body within curly braces `{}`:

```
fun main() {
    var carsInGarage = 0
    val maxCapacity = 3
    while (carsInGarage < maxCapacity) {
        println("Car entered. Cars now in garage: ${++carsInGarage}")
    }
    // Car entered. Cars now in garage: 1
    // Car entered. Cars now in garage: 2
    // Car entered. Cars now in garage: 3

    println("Garage is full!")
    // Garage is full!
}
```

For a `do-while` loop, place the body within curly braces `{}` first before the condition to check in parentheses `()`:

```
import kotlin.random.Random

fun main() {
    var roll: Int
    do {
        roll = Random.nextInt(1, 7)
        println("Rolled a $roll")
    } while (roll ≠ 6)
    // Rolled a 2
    // Rolled a 6

    println("Got a 6! Game over.")
    // Got a 6! Game over.
}
```

Break and continue in loops

Kotlin supports traditional `break` and `continue` operators in loops. See Returns and jumps.

Returns and jumps

Kotlin has three structural jump expressions:

- `return` by default returns from the nearest enclosing function or anonymous function.
- `break` terminates the nearest enclosing loop.
- `continue` proceeds to the next step of the nearest enclosing loop.

All of these expressions can be used as part of larger expressions:

```
val s = person.name ?: return
```

The type of these expressions is the Nothing type.

Break and continue labels

Any expression in Kotlin may be marked with a *label*. Labels have the form of an identifier followed by the `@` sign, such as `abc@` or `fooBar@`. To label an expression, just add a label in front of it.

```
loop@ for (i in 1..100) {
    // ...
}
```

Now, you can qualify a `break` or a `continue` with a label:

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}
```

A `break` qualified with a label jumps to the execution point right after the loop marked with that label. A `continue` proceeds to the next iteration of that loop.

In some cases, you can apply `break` and `continue` *non-locally* without explicitly defining labels. Such non-local usages are valid in lambda expressions used in enclosing inline functions.

Return to labels

In Kotlin, functions can be nested using function literals, local functions, and object expressions. A qualified `return` allows you to return from an outer function.

The most important use case is returning from a lambda expression. To return from a lambda expression, label it and qualify the `return`:

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach lit@{
        if (it == 3) return@lit // local return to the caller of the lambda - the forEach loop
        print(it)
    }
    print(" done with explicit label")
}

fun main() {
    foo()
}
```

Now, it returns only from the lambda expression. Often it is more convenient to use *implicit labels*, because such a label has the same name as the function to which the lambda is passed.

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return@forEach // local return to the caller of the lambda - the forEach loop
        print(it)
    }
    print(" done with implicit label")
}

fun main() {
    foo()
}
```

Alternatively, you can replace the lambda expression with an anonymous function. A `return` statement in an anonymous function will return from the anonymous function itself.

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach(fun(value: Int) {
        if (value == 3) return  // local return to the caller of the anonymous function - the forEach
loop
        print(value)
    })
    print(" done with anonymous function")
}

fun main() {
    foo()
}
```

Note that the use of local returns in the previous three examples is similar to the use of `continue` in regular loops.

There is no direct equivalent for `break`, but it can be simulated by adding an outer `run` lambda and non-locally returning from it:

```
fun foo() {
    run loop@{
        listOf(1, 2, 3, 4, 5).forEach {
            if (it == 3) return@loop // non-local return from the lambda passed to run
            print(it)
        }
    }
    print(" done with nested loop")
}

fun main() {
    foo()
}
```

The non-local return here is possible since the nested `forEach()` lambda acts as an inline function.

When returning a value, the parser gives preference to the qualified return:

```
return@a 1
```

This means "return `1` at label `@a` " rather than "return a labeled expression `(@a 1)` ".

> In some cases, you can return from a lambda expression without using labels. Such *non-local* returns are located in a lambda but exit the enclosing inline function.

Exception and error handling

Exceptions help your code run more predictably, even when runtime errors occur that could disrupt program execution. Kotlin treats all exceptions as *unchecked* by default. Unchecked exceptions simplify the exception handling process: you can catch exceptions, but you don't need to explicitly handle or declare them.

> Learn more about how Kotlin handles exceptions when interacting with Java, Swift, and Objective-C in the Exception interoperability with Java, Swift, and Objective-C section.

Working with exceptions consists of two primary actions:

- **Throwing exceptions:** indicate when a problem occurs.
- **Catching exceptions:** handle the unexpected exception manually by resolving the issue or notifying the developer or application user.

Exceptions are represented by subclasses of the `Exception` class, which is a subclass of the `Throwable` class. For more information about the hierarchy, see the Exception hierarchy section. Since `Exception` is an `open class`, you can create custom exceptions to suit your application's specific needs.

Throw exceptions

You can manually throw exceptions with the `throw` keyword. Throwing an exception indicates that an unexpected runtime error has occurred in the code. Exceptions are objects, and throwing one creates an instance of an exception class.

You can throw an exception without any parameters:

```
throw IllegalArgumentException()
```

To better understand the source of the problem, include additional information, such as a custom message and the original cause:

```
val cause = IllegalStateException("Original cause: illegal state")

// Throws an IllegalArgumentException if userInput is negative
// Additionally, it shows the original cause, represented by the cause IllegalStateException
if (userInput < 0) {
    throw IllegalArgumentException("Input must be non-negative", cause)
}
```

In this example, an `IllegalArgumentException` is thrown when the user inputs a negative value. You can create custom error messages and keep the original cause (`cause`) of the exception, which will be included in the stack trace.

Throw exceptions with precondition functions

Kotlin offers additional ways to automatically throw exceptions using precondition functions. Precondition functions include:

Precondition function	Use case	Exception thrown
`require()`	Checks user input validity	`IllegalArgumentException`
`check()`	Checks object or variable state validity	`IllegalStateException`
`error()`	Indicates an illegal state or condition	`IllegalStateException`

These functions are suitable for situations where the program's flow cannot continue if specific conditions aren't met. This streamlines your code and makes handling these checks efficient.

require() function

Use the `require()` function to validate input arguments when they are crucial for the function's operation, and the function can't proceed if these arguments are invalid.

If the condition in `require()` is not met, it throws an `IllegalArgumentException` :

```
fun getIndices(count: Int): List<Int> {
    require(count ⩾ 0) { "Count must be non-negative. You set count to $count." }
    return List(count) { it + 1 }
}

fun main() {
    // This fails with an IllegalArgumentException
    println(getIndices(-1))

    // Uncomment the line below to see a working example
    // println(getIndices(3))
    // [1, 2, 3]
}
```

The `require()` function allows the compiler to perform smart casting. After a successful check, the variable is automatically cast to a non-nullable type. These functions are often used for nullability checks to ensure that the variable is not null before proceeding. For example:

```
fun printNonNullString(str: String?) {
    // Nullability check
    require(str ≠ null)
    // After this successful check, 'str' is guaranteed to be
    // non-null and is automatically smart cast to non-nullable String
    println(str.length)
}
```

check() function

Use the `check()` function to validate the state of an object or variable. If the check fails, it indicates a logic error that needs to be addressed.

If the condition specified in the `check()` function is `false`, it throws an `IllegalStateException`:

```
fun main() {
    var someState: String? = null

    fun getStateValue(): String {

        val state = checkNotNull(someState) { "State must be set beforehand!" }
        check(state.isNotEmpty()) { "State must be non-empty!" }
        return state
    }
    // If you uncomment the line below then the program fails with IllegalStateException
    // getStateValue()

    someState = ""

    // If you uncomment the line below then the program fails with IllegalStateException
    // getStateValue()
    someState = "non-empty-state"

    // This prints "non-empty-state"
    println(getStateValue())
}
```

The `check()` function allows the compiler to perform smart casting. After a successful check, the variable is automatically cast to a non-nullable type. These functions are often used for nullability checks to ensure that the variable is not null before proceeding. For example:

```
fun printNonNullString(str: String?) {
    // Nullability check
    check(str != null)
    // After this successful check, 'str' is guaranteed to be
    // non-null and is automatically smart cast to non-nullable String
    println(str.length)
}
```

error() function

The `error()` function is used to signal an illegal state or a condition in the code that logically should not occur. It's suitable for scenarios when you want to throw an exception intentionally in your code, such as when the code encounters an unexpected state. This function is particularly useful in `when` expressions, providing a clear way to handle cases that shouldn't logically happen.

In the following example, the `error()` function is used to handle an undefined user role. If the role is not one of the predefined ones, an `IllegalStateException` is thrown:

```
class User(val name: String, val role: String)

fun processUserRole(user: User) {
    when (user.role) {
        "admin" -> println("${user.name} is an admin.")
        "editor" -> println("${user.name} is an editor.")
        "viewer" -> println("${user.name} is a viewer.")
        else -> error("Undefined role: ${user.role}")
    }
}

fun main() {
    // This works as expected
    val user1 = User("Alice", "admin")
    processUserRole(user1)
    // Alice is an admin.

    // This throws an IllegalStateException
    val user2 = User("Bob", "guest")
    processUserRole(user2)
}
```

Handle exceptions using try-catch blocks

When an exception is thrown, it interrupts the normal execution of the program. You can handle exceptions gracefully with the `try` and `catch` keywords to keep your program stable. The `try` block contains the code that might throw an exception, while the `catch` block catches and handles the exception if it occurs. The exception is caught by the first `catch` block that matches its specific type or a superclass of the exception.

Here's how you can use the `try` and `catch` keywords together:

```
try {
    // Code that may throw an exception
} catch (e: SomeException) {
    // Code for handling the exception
}
```

It's a common approach to use `try-catch` as an expression, so it can return a value from either the `try` block or the `catch` block:

```
fun main() {
    val num: Int = try {

        // If count() completes successfully, its return value is assigned to num
        count()

    } catch (e: ArithmeticException) {

        // If count() throws an exception, the catch block returns -1,
        // which is assigned to num
        -1
    }
    println("Result: $num")
}

// Simulates a function that might throw ArithmeticException
fun count(): Int {

    // Change this value to return a different value to num
    val a = 0

    return 10 / a
}
```

You can handle an exception without using the exception instance. For example, you can provide a fallback value or a generic error message in the `catch` block.

Use an underscore (`_`) instead of the exception parameter name to indicate that the exception instance is intentionally ignored:

```
import java.io.File
import java.io.IOException

fun main() {
    val userSettings = try {
        File("user-settings.json").readText()

    // Catches IOException without using the exception instance
    } catch (_: IOException) {
        // Uses a fallback value if loading the file fails
        "{}"
    }

    println(userSettings)
}
```

You can use multiple `catch` handlers for the same `try` block. You can add as many `catch` blocks as needed to handle different exceptions distinctively. When you have multiple `catch` blocks, it's important to order them from the most specific to the least specific exception, following a top-to-bottom order in your code. This ordering aligns with the program's execution flow.

Consider this example with custom exceptions:

```
open class WithdrawalException(message: String) : Exception(message)
class InsufficientFundsException(message: String) : WithdrawalException(message)

fun processWithdrawal(amount: Double, availableFunds: Double) {
    if (amount > availableFunds) {
        throw InsufficientFundsException("Insufficient funds for the withdrawal.")
    }
    if (amount < 1 || amount % 1 != 0.0) {
        throw WithdrawalException("Invalid withdrawal amount.")
    }
    println("Withdrawal processed")
}

fun main() {
    val availableFunds = 500.0

    // Change this value to test different scenarios
    val withdrawalAmount = 500.5

    try {
        processWithdrawal(withdrawalAmount.toDouble(), availableFunds)

    // The order of catch blocks is important!
    } catch (e: InsufficientFundsException) {
        println("Caught an InsufficientFundsException: ${e.message}")
    } catch (e: WithdrawalException) {
        println("Caught a WithdrawalException: ${e.message}")
    }
}
```

A general catch block handling `WithdrawalException`, catches all exceptions of its type, including specific ones like `InsufficientFundsException`, unless they are caught earlier by a more specific catch block.

The finally block

The `finally` block contains code that always executes, regardless of whether the `try` block completes successfully or throws an exception. With the `finally` block, you can clean up code after the execution of `try` and `catch` blocks. This is especially important when working with resources like files or network connections, as `finally` guarantees they are properly closed or released.

Here is how you would typically use the `try-catch-finally` blocks together:

```
try {
    // Code that may throw an exception
}
catch (e: YourException) {
    // Exception handler
}
finally {
    // Code that is always executed
}
```

The returned value of a `try` expression is determined by the last executed expression in either the `try` or `catch` block. If no exceptions occur, the result comes from the `try` block; if an exception is handled, it comes from the `catch` block. The `finally` block is always executed, but it doesn't change the result of the `try-catch` block.

Let's look at an example to demonstrate:

```
fun divideOrNull(a: Int): Int {

    // The try block is always executed
    // An exception here (division by zero) causes an immediate jump to the catch block
    try {
        val b = 44 / a
        println("try block: Executing division: $b")
        return b
    }

    // The catch block is executed due to the ArithmeticException (division by zero if a ==0)
    catch (e: ArithmeticException) {
        println("catch block: Encountered ArithmeticException $e")
        return -1
    }
    finally {
        println("finally block: The finally block is always executed")
    }
}

fun main() {

    // Change this value to get a different result. An ArithmeticException will return: -1
    divideOrNull(0)
}
```

In Kotlin, the idiomatic way to manage resources that implement the `AutoClosable` interface, such as file streams like `FileInputStream` or `FileOutputStream`, is to use the `.use()` function. This function automatically closes the resource when the block of code completes, regardless of whether an exception is thrown, thereby eliminating the need for a `finally` block. Consequently, Kotlin does not require a special syntax like Java's try-with-resources for resource management.

```
FileWriter("test.txt").use { writer →
    writer.write("some text")
    // After this block, the .use function automatically calls writer.close(), similar to a
finally block
}
```

If your code requires resource cleanup without handling exceptions, you can also use `try` with the `finally` block without `catch` blocks:

```
class MockResource {
    fun use() {
        println("Resource being used")
        // Simulate a resource being used
        // This throws an ArithmeticException if division by zero occurs
        val result = 100 / 0

        // This line is not executed if an exception is thrown
        println("Result: $result")
    }

    fun close() {
        println("Resource closed")
    }
}

fun main() {
    val resource = MockResource()
//sampleStart
    try {

        // Attempts to use the resource
        resource.use()

    } finally {

        // Ensures that the resource is always closed, even if an exception occurs
        resource.close()
    }

    // This line is not printed if an exception is thrown
    println("End of the program")
}
```

As you can see, the `finally` block guarantees that the resource is closed, regardless of whether an exception occurs.

In Kotlin, you have the flexibility to use only a `catch` block, only a `finally` block, or both, depending on your specific needs, but a `try` block must always be accompanied by at least one `catch` block or a `finally` block.

Create custom exceptions

In Kotlin, you can define custom exceptions by creating classes that extend the built-in `Exception` class. This allows you to create more specific error types tailored to your application's needs.

To create one, you can define a class that extends `Exception`:

```
class MyException: Exception("My message")
```

In this example, there is a default error message, "My message", but you can leave it blank if you want.

> Exceptions in Kotlin are stateful objects, carrying information specific to the context of their creation, referred to as the stack trace. Avoid creating exceptions using object declarations. Instead, create a new instance of the exception every time you need one. This way, you can ensure the exception's state accurately reflects the specific context.

Custom exceptions can also be a subclass of any pre-existent exception subclass, like the `ArithmeticException` subclass:

```
class NumberTooLargeException: ArithmeticException("My message")
```

If you want to create subclasses of custom exceptions, you must declare the parent class as `open` because classes are final by default and cannot be subclassed otherwise.

For example:

```
// Declares a custom exception as an open class, making it subclassable
open class MyCustomException(message: String): Exception(message)

// Creates a subclass of the custom exception
class SpecificCustomException: MyCustomException("Specific error message")
```

Custom exceptions behave just like built-in exceptions. You can throw them using the `throw` keyword, and handle them with `try-catch-finally` blocks. Let's look at an example to demonstrate:

```
class NegativeNumberException: Exception("Parameter is less than zero.")
class NonNegativeNumberException: Exception("Parameter is a non-negative number.")

fun myFunction(number: Int) {
    if (number < 0) throw NegativeNumberException()
    else if (number ⩾ 0) throw NonNegativeNumberException()
}

fun main() {

    // Change the value in this function to a get a different exception
    myFunction(1)
}
```

In applications with diverse error scenarios, creating a hierarchy of exceptions can help make the code clearer and more specific. You can achieve this by using an abstract class or a sealed class as a base for common exception features and creating specific subclasses for detailed exception types. Additionally, custom exceptions including parameters with default values offer flexibility, allowing initialization with varied messages, which enables more granular error handling.

Let's look at an example using the sealed class `AccountException` as the base for an exception hierarchy, and class `APIKeyExpiredException`, a subclass, which showcases the use of parameters with default values for improved exception detail:

```
// Creates a sealed class as the base for an exception hierarchy for account-related errors
sealed class AccountException(message: String, cause: Throwable? = null):
Exception(message, cause)

// Creates a subclass of AccountException
class InvalidAccountCredentialsException : AccountException("Invalid account credentials detected")

// Creates a subclass of AccountException, which allows the addition of custom messages and causes
class APIKeyExpiredException(message: String = "API key expired", cause: Throwable? = null)     :
AccountException(message, cause)

// Change values of placeholder functions to get different results
fun areCredentialsValid(): Boolean = true
fun isAPIKeyExpired(): Boolean = true

// Validates account credentials and API key
fun validateAccount() {
    if (!areCredentialsValid()) throw InvalidAccountCredentialsException()
    if (isAPIKeyExpired()) {
        // Example of throwing APIKeyExpiredException with a specific cause
        val cause = RuntimeException("API key validation failed due to network error")
        throw APIKeyExpiredException(cause = cause)
    }
}

fun main() {
    try {
        validateAccount()
        println("Operation successful: Account credentials and API key are valid.")
    } catch (e: AccountException) {
        println("Error: ${e.message}")
        e.cause?.let { println("Caused by: ${it.message}") }
    }
}
```

The Nothing type

In Kotlin, every expression has a type. The type of the expression `throw IllegalArgumentException()` is `Nothing`, a built-in type that is a subtype of all other types, also known as the bottom type. This means `Nothing` can be used as a return type or generic type where any other type is expected, without causing type errors.

`Nothing` is a special type in Kotlin used to represent functions or expressions that never complete successfully, either because they always throw an exception or enter an endless execution path like an infinite loop. You can use `Nothing` to mark functions that are not yet implemented or are designed to always throw an exception, clearly indicating your intentions to both the compiler and code readers. If the compiler infers a `Nothing` type in a function signature, it will warn you. Explicitly defining `Nothing` as the return type can eliminate this warning.

This Kotlin code demonstrates the use of the `Nothing` type, where the compiler marks the code following the function call as unreachable:

```kotlin
class Person(val name: String?)

fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
    // This function will never return successfully.
    // It will always throw an exception.
}

fun main() {
    // Creates an instance of Person with 'name' as null
    val person = Person(name = null)

    val s: String = person.name ?: fail("Name required")

    // 's' is guaranteed to be initialized at this point
    println(s)
}
```

Kotlin's `TODO()` function, which also uses the `Nothing` type, serves as a placeholder to highlight areas of the code that need future implementation:

```kotlin
fun notImplementedFunction(): Int {
    TODO("This function is not yet implemented")
}

fun main() {
    val result = notImplementedFunction()
    // This throws a NotImplementedError
    println(result)
}
```

As you can see, the `TODO()` function always throws a `NotImplementedError` exception.

Exception classes

Let's explore some common exception types found in Kotlin, which are all subclasses of the `RuntimeException` class:

- `ArithmeticException` : This exception occurs when an arithmetic operation is impossible to perform, like division by zero.

  ```kotlin
  val example = 2 / 0 // throws ArithmeticException
  ```

- `IndexOutOfBoundsException` : This exception is thrown to indicate that an index of some sort, such as an array or string, is out of range.

  ```kotlin
  val myList = mutableListOf(1, 2, 3)
  myList.removeAt(3)  // throws IndexOutOfBoundsException
  ```

To avoid this exception, use a safer alternative, such as the `getOrNull()` function:

```
val myList = listOf(1, 2, 3)
// Returns null, instead of IndexOutOfBoundsException
val element = myList.getOrNull(3)
println("Element at index 3: $element")
```

- `NoSuchElementException` : This exception is thrown when an element that does not exist in a particular collection is accessed. It occurs when using methods that expect a specific element, such as `first()` or `last()` .

```
val emptyList = listOf<Int>()
val firstElement = emptyList.first()  // throws NoSuchElementException
```

To avoid this exception, use a safer alternative, such as the `firstOrNull()` function:

```
val emptyList = listOf<Int>()
// Returns null, instead of NoSuchElementException
val firstElement = emptyList.firstOrNull()
println("First element in empty list: $firstElement")
```

- `NumberFormatException` : This exception occurs when attempting to convert a string to a numeric type, but the string doesn't have an appropriate format.

```
val string = "This is not a number"
val number = string.toInt() // throws NumberFormatException
```

To avoid this exception, use a safer alternative, such as the `toIntOrNull()` function:

```
val nonNumericString = "not a number"
// Returns null, instead of NumberFormatException
val number = nonNumericString.toIntOrNull()
println("Converted number: $number")
```

- `NullPointerException` : This exception is thrown when an application attempts to use an object reference that has the `null` value. Even though Kotlin's null safety features significantly reduce the risk of NullPointerExceptions, they can still occur either through deliberate use of the `!!` operator or when interacting with Java, which lacks Kotlin's null safety.

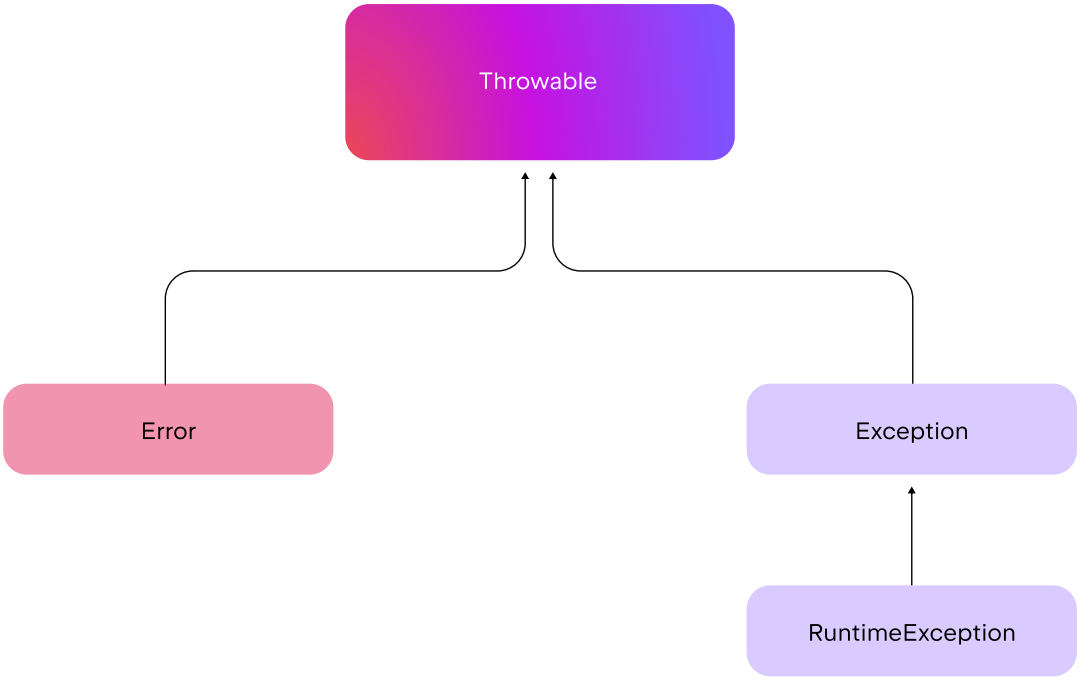
```
val text: String? = null
println(text!!.length)  // throws a NullPointerException
```

While all exceptions are unchecked in Kotlin, and you don't have to catch them explicitly, you still have the flexibility to catch them if desired.

Exception hierarchy

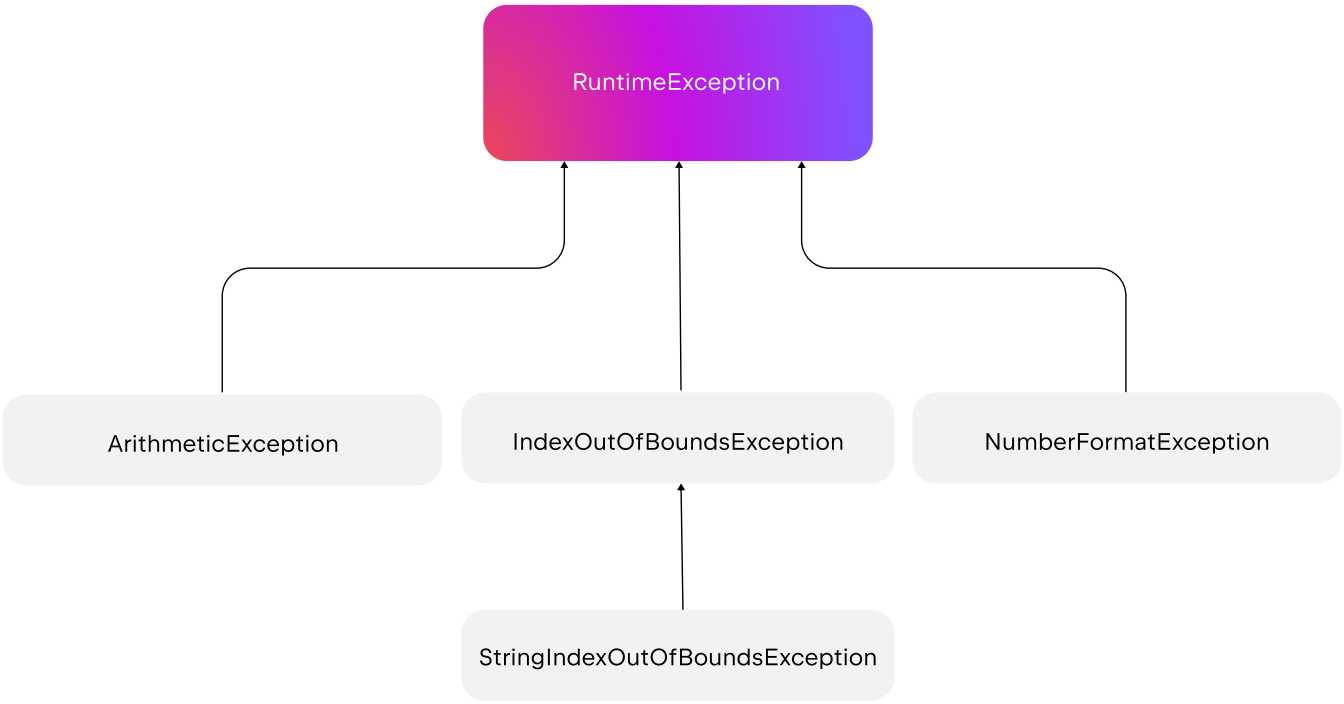
The root of the Kotlin exception hierarchy is the `Throwable` class. It has two direct subclasses, `Error` and `Exception`:

- The `Error` subclass represents serious fundamental problems that an application might not be able to recover from by itself. These are problems that you generally would not attempt to handle, such as `OutOfMemoryError` or `StackOverflowError`.
- The `Exception` subclass is used for conditions that you might want to handle. Subtypes of the `Exception` type, such as the `RuntimeException` and `IOException` (Input/Output Exception), deal with exceptional events in applications.



Exception hierarchy - the Throwable class

`RuntimeException` is usually caused by insufficient checks in the program code and can be prevented programmatically. Kotlin helps prevent common `RuntimeExceptions` like `NullPointerException` and provides compile-time warnings for potential runtime errors, such as division by zero. The following picture demonstrates a hierarchy of subtypes descended from `RuntimeException`:



Hierarchy of RuntimeExceptions

Stack trace

The *stack trace* is a report generated by the runtime environment, used for debugging. It shows the sequence of function calls leading to a specific point in the program, especially where an error or exception occurred.

Let's see an example where the stack trace is automatically printed because of an exception in a JVM environment:

```
fun main() {
//sampleStart
    throw ArithmeticException("This is an arithmetic exception!")
}
```

Running this code in a JVM environment produces the following output:

```
Exception in thread "main" java.lang.ArithmeticException: This is an arithmetic exception!
    at MainKt.main(Main.kt:3)
    at MainKt.main(Main.kt)
```

The first line is the exception description, which includes:

- Exception type: `java.lang.ArithmeticException`
- Thread: `main`
- Exception message: `"This is an arithmetic exception!"`

Each other line that starts with an `at` after the exception description is the stack trace. A single line is called a *stack trace element* or a *stack frame*:

- `at MainKt.main (Main.kt:3)`: This shows the method name (`MainKt.main`) and the source file and line number where the method was called (`Main.kt:3`).
- `at MainKt.main (Main.kt)`: This shows that the exception occurs in the `main()` function of the `Main.kt` file.

Exception interoperability with Java, Swift, and Objective-C

Since Kotlin treats all exceptions as unchecked, it can lead to complications when such exceptions are called from languages that distinguish between checked and unchecked exceptions. To address this disparity in exception handling between Kotlin and languages like Java, Swift, and Objective-C, you can use the `@Throws` annotation. This annotation alerts callers about possible exceptions. For more information, see Calling Kotlin from Java and Interoperability with Swift/Objective-C.

Functions

To declare a function in Kotlin:

- Use the `fun` keyword.
- Specify the parameters in parentheses `()`.
- Include the return type if needed.

For example:

```
// 'double' is the name of the function
// 'x' is a parameter of Int type
// The expected return value is of Int type too
fun double(x: Int): Int {
    return 2 * x
}

fun main() {
    println(double(5))
    // 10
}
```

Function usage

Functions are called using the standard approach:

```
val result = double(2)
```

To call a member or extension function, use a period `.`:

```
// Creates an instance of the Stream class and calls read()
Stream().read()
```

Parameters

Declare function parameters using Pascal notation: `name: Type`. You must separate parameters using commas and give each parameter a type explicitly:

```
fun powerOf(number: Int, exponent: Int): Int { /*...*/ }
```

When you pass an object to a function, the compiler passes a copy of the reference to that object. The copied reference points to the same object, so the function can modify the object's mutable state.

Function parameters are read-only inside the function body (implicitly declared as `val`), so you can't reassign them:

```
class Counter(var value: Int)

fun reset(counter: Counter) {
    counter.value = 0    // Allowed: modifies the object
    counter = Counter(0) // Error: 'val' cannot be reassigned
}
```

You can use a trailing comma when declaring function parameters:

```
fun powerOf(
    number: Int,
    exponent: Int, // trailing comma
) { /*...*/ }
```

Trailing commas help with refactorings and code maintenance: you can move parameters within the declaration without worrying about which is going to be the last one.

Kotlin functions can receive other functions as parameters — and be passed as arguments. For more information, see Higher-order functions and lambdas.

Parameters with default values

You can make a function parameter optional by specifying a default value for it. Kotlin uses the default value when you call the function without providing an argument that corresponds to that parameter. Parameters with default values are also known as *optional parameters*.

Optional parameters reduce the need for multiple overloads, since you don't have to declare different versions of a function just to allow skipping a parameter with a reasonable default.

Set a default value by appending `=` to the parameter declaration:

```
fun read(
    b: ByteArray,
    // The default value of 'off' is 0
    off: Int = 0,
    // The default value of 'len' is calculated
    // as the size of the 'b' array
    len: Int = b.size,
) { /*...*/ }
```

When you declare a parameter **with** a default value before a parameter **without** a default value, you can only use the default value by naming the argument:

```
fun greeting(
    userId: Int = 0,
    message: String,
) { /*...*/ }

fun main() {
    // Uses 0 as the default value for 'userId'
    greeting(message = "Hello!")

    // Error: No value passed for parameter 'message'
    greeting("Hello!")
}
```

Trailing lambdas are an exception to this rule, since the last parameter must correspond to the passed function:

```
fun main () {
//sampleStart
fun greeting(
    userId: Int = 0,
    message: () -> Unit,
)
{ println(userId)
  message() }

// Uses the default value for 'userId'
greeting() { println ("Hello!") }
// 0
// Hello!
}
```

Overriding methods always use the base method's default parameter values. When you override a method that has default parameter values, you must omit the default parameter values from the signature:

```
open class Shape {
    open fun draw(width: Int = 10, height: Int = 5) { /*...*/ }
}

class Rectangle : Shape() {
    // It's not allowed to specify default values here
    // but this function also uses 10 for 'width' and 5 for 'height'
    // by default.
    override fun draw(width: Int, height: Int) { /*...*/ }
}
```

Non-constant expressions as default values

You can assign a parameter a default value that isn't constant. For example, the default can be the result of a function call or a calculation that uses the values of other arguments, like the `len` parameter in this example:

```
fun read(
    b: ByteArray,
    off: Int = 0,
    len: Int = b.size,
) { /*...*/ }
```

Parameters that refer to the values of other parameters must be declared later in the order. In this example, `len` must be declared after `b`.

In general, you can assign any expression as the default value of a parameter. However, default values are only evaluated when the function is called **without** the corresponding parameter and a default value needs to be assigned. For example, this function prints out a line only when it is called without the `print` parameter:

```
fun main() {
    fun read(
        b: Int,
        print: Unit? = println("No argument passed for 'print'")
    ) { println(b) }

    // Prints "No argument passed for 'print'", then "1"
    read(1)
    // Prints only "1"
    read(1, null)
    //sampleEnd
}
```

If the last parameter in a function declaration has a functional type, you can pass the corresponding lambda argument either as a named argument or outside the parentheses:

```
fun main() {
        fun log(
        level: Int = 0,
        code:  Int = 1,
        action: () -> Unit,
    ) { println (level)
        println (code)
        action() }

    // Passes 1 for 'level' and uses the default value 1 for 'code'
    log(1) { println("Connection established") }

    // Uses both default values, 0 for 'level' and 1 for 'code'
    log(action = { println("Connection established") })

    // Equivalent to the previous call, uses both default values
    log { println("Connection established") }
    //sampleEnd
}
```

Named arguments

You can name one or more of a function's arguments when calling it. This can be helpful when a function call has many arguments. In such cases, it's difficult to associate a value with an argument, especially if it's `null` or a boolean value.

When you use named arguments in a function call, you can list them in any order.

Consider the `reformat()` function, which has 4 arguments with default values:

```
fun reformat(
    str: String,
    normalizeCase: Boolean = true,
    upperCaseFirstLetter: Boolean = true,
    divideByCamelHumps: Boolean = false,
    wordSeparator: Char = ' ',
) { /*...*/ }
```

When calling this function, you can name some of the arguments:

```
reformat(
    "String!",
    normalizeCase = false,
    upperCaseFirstLetter = false,
    divideByCamelHumps = true,
    '_'
)
```

You can skip all the arguments with default values:

```
reformat("This is a long String!")
```

You can also skip *some* arguments with default values, rather than omitting them all. However, after the first skipped argument, you must name all subsequent arguments:

```
reformat(
    "This is a short String!",
    upperCaseFirstLetter = false,
    wordSeparator = '_'
)
```

You can pass a variable number of arguments (`vararg`) by naming the corresponding argument. In this example, it's an array:

```
fun mergeStrings(vararg strings: String) { /*...*/ }

mergeStrings(strings = arrayOf("a", "b", "c"))
```

When calling Java functions on the JVM, you can't use the named argument syntax because Java bytecode does not always preserve the names of function parameters.

Return types

When you declare a function with a block body (by putting instructions within curly braces `{}`), you must always specify a return type explicitly. The only exception is when they return `Unit` , in which case specifying the return type is optional.

Kotlin doesn't infer return types for functions with block bodies. Their control flow can be complex, which makes the return type unclear to the reader and sometimes even to the compiler. However, Kotlin can infer the return type for single-expression functions if you don't specify it.

Single-expression functions

When the function body consists of a single expression, you can omit the curly braces and specify the body after an `=` symbol:

```
fun double(x: Int): Int = x * 2
```

Most of the time you don't have to explicitly declare the return type:

```
// Compiler infers that the function returns Int
fun double(x: Int) = x * 2
```

The compiler can sometimes run into problems when inferring return types from single expressions. In such cases, you should add the return type explicitly. For example, functions that are recursive or mutually recursive (calling each other) and functions with typeless expressions like `fun empty() = null` always require a return type.

When you do use an inferred return type, make sure to check the actual result because the compiler may infer a type that is less useful to you. In the example above, if you want the `double()` function to return `Number` instead of `Int`, you have to declare this explicitly.

If you use a `return` statement inside an expression body, you must specify the return type explicitly:

```
fun getDisplayNameOrDefault(userId: String?): String =
    getDisplayName(userId ?: return "default")
```

Unit-returning functions

If a function has a block body (instructions within curly braces `{}`) and does not return a useful value, the compiler assumes its return type is `Unit`. `Unit` is a type that has only one value, also called `Unit`.

You don't have to specify `Unit` as a return type, except for functional type parameters. You never have to return `Unit` explicitly.

For example, you can declare a `printHello()` function without returning `Unit`:

```
// The declaration of the functional type parameter ('action') still
// needs an explicit return type
fun printHello(name: String?, action: () -> Unit) {
    if (name != null)
        println("Hello $name")
    else
        println("Hi there!")

    action()
}

fun main() {
    printHello("Kodee") {
        println("This runs after the greeting.")
    }
    // Hello Kodee
    // This runs after the greeting.

    printHello(null) {
        println("No name provided, but action still runs.")
    }
    // No name provided, but action still runs
}
```

Which is equivalent to this verbose declaration:

```
fun printHello(name: String?, action: () -> Unit): Unit {
    if (name != null)
        println("Hello $name")
    else
        println("Hi there!")

    action()
    return Unit
}
fun main() {
    printHello("Kodee") {
        println("This runs after the greeting.")
    }
    // Hello Kodee
    // This runs after the greeting.

    printHello(null) {
        println("No name provided, but action still runs.")
    }
    // No name provided, but action still runs
}
```

Variable number of arguments (varargs)

To pass a variable number of arguments to a function, you can mark one of its parameters (usually the last one) with the `vararg` modifier. Inside a function, you can use a `vararg` -parameter of type `T` as an array of `T` :

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

Then you can pass a variable number of arguments to the function:

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}

fun main() {
        val list = asList(1, 2, 3)
    println(list)
    // [1, 2, 3]
    //sampleEnd
}
```

Only one parameter can be marked as `vararg` . If you declare a `vararg` parameter anywhere other than last in the parameter list, you must pass values for the following parameters using named arguments. If a parameter has a function type, you can also pass its value by placing a lambda outside the parentheses.

When you call a `vararg` -function, you can pass arguments individually, as in the example of `asList(1, 2, 3)` . If you already have an array and want to pass its contents to a function as a `vararg` parameter or as a part of it, use the spread operator by prefixing the array name with `*` :

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts)
        result.add(t)
    return result
}

fun main() {
        val a = arrayOf(1, 2, 3)

    // The function receives the array [-1, 0, 1, 2, 3, 4]
    list = asList(-1, 0, *a, 4)

    println(list)
    // [-1, 0, 1, 2, 3, 4]
    //sampleEnd
}
```

If you want to pass a primitive type array as `vararg`, you need to convert it to a regular (typed) array using the `.toTypedArray()` function:

```
// 'a' is an IntArray, which is a primitive type array
val a = intArrayOf(1, 2, 3)
val list = asList(-1, 0, *a.toTypedArray(), 4)
```

Infix notation

You can declare functions that can be called without parentheses or the period by using the `infix` keyword. This can help make simple function calls in your code easier to read.

```
infix fun Int.shl(x: Int): Int { /*...*/ }

// Calls the function using the general notation
1.shl(2)

// Calls the function using the infix notation
1 shl 2
```

Infix functions must meet the following requirements:

- They must be member functions of a class or extension functions.
- They must have a single parameter.
- The parameter must not accept a variable number of arguments (`vararg`) and must have no default value.

Infix function calls have lower precedence than arithmetic operators, type casts, and the `rangeTo` operator. The following expressions are equivalent:

- `1 shl 2 + 3` is equivalent to `1 shl (2 + 3)`
- `0 until n * 2` is equivalent to `0 until (n * 2)`
- `xs union ys as Set<*>` is equivalent to `xs union (ys as Set<*>)`

On the other hand, an infix function call's precedence is higher than that of the boolean operators `&&` and `||`, `is` - and `in` -checks, and some other operators. These expressions are equivalent as well:

- `a && b xor c` is equivalent to `a && (b xor c)`
- `a xor b in c` is equivalent to `(a xor b) in c`

Note that infix functions always require both the receiver and the parameter to be specified. When you call a method on the current receiver using the infix notation, use `this` explicitly. This ensures unambiguous parsing.

```
class MyStringCollection {
    val items = mutableListOf<String>()

    infix fun add(s: String) {
        println("Adding: $s")
        items += s
    }

    fun build() {
        add("first")      // Correct: ordinary function call
        this add "second" // Correct: infix call with an explicit receiver
        // add "third"    // Compiler error: needs an explicit receiver
    }

    fun printAll() = println("Items = $items")
}

fun main() {
    val myStrings = MyStringCollection()
    // Adds "first" and "second" to the list
    myStrings.build()

    myStrings.printAll()
    // Adding: first
    // Adding: second
    // Items = [first, second]
}
```

Function scope

You can declare Kotlin functions at the top level in a file, meaning you do not need to create a class to hold a function. Functions can also be declared locally as *member functions* or *extension functions*.

Local functions

Kotlin supports local functions, which are functions declared inside other functions. For example, the following code implements the Depth-first search algorithm for a given graph. The local `dfs()` function inside the outer `dfs()` function to hide the implementation and handle recursive calls:

```
class Person(val name: String) {
    val friends = mutableListOf<Person>()
}
class SocialGraph(val people: List<Person>)
fun dfs(graph: SocialGraph) {
    fun dfs(current: Person, visited: MutableSet<Person>) {
        if (!visited.add(current)) return
        println("Visited ${current.name}")
        for (friend in current.friends)
            dfs(friend, visited)
    }
    dfs(graph.people[0], HashSet())
}
fun main() {
    val alice = Person("Alice")
    val bob = Person("Bob")
    val charlie = Person("Charlie")
    alice.friends += bob
    bob.friends += charlie
    charlie.friends += alice
    val network = SocialGraph(listOf(alice, bob, charlie))
    dfs(network)
}
```

A local function can access local variables of outer functions (the closure). In the case above, the `visited` function parameter can be a local variable:

```
class Person(val name: String) {
    val friends = mutableListOf<Person>()
}
class SocialGraph(val people: List<Person>)
fun dfs(graph: SocialGraph) {
    val visited = HashSet<Person>()
    fun dfs(current: Person) {
        if (!visited.add(current)) return
        println("Visited ${current.name}")
        for (friend in current.friends)
            dfs(friend)
    }
    dfs(graph.people[0])
}
fun main() {
    val alice = Person("Alice")
    val bob = Person("Bob")
    val charlie = Person("Charlie")
    alice.friends += bob
    bob.friends += charlie
    charlie.friends += alice
    val network = SocialGraph(listOf(alice, bob, charlie))
    dfs(network)
}
```

Member functions

A member function is a function that is defined inside a class or object:

```
class Sample {
    fun foo() { print("Foo") }
}
```

To call member functions, write the instance or object name, then add a `.` and write the function name:

```
// Creates an instance of the Stream class and calls read()
Stream().read()
```

For more information on classes and overriding members see Classes and Inheritance.

Generic functions

You can specify generic parameters for a function by using angle brackets `<>` before the function name:

```
fun <T> singletonList(item: T): List<T> { /*...*/ }
```

For more information on generic functions, see Generics.

Tail recursive functions

Kotlin supports a style of functional programming known as tail recursion. For some algorithms that would normally use loops, you can use a recursive function instead without the risk of stack overflow. When a function is marked with the `tailrec` modifier and meets the required formal conditions, the compiler optimizes out the recursion, leaving behind a fast and efficient loop based version instead:

```
import kotlin.math.cos
import kotlin.math.abs

// An arbitrary "good enough" precision
val eps = 1E-10

tailrec fun findFixPoint(x: Double = 1.0): Double =
    if (abs(x - cos(x)) < eps) x else findFixPoint(cos(x))
```

This code calculates the fixed point of cosine (a mathematical constant). The function calls `cos()` repeatedly starting at `1.0` until the result no longer changes, yielding a result of `0.7390851332151611` for the specified `eps` precision. The code is equivalent to this more traditional style:

```
import kotlin.math.cos
import kotlin.math.abs

// An arbitrary "good enough" precision
val eps = 1E-10

private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = cos(x)
        if (abs(x - y) < eps) return x
        x = cos(x)
    }
}
```

You can apply the `tailrec` modifier to a function only when it calls itself as its final operation. You cannot use tail recursion when there is more code after the recursive call, within `try` / `catch` / `finally` blocks, or when the function is open.

See also:

- Inline functions
- Extension functions
- Higher-order functions and lambdas

Higher-order functions and lambdas

Kotlin functions are first-class, which means they can be stored in variables and data structures, and can be passed as arguments to and returned from other higher-order functions. You can perform any operations on functions that are possible for other non-function values.

To facilitate this, Kotlin, as a statically typed programming language, uses a family of function types to represent functions, and provides a set of specialized language constructs, such as lambda expressions.

Higher-order functions

A higher-order function is a function that takes functions as parameters, or returns a function.

A good example of a higher-order function is the functional programming idiom `fold` for collections. It takes an initial accumulator value and a combining function and builds its return value by consecutively combining the current accumulator value with each collection element, replacing the accumulator value each time:

```
fun <T, R> Collection<T>.fold(
    initial: R,
    combine: (acc: R, nextElement: T) -> R
): R {
    var accumulator: R = initial
    for (element: T in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}
```

In the code above, the `combine` parameter has the function type `(R, T) -> R`, so it accepts a function that takes two arguments of types `R` and `T` and returns a value of type `R`. It is invoked inside the `for` loop, and the return value is then assigned to `accumulator`.

To call `fold`, you need to pass an instance of the function type to it as an argument, and lambda expressions (described in more detail below) are widely used for this purpose at higher-order function call sites:

```
fun main() {
        val items = listOf(1, 2, 3, 4, 5)

    // Lambdas are code blocks enclosed in curly braces.
    items.fold(0, {
        // When a lambda has parameters, they go first, followed by '->'
        acc: Int, i: Int ->
        print("acc = $acc, i = $i, ")
        val result = acc + i
        println("result = $result")
        // The last expression in a lambda is considered the return value:
        result
    })

    // Parameter types in a lambda are optional if they can be inferred:
    val joinedToString = items.fold("Elements:", { acc, i -> acc + " " + i })

    // Function references can also be used for higher-order function calls:
    val product = items.fold(1, Int::times)
    //sampleEnd
    println("joinedToString = $joinedToString")
    println("product = $product")
}
```

Function types

Kotlin uses function types, such as `(Int) -> String`, for declarations that deal with functions: `val onClick: () -> Unit = ...`.

These types have a special notation that corresponds to the signatures of the functions - their parameters and return values:

- All function types have a parenthesized list of parameter types and a return type: `(A, B) -> C` denotes a type that represents functions that take two arguments of types `A` and `B` and return a value of type `C`. The list of parameter types may be empty, as in `() -> A`. The `Unit` return type cannot be omitted.
- Function types can optionally have an additional *receiver* type, which is specified before the dot in the notation: the type `A.(B) -> C` represents functions that can be called on a receiver object `A` with a parameter `B` and return a value `C`. Function literals with receiver are often used along with these types.
- Suspending functions belong to a special kind of function type that have a *suspend* modifier in their notation, such as `suspend () -> Unit` or `suspend A.(B) -> C`.

The function type notation can optionally include names for the function parameters: `(x: Int, y: Int) -> Point`. These names can be used for documenting the meaning of the parameters.

To specify that a function type is nullable, use parentheses as follows: `((Int, Int) -> Int)?`.

Function types can also be combined using parentheses: `(Int) -> ((Int) -> Unit)`.

> The arrow notation is right-associative, `(Int) -> (Int) -> Unit` is equivalent to the previous example, but not to `((Int) -> (Int)) -> Unit`.

You can also give a function type an alternative name by using a type alias:

```
typealias ClickHandler = (Button, ClickEvent) -> Unit
```

Instantiating a function type

There are several ways to obtain an instance of a function type:

- Use a code block within a function literal, in one of the following forms:
  - a lambda expression: `{ a, b -> a + b }`,
  - an anonymous function: `fun(s: String): Int { return s.toIntOrNull() ?: 0 }`

  Function literals with receiver can be used as values of function types with receiver.
- Use a callable reference to an existing declaration:
  - a top-level, local, member, or extension function: `::isOdd`, `String::toInt`,
  - a top-level, member, or extension property: `List<Int>::size`,
  - a constructor: `::Regex`

  These include bound callable references that point to a member of a particular instance: `foo::toString`.
- Use instances of a custom class that implements a function type as an interface:

```
class IntTransformer: (Int) -> Int {
    override operator fun invoke(x: Int): Int = TODO()
}

val intFunction: (Int) -> Int = IntTransformer()
```

The compiler can infer the function types for variables if there is enough information:

```
val a = { i: Int -> i + 1 } // The inferred type is (Int) -> Int
```

Non-literal values of function types with and without a receiver are interchangeable, so the receiver can stand in for the first parameter, and vice versa. For instance, a value of type `(A, B) -> C` can be passed or assigned where a value of type `A.(B) -> C` is expected, and the other way around:

```
fun main() {
        val repeatFun: String.(Int) -> String = { times -> this.repeat(times) }
    val twoParameters: (String, Int) -> String = repeatFun // OK

    fun runTransformation(f: (String, Int) -> String): String {
        return f("hello", 3)
    }
    val result = runTransformation(repeatFun) // OK
    //sampleEnd
    println("result = $result")
}
```

A function type with no receiver is inferred by default, even if a variable is initialized with a reference to an extension function. To alter that, specify the variable type explicitly.

Invoking a function type instance

A value of a function type can be invoked by using its `invoke(...)` operator: `f.invoke(x)` or just `f(x)`.

If the value has a receiver type, the receiver object should be passed as the first argument. Another way to invoke a value of a function type with receiver is to prepend it with the receiver object, as if the value were an extension function: `1.foo(2)`.

Example:

```
fun main() {
        val stringPlus: (String, String) → String = String::plus
    val intPlus: Int.(Int) → Int = Int::plus

    println(stringPlus.invoke("←", "→"))
    println(stringPlus("Hello, ", "world!"))

    println(intPlus.invoke(1, 1))
    println(intPlus(1, 2))
    println(2.intPlus(3)) // extension-like call
    //sampleEnd
}
```

Inline functions

Sometimes it is beneficial to use inline functions, which provide flexible control flow, for higher-order functions.

Lambda expressions and anonymous functions

Lambda expressions and anonymous functions are *function literals*. Function literals are functions that are not declared but are passed immediately as an expression. Consider the following example:

```
max(strings, { a, b → a.length < b.length })
```

The function `max` is a higher-order function, as it takes a function value as its second argument. This second argument is an expression that is itself a function, called a function literal, which is equivalent to the following named function:

```
fun compare(a: String, b: String): Boolean = a.length < b.length
```

You can also create a *suspending lambda expression* using the `suspend` keyword. A suspending lambda has the function type `suspend () -> Unit` and can call other suspending functions:

```
val suspendingTask = suspend { doSuspendingWork() }
```

Lambda expression syntax

The full syntactic form of lambda expressions is as follows:

```
val sum: (Int, Int) → Int = { x: Int, y: Int → x + y }
```

- A lambda expression is always surrounded by curly braces.
- Parameter declarations in the full syntactic form go inside curly braces and have optional type annotations.
- The body goes after the `->`.

- If the inferred return type of the lambda is not `Unit`, the last (or possibly single) expression inside the lambda body is treated as the return value.

If you leave all the optional annotations out, what's left looks like this:

```
val sum = { x: Int, y: Int -> x + y }
```

Passing trailing lambdas

According to Kotlin convention, if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses:

```
val product = items.fold(1) { acc, e -> acc * e }
```

Such syntax is also known as *trailing lambda*.

If the lambda is the only argument in that call, the parentheses can be omitted entirely:

```
run { println("...") }
```

it: implicit name of a single parameter

It's very common for a lambda expression to have only one parameter.

If the compiler can parse the signature without any parameters, the parameter does not need to be declared and `->` can be omitted. The parameter will be implicitly declared under the name `it`:

```
ints.filter { it > 0 } // this literal is of type '(it: Int) -> Boolean'
```

Returning a value from a lambda expression

You can explicitly return a value from the lambda using the qualified return syntax. Otherwise, the value of the last expression is implicitly returned.

Therefore, the two following snippets are equivalent:

```
ints.filter {
    val shouldFilter = it > 0
    shouldFilter
}

ints.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}
```

This convention, along with passing a lambda expression outside of parentheses, allows for LINQ-style code:

```
strings.filter { it.length == 5 }.sortedBy { it }.map { it.uppercase() }
```

Underscore for unused variables

If the lambda parameter is unused, you can place an underscore instead of its name:

```
map.forEach { (_, value) -> println("$value!") }
```

Destructuring in lambdas

Destructuring in lambdas is described as a part of destructuring declarations.

Anonymous functions

The lambda expression syntax above is missing one thing – the ability to specify the function's return type. In most cases, this is unnecessary because the return type can be inferred automatically. However, if you do need to specify it explicitly, you can use an alternative syntax: an *anonymous function*.

```
fun(x: Int, y: Int): Int = x + y
```

An anonymous function looks very much like a regular function declaration, except its name is omitted. Its body can be either an expression (as shown above) or a block:

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

The parameters and the return type are specified in the same way as for regular functions, except the parameter types can be omitted if they can be inferred from the context:

```
ints.filter(fun(item) = item > 0)
```

The return type inference for anonymous functions works just like for normal functions: the return type is inferred automatically for anonymous functions with an expression body, but it has to be specified explicitly (or is assumed to be `Unit`) for anonymous functions with a block body.

> When passing anonymous functions as parameters, place them inside the parentheses. The shorthand syntax that allows you to leave the function outside the parentheses works only for lambda expressions.

Another difference between lambda expressions and anonymous functions is the behavior of non-local returns. A `return` statement without a label always returns from the function declared with the `fun` keyword. This means that a `return` inside a lambda expression will return from the enclosing function, whereas a `return` inside an anonymous function will return from the anonymous function itself.

Closures

A lambda expression or anonymous function (as well as a local function and an object expression) can access its *closure*, which includes the variables declared in the outer scope. The variables captured in the closure can be modified in the lambda:

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
}
print(sum)
```

Function literals with receiver

Function types with receiver, such as `A.(B) -> C`, can be instantiated with a special form of function literals – function literals with receiver.

As mentioned above, Kotlin provides the ability to call an instance of a function type with receiver while providing the *receiver object*.

Inside the body of the function literal, the receiver object passed to a call becomes an *implicit* `this`, so that you can access the members of that receiver object without any additional qualifiers, or access the receiver object using a `this` expression.

This behavior is similar to that of extension functions, which also allow you to access the members of the receiver object inside the function body.

Here is an example of a function literal with receiver along with its type, where `plus` is called on the receiver object:

```
val sum: Int.(Int) -> Int = { other -> plus(other) }
```

The anonymous function syntax allows you to specify the receiver type of a function literal directly. This can be useful if you need to declare a variable of a function type with receiver, and then to use it later.

```
val sum = fun Int.(other: Int): Int = this + other
```

Lambda expressions can be used as function literals with receiver when the receiver type can be inferred from the context. One of the most important examples of their usage is type-safe builders:

```
class HTML {
    fun body() { ... }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()  // create the receiver object
    html.init()        // pass the receiver object to the lambda
    return html
}

html {       // lambda with receiver begins here
    body()   // calling a method on the receiver object
}
```

Type-safe builders

By using well-named functions as builders in combination with function literals with receiver, it is possible to create type-safe, statically-typed builders in Kotlin.

Type-safe builders allow creating Kotlin-based domain-specific languages (DSLs) suitable for building complex hierarchical data structures in a semi-declarative way. Sample use cases for the builders are:

- Generating markup with Kotlin code, such as HTML or XML
- Configuring routes for a web server: Ktor

Consider the following code:

```
package html

fun main() {
        val result = html {
        head {
            title { +"HTML encoding with Kotlin" }
        }
        body {
            h1 { +"HTML encoding with Kotlin" }
            p {
                +"this format can be used as an"
                +"alternative markup to HTML"
            }

            // An element with attributes and text content
            a(href = "http://kotlinlang.org") { +"Kotlin" }

            // Mixed content
            p {
                +"This is some"
                b { +"mixed" }
                +"text. For more see the"
                a(href = "http://kotlinlang.org") {
                    +"Kotlin"
                }
                +"project"
            }
            p {
                +"some text"
                ul {
                    for (i in 1..5)
                        li { +"${i}*2 = ${i*2}" }
                }
            }
        }
    }
    //sampleEnd
    println(result)
}

interface Element {
    fun render(builder: StringBuilder, indent: String)
}

class TextElement(val text: String) : Element {
    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent$text\n")
    }
}

@DslMarker
annotation class HtmlTagMarker

@HtmlTagMarker
abstract class Tag(val name: String) : Element {
    val children = arrayListOf<Element>()
    val attributes = hashMapOf<String, String>()

    protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
        tag.init()
        children.add(tag)
        return tag
    }
```

```
    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent<$name${renderAttributes()}>\n")
        for (c in children) {
            c.render(builder, indent + "  ")
        }
        builder.append("$indent</$name>\n")
    }

    private fun renderAttributes(): String {
        val builder = StringBuilder()
        for ((attr, value) in attributes) {
            builder.append(" $attr=\"$value\"")
        }
        return builder.toString()
    }

    override fun toString(): String {
        val builder = StringBuilder()
        render(builder, "")
        return builder.toString()
    }
}

abstract class TagWithText(name: String) : Tag(name) {
    operator fun String.unaryPlus() {
        children.add(TextElement(this))
    }
}
class HTML() : TagWithText("html") {
    fun head(init: Head.() -> Unit) = initTag(Head(), init)
    fun body(init: Body.() -> Unit) = initTag(Body(), init)
}


class Head() : TagWithText("head") {
    fun title(init: Title.() -> Unit) = initTag(Title(), init)
}

class Title() : TagWithText("title")

abstract class BodyTag(name: String) : TagWithText(name) {
    fun b(init: B.() -> Unit) = initTag(B(), init)
    fun p(init: P.() -> Unit) = initTag(P(), init)
    fun h1(init: H1.() -> Unit) = initTag(H1(), init)
    fun ul(init: UL.() -> Unit) = initTag(UL(), init)
    fun a(href: String, init: A.() -> Unit) {
        val a = initTag(A(), init)
        a.href = href
    }
}

class Body() : BodyTag("body")
class UL() : BodyTag("ul") {
    fun li(init: LI.() -> Unit) = initTag(LI(), init)
}

class B() : BodyTag("b")
class LI() : BodyTag("li")
class P() : BodyTag("p")
class H1() : BodyTag("h1")

class A : BodyTag("a") {
    var href: String
        get() = attributes["href"]!!
```

```
        set(value) {
            attributes["href"] = value
        }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

```
<html>
  <head>
    <title>
      HTML encoding with Kotlin
    </title>
  </head>
  <body>
    <h1>
      HTML encoding with Kotlin
    </h1>
    <p>
      this format can be used as an
      alternative markup to HTML
    </p>
    <a href="http://kotlinlang.org">
      Kotlin
    </a>
    <p>
      This is some
      <b>
        mixed
      </b>
      text. For more see the
      <a href="http://kotlinlang.org">
        Kotlin
      </a>
      project
    </p>
    <p>
      some text
      <ul>
        <li>
          1*2 = 2
        </li>
        <li>
          2*2 = 4
        </li>
        <li>
          3*2 = 6
        </li>
        <li>
          4*2 = 8
        </li>
        <li>
          5*2 = 10
        </li>
      </ul>
    </p>
  </body>
</html>
```

How it works

Assume that you need to implement a type-safe builder in Kotlin. First of all, define the model you want to build. In this case, you need to model HTML tags. It is easily done with a bunch of classes. For example, `HTML` is a class that describes the `<html>` tag defining children like `<head>` and `<body>`. (See its declaration below.)

Now, let's recall why you can say something like this in the code:

```
html {
 // ...
}
```

`html` is actually a function call that takes a lambda expression as an argument. This function is defined as follows:

```
fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

This function takes one parameter named `init`, which is itself a function. The type of the function is `HTML.() -> Unit`, which is a *function type with receiver*. This means that you need to pass an instance of type `HTML` (a *receiver*) to the function, and you can call members of that instance inside the function.

The receiver can be accessed through the `this` keyword:

```
html {
    this.head { ... }
    this.body { ... }
}
```

(`head` and `body` are member functions of `HTML`.)

Now, `this` can be omitted, as usual, and you get something that looks very much like a builder already:

```
html {
    head { ... }
    body { ... }
}
```

So, what does this call do? Let's look at the body of `html` function as defined above. It creates a new instance of `HTML`, then it initializes it by calling the function that is passed as an argument (in this example, this boils down to calling `head` and `body` on the `HTML` instance), and then it returns this instance. This is exactly what a builder should do.

The `head` and `body` functions in the `HTML` class are defined similarly to `html`. The only difference is that they add the built instances to the `children` collection of the enclosing `HTML` instance:

```
fun head(init: Head.() -> Unit): Head {
    val head = Head()
    head.init()
    children.add(head)
    return head
}

fun body(init: Body.() -> Unit): Body {
    val body = Body()
    body.init()
    children.add(body)
    return body
}
```

Actually, these two functions do just the same thing, so you can have a generic version, `initTag`:

```
protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
    tag.init()
    children.add(tag)
    return tag
}
```

So, now your functions are very simple:

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)

fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

And you can use them to build `<head>` and `<body>` tags.

One other thing to be discussed here is how you add text to tag bodies. In the example above, you say something like:

```
html {
    head {
        title {+"XML encoding with Kotlin"}
    }
    // ...
}
```

So basically, you just put a string inside a tag body, but there is this little `+` in front of it, so it is a function call that invokes a prefix `unaryPlus()` operation. That operation is actually defined by an extension function `unaryPlus()` that is a member of the `TagWithText` abstract class (a parent of `Title`):

```
operator fun String.unaryPlus() {
    children.add(TextElement(this))
}
```

So, what the prefix `+` does here is wrapping a string into an instance of `TextElement` and adding it to the `children` collection, so that it becomes a proper part of the tag tree.

All this is defined in a package `com.example.html` that is imported at the top of the builder example above. In the last section, you can read through the full definition of this package.

Scope control: @DslMarker

When using DSLs, one might have come across the problem that too many functions can be called in the context. You can call methods of every available implicit receiver inside a lambda and therefore get an inconsistent result, like the tag `head` inside another `head`:

```
html {
    head {
        head {} // should be forbidden
    }
    // ...
}
```

In this example, only members of the nearest implicit receiver `this@head` must be available; `head()` is a member of the outer receiver `this@html`, so it must be illegal to call it.

To address this problem, there is a special mechanism to control receiver scope.

To make the compiler start controlling scopes, you only have to annotate the types of all receivers used in the DSL with the same marker annotation. For instance, for HTML Builders you declare an annotation `@HtmlTagMarker`:

```
@DslMarker
@Target(AnnotationTarget.CLASS)
annotation class HtmlTagMarker
```

An annotation class is called a DSL marker if it is annotated with the `@DslMarker` annotation.

The `@Target` annotation restricts where `@HtmlTagMarker` can be applied. DSL markers only affect scope control when applied to:

- Type declarations (`CLASS`): classes or interfaces used as DSL receivers.
- Type usages (`TYPE`): receiver types in function type signatures.
- Type aliases (`TYPEALIAS`): type aliases that expand to DSL receiver types.

Applying a DSL marker to other targets (such as functions or properties) has no effect on scope control.

> For more details on how DSL marker works, see the corresponding KEEP document.

In our DSL, all the tag classes extend the same superclass `Tag`. It's enough to annotate only the superclass with `@HtmlTagMarker` and after that the Kotlin compiler will treat all the inherited classes as annotated:

```
@HtmlTagMarker
abstract class Tag(val name: String) { ... }
```

You don't have to annotate the `HTML` or `Head` classes with `@HtmlTagMarker` because their superclass is already annotated:

```
class HTML() : Tag("html") { ... }

class Head() : Tag("head") { ... }
```

After you've added this annotation, the Kotlin compiler knows which implicit receivers are part of the same DSL and allows to call members of the nearest receivers only:

```
html {
    head {
        head { } // error: a member of outer receiver
    }
    // ...
}
```

Note that it's still possible to call the members of the outer receiver, but to do that you have to specify this receiver explicitly:

```
html {
    head {
        this@html.head { } // possible
    }
    // ...
}
```

You can also apply the `@DslMarker` annotation directly to function types. This requires including `AnnotationTarget.TYPE` in the annotation targets:

```
@DslMarker
@Target(AnnotationTarget.CLASS, AnnotationTarget.TYPE)
annotation class HtmlTagMarker
```

As a result, the `@DslMarker` annotation can be applied to function types, most commonly to lambdas with receivers. For example:

```
fun html(init: @HtmlTagMarker HTML.() -> Unit): HTML { ... }

fun HTML.head(init: @HtmlTagMarker Head.() -> Unit): Head { ... }

fun Head.title(init: @HtmlTagMarker Title.() -> Unit): Title { ... }
```

When you call these functions, the `@DslMarker` annotation restricts access to outer receivers in the body of a lambda marked with it unless you specify them explicitly:

```
html {
    head {
        title {
            // Access to title, head or other functions of outer receivers is restricted here.
        }
    }
}
```

Only the nearest receiver's members and extensions are accessible within a lambda, preventing unintended interactions between nested scopes.

When both a member of an implicit receiver and a declaration from a context parameter are in a scope with the same name, the compiler reports a warning because the implicit receiver is shadowed by the context parameter. To resolve this, use a `this` qualifier to explicitly call the receiver, or use `contextOf<T>()` to call the context declaration:

```
interface HtmlTag {
    fun setAttribute(name: String, value: String)
}

// Declares a top-level function with the same name,
// which is available through a context parameter
context(tag: HtmlTag)
fun setAttribute(name: String, value: String) { tag.setAttribute(name, value) }

fun test(head: HtmlTag, extraInfo: HtmlTag) {
    with(head) {
        // Introduces a context value of the same type in an inner scope
        context(extraInfo) {
            // Reports a warning:
            // Uses an implicit receiver shadowed by a context parameter
            setAttribute("user", "1234")

            // Calls the receiver's member explicitly
            this.setAttribute("user", "1234")

            // Calls the context declaration explicitly
            contextOf<HtmlTag>().setAttribute("user", "1234")
        }
    }
}
```

Full definition of the com.example.html package

This is how the package `com.example.html` is defined (only the elements used in the example above). It builds an HTML tree. It makes heavy use of extension functions and lambdas with receiver.

```
package com.example.html

interface Element {
    fun render(builder: StringBuilder, indent: String)
}

class TextElement(val text: String) : Element {
    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent$text\n")
    }
}

@DslMarker
@Target(AnnotationTarget.CLASS, AnnotationTarget.TYPE)
annotation class HtmlTagMarker

@HtmlTagMarker
abstract class Tag(val name: String) : Element {
    val children = arrayListOf<Element>()
    val attributes = hashMapOf<String, String>()

    protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
        tag.init()
        children.add(tag)
        return tag
    }

    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent<$name${renderAttributes()}>\n")
        for (c in children) {
            c.render(builder, indent + "  ")
        }
        builder.append("$indent</$name>\n")
    }

    private fun renderAttributes(): String {
        val builder = StringBuilder()
        for ((attr, value) in attributes) {
            builder.append(" $attr=\"$value\"")
        }
        return builder.toString()
    }

    override fun toString(): String {
        val builder = StringBuilder()
        render(builder, "")
        return builder.toString()
    }
}

abstract class TagWithText(name: String) : Tag(name) {
    operator fun String.unaryPlus() {
        children.add(TextElement(this))
    }
}

class HTML : TagWithText("html") {
    fun head(init: Head.() -> Unit) = initTag(Head(), init)

    fun body(init: Body.() -> Unit) = initTag(Body(), init)
}

class Head : TagWithText("head") {
```

```
    fun title(init: Title.() -> Unit) = initTag(Title(), init)
}

class Title : TagWithText("title")

abstract class BodyTag(name: String) : TagWithText(name) {
    fun b(init: B.() -> Unit) = initTag(B(), init)
    fun p(init: P.() -> Unit) = initTag(P(), init)
    fun h1(init: H1.() -> Unit) = initTag(H1(), init)
    fun a(href: String, init: A.() -> Unit) {
        val a = initTag(A(), init)
        a.href = href
    }
}

class Body : BodyTag("body")
class B : BodyTag("b")
class P : BodyTag("p")
class H1 : BodyTag("h1")

class A : BodyTag("a") {
    var href: String
        get() = attributes["href"]!!
        set(value) {
            attributes["href"] = value
        }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

Using builders with builder type inference

Kotlin supports *builder type inference* (or builder inference), which can come in useful when you are working with generic builders. It helps the compiler infer the type arguments of a builder call based on the type information about other calls inside its lambda argument.

Consider this example of `buildMap()` usage:

```
fun addEntryToMap(baseMap: Map<String, Number>, additionalEntry: Pair<String, Int>?) {
   val myMap = buildMap {
       putAll(baseMap)
       if (additionalEntry != null) {
           put(additionalEntry.first, additionalEntry.second)
       }
   }
}
```

There is not enough type information here to infer type arguments in a regular way, but builder inference can analyze the calls inside the lambda argument. Based on the type information about `putAll()` and `put()` calls, the compiler can automatically infer type arguments of the `buildMap()` call into `String` and `Number`. Builder inference allows to omit type arguments while using generic builders.

Writing your own builders

Requirements for enabling builder inference

> Before Kotlin 1.7.0, enabling builder inference for a builder function required `-Xenable-builder-inference` compiler option. In 1.7.0 the option is enabled by default.

To let builder inference work for your own builder, make sure its declaration has a builder lambda parameter of a function type with a receiver. There are also two requirements for the receiver type:

1. It should use the type arguments that builder inference is supposed to infer. For example:

```
fun <V> buildList(builder: MutableList<V>.() -> Unit) { ... }
```

> Note that passing the type parameter's type directly like `fun <T> myBuilder(builder: T.() -> Unit)` is not yet supported.

2. It should provide public members or extensions that contain the corresponding type parameters in their signature. For example:

```
class ItemHolder<T> {
    private val items = mutableListOf<T>()

    fun addItem(x: T) {
        items.add(x)
    }

    fun getLastItem(): T? = items.lastOrNull()
}

fun <T> ItemHolder<T>.addAllItems(xs: List<T>) {
    xs.forEach { addItem(it) }
}

fun <T> itemHolderBuilder(builder: ItemHolder<T>.() -> Unit): ItemHolder<T> =
    ItemHolder<T>().apply(builder)

fun test(s: String) {
    val itemHolder1 = itemHolderBuilder { // Type of itemHolder1 is ItemHolder<String>
        addItem(s)
    }
    val itemHolder2 = itemHolderBuilder { // Type of itemHolder2 is ItemHolder<String>
        addAllItems(listOf(s))
    }
    val itemHolder3 = itemHolderBuilder { // Type of itemHolder3 is ItemHolder<String?>
        val lastItem: String? = getLastItem()
        // ...
    }
}
```

Supported features

Builder inference supports:

- Inferring several type arguments

```
fun <K, V> myBuilder(builder: MutableMap<K, V>.() -> Unit): Map<K, V> { ... }
```

- Inferring type arguments of several builder lambdas within one call including interdependent ones

```
fun <K, V> myBuilder(
    listBuilder: MutableList<V>.() -> Unit,
    mapBuilder: MutableMap<K, V>.() -> Unit
): Pair<List<V>, Map<K, V>> =
    mutableListOf<V>().apply(listBuilder) to mutableMapOf<K, V>().apply(mapBuilder)

fun main() {
    val result = myBuilder(
        { add(1) },
        { put("key", 2) }
    )
    // result has Pair<List<Int>, Map<String, Int>> type
}
```

- Inferring type arguments whose type parameters are lambda's parameter or return types

```
fun <K, V> myBuilder1(
    mapBuilder: MutableMap<K, V>.() -> K
): Map<K, V> = mutableMapOf<K, V>().apply { mapBuilder() }

fun <K, V> myBuilder2(
    mapBuilder: MutableMap<K, V>.(K) -> Unit
): Map<K, V> = mutableMapOf<K, V>().apply { mapBuilder(2 as K) }

fun main() {
    // result1 has the Map<Long, String> type inferred
    val result1 = myBuilder1 {
        put(1L, "value")
        2
    }
    val result2 = myBuilder2 {
        put(1, "value 1")
        // You can use `it` as "postponed type variable" type
        // See the details in the section below
        put(it, "value 2")
    }
}
```

How builder inference works

Postponed type variables

Builder inference works in terms of *postponed type variables*, which appear inside the builder lambda during builder inference analysis. A postponed type variable is a type argument's type, which is in the process of inferring. The compiler uses it to collect type information about the type argument.

Consider the example with `buildList()`:

```
val result = buildList {
    val x = get(0)
}
```

Here `x` has a type of postponed type variable: the `get()` call returns a value of type `E`, but `E` itself is not yet fixed. At this moment, a concrete type for `E` is unknown.

When a value of a postponed type variable gets associated with a concrete type, builder inference collects this information to infer the resulting type of the corresponding type argument at the end of the builder inference analysis. For example:

```
val result = buildList {
    val x = get(0)
    val y: String = x
} // result has the List<String> type inferred
```

After the postponed type variable gets assigned to a variable of the `String` type, builder inference gets the information that `x` is a subtype of `String`. This assignment is the last statement in the builder lambda, so the builder inference analysis ends with the result of inferring the type argument `E` into `String`.

Note that you can always call `equals()`, `hashCode()`, and `toString()` functions with a postponed type variable as a receiver.

Contributing to builder inference results

Builder inference can collect different varieties of type information that contribute to the analysis result. It considers:

- Calling methods on a lambda's receiver that use the type parameter's type

  ```
  val result = buildList {
      // Type argument is inferred into String based on the passed "value" argument
      add("value")
  } // result has the List<String> type inferred
  ```

- Specifying the expected type for calls that return the type parameter's type

  ```
  val result = buildList {
      // Type argument is inferred into Float based on the expected type
      val x: Float = get(0)
  } // result has the List<Float> type
  ```

  ```
  class Foo<T> {
      val items = mutableListOf<T>()
  }

  fun <K> myBuilder(builder: Foo<K>.() -> Unit): Foo<K> = Foo<K>().apply(builder)

  fun main() {
      val result = myBuilder {
          val x: List<CharSequence> = items
          // ...
      } // result has the Foo<CharSequence> type
  }
  ```

- Passing postponed type variables' types into methods that expect concrete types

```
fun takeMyLong(x: Long) { ... }

fun String.isMoreThan3() = length > 3

fun takeListOfStrings(x: List<String>) { ... }

fun main() {
    val result1 = buildList {
        val x = get(0)
        takeMyLong(x)
    } // result1 has the List<Long> type

    val result2 = buildList {
        val x = get(0)
        val isLong = x.isMoreThan3()
    // ...
    } // result2 has the List<String> type

    val result3 = buildList {
        takeListOfStrings(this)
    } // result3 has the List<String> type
}
```

- Taking a callable reference to the lambda receiver's member

```
fun main() {
    val result = buildList {
        val x: KFunction1<Int, Float> = ::get
    } // result has the List<Float> type
}
```

```
fun takeFunction(x: KFunction1<Int, Float>) { ... }

fun main() {
    val result = buildList {
        takeFunction(::get)
    } // result has the List<Float> type
}
```

At the end of the analysis, builder inference considers all collected type information and tries to merge it into the resulting type. See the example.

```
val result = buildList { // Inferring postponed type variable E
    // Considering E is Number or a subtype of Number
    val n: Number? = getOrNull(0)
    // Considering E is Int or a supertype of Int
    add(1)
    // E gets inferred into Int
} // result has the List<Int> type
```

The resulting type is the most specific type that corresponds to the type information collected during the analysis. If the given type information is contradictory and cannot be merged, the compiler reports an error.

Note that the Kotlin compiler uses builder inference only if regular type inference cannot infer a type argument. This means you can contribute type information outside a builder lambda, and then builder inference analysis is not required. Consider the example:

```
fun someMap() = mutableMapOf<CharSequence, String>()

fun <E> MutableMap<E, String>.f(x: MutableMap<E, String>) { ... }

fun main() {
    val x: Map<in String, String> = buildMap {
        put("", "")
        f(someMap()) // Type mismatch (required String, found CharSequence)
    }
}
```

Here a type mismatch appears because the expected type of the map is specified outside the builder lambda. The compiler analyzes all the statements inside with the fixed receiver type `Map<in String, String>`.

Context parameters

> Context parameters replace an older experimental feature called context receivers. You can find their main differences in the design document for context parameters. To migrate from context receivers to context parameters, you can use assisted support in IntelliJ IDEA, as described in the related blog post.

Context parameters allow functions and properties to declare dependencies that are implicitly available in the surrounding context.

With context parameters, you don't need to manually pass around values, such as services or dependencies, that are shared and rarely change across sets of function calls.

To declare context parameters for properties and functions, use the `context` keyword followed by a list of parameters, with each parameter declared as `name: Type`. Here is an example with a dependency on the `UserService` interface:

```
// UserService defines the dependency required in context
interface UserService {
    fun log(message: String)
    fun findUserById(id: Int): String
}

// Declares a function with a context parameter
context(users: UserService)
fun outputMessage(message: String) {
    // Uses log from the context
    users.log("Log: $message")
}

// Declares a property with a context parameter
context(users: UserService)
val firstUser: String
    // Uses findUserById from the context
    get() = users.findUserById(1)

fun main() {
    val users = object : UserService {
        override fun log(message: String) {
            println(message)
        }

        override fun findUserById(id: Int): String {
            return "User $id"
        }
    }

    context(users) {
        outputMessage("Looking up the first user")
        println(firstUser)
        // User 1
    }
}
```

You can use `_` as a context parameter name when you don't need to refer to the parameter directly. An anonymous context parameter can still satisfy the context parameters required by called functions, but you can't access it by name. To access its value explicitly, use `contextOf<T>()`:

```
// Uses "_" as context parameter name
context(_: UserService)
fun logWelcome() {
    // The anonymous parameter satisfies the UserService context parameter
    // required by outputMessage()
    outputMessage("Welcome!")

    // Retrieves the UserService value explicitly
    contextOf<UserService>().log("Hi!")
}
```

Context parameters resolution

Kotlin resolves context parameters at the call site by searching for matching context values in the current scope. Kotlin matches them by their type. If multiple compatible values exist at the same scope level, the compiler reports an ambiguity:

```
// UserService defines the dependency required in context
interface UserService {
    fun log(message: String)
}

// Declares a function with a context parameter
context(users: UserService)
fun outputMessage(message: String) {
    users.log("Log: $message")
}

fun main() {
    // Implements UserService
    val serviceA = object : UserService {
        override fun log(message: String) = println("A: $message")
    }

    // Implements UserService
    val serviceB = object : UserService {
        override fun log(message: String) = println("B: $message")
    }

    // Both serviceA and serviceB match the expected UserService type at the call site
    context(serviceA, serviceB) {
        // This results in an ambiguity error
        outputMessage("This will not compile")
    }
}
```

Experimental

Pass context arguments explicitly

When overloads differ only by context parameters, a call can become ambiguous if multiple matching context values are available.

To resolve the ambiguity, pass an explicit context argument at the call site:

```
class EmailSender
class SmsSender

context(emailSender: EmailSender)
fun sendNotification() {
    println("Sent email notification")
}

context(smsSender: SmsSender)
fun sendNotification() {
    println("Sent SMS notification")
}

context(defaultEmailSender: EmailSender, defaultSmsSender: SmsSender)
fun notifyUser() {
    // Selects the overload with the EmailSender context parameter
    sendNotification(emailSender = defaultEmailSender)

    // Selects the overload with the SmsSender context parameter
    sendNotification(smsSender = defaultSmsSender)
}
```

You can also use explicit context arguments to reduce nesting in some function calls:

- For a single call, use explicit context arguments to make the call easier to read.
- If multiple calls use the same context arguments, use the `context()` function.

This feature is Experimental. To opt in, add the following compiler option to your build file:

Gradle

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xexplicit-context-arguments")
    }
}
```

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-Xexplicit-context-arguments</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Restrictions

Context parameters are in continuous improvement, and some of the current restrictions include:

- Constructors can't declare context parameters.
- Properties with context parameters can't have backing fields or initializers.
- Properties with context parameters can't use delegation.

Despite these restrictions, context parameters simplify managing dependencies through simplified dependency injection, improved DSL design, and scoped operations.

Inline functions

Using higher-order functions imposes certain runtime penalties: each function is an object, and it captures a closure. A closure is a scope of variables that can be accessed in the body of the function. Memory allocations (both for function objects and classes) and virtual calls introduce runtime overhead.

But it appears that in many cases this kind of overhead can be eliminated by inlining the lambda expressions. The functions shown below are good examples of this situation. The `lock()` function could be easily inlined at call-sites. Consider the following case:

```
lock(l) { foo() }
```

Instead of creating a function object for the parameter and generating a call, the compiler could emit the following code:

```
l.lock()
try {
    foo()
} finally {
    l.unlock()
}
```

To make the compiler do this, mark the `lock()` function with the `inline` modifier:

```
inline fun <T> lock(lock: Lock, body: () -> T): T { ... }
```

The `inline` modifier affects both the function itself and the lambdas passed to it: all of those will be inlined into the call site.

Inlining may cause the generated code to grow. However, if you do it in a reasonable way (avoiding inlining large functions), it will pay off in performance, especially at "megamorphic" call-sites inside loops.

noinline

If you don't want all of the lambdas passed to an inline function to be inlined, mark some of your function parameters with the `noinline` modifier:

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) { ... }
```

Inlinable lambdas can only be called inside inline functions or passed as inlinable arguments. `noinline` lambdas, however, can be manipulated in any way you like, including being stored in fields or passed around.

> If an inline function has no inlinable function parameters and no reified type parameters, the compiler will issue a warning, since inlining such functions is very unlikely to be beneficial (you can use the `@Suppress("NOTHING_TO_INLINE")` annotation to suppress the warning if you are sure the inlining is needed).

Non-local jump expressions

Returns

In Kotlin, you can only use a normal, unqualified `return` to exit a named function or an anonymous function. To exit a lambda, use a label. A bare `return` is forbidden inside a lambda because a lambda cannot make the enclosing function `return` :

```
fun ordinaryFunction(block: () -> Unit) {
    println("hi!")
}
fun foo() {
    ordinaryFunction {
        return // ERROR: cannot make `foo` return here
    }
}
fun main() {
    foo()
}
```

But if the function the lambda is passed to is inlined, the return can be inlined, as well. So it is allowed:

```
inline fun inlined(block: () -> Unit) {
    println("hi!")
}
fun foo() {
    inlined {
        return // OK: the lambda is inlined
    }
}
fun main() {
    foo()
}
```

Such returns (located in a lambda, but exiting the enclosing function) are called *non-local* returns. This sort of construct usually occurs in loops, which inline functions often enclose:

```
fun hasZeros(ints: List<Int>): Boolean {
    ints.forEach {
        if (it == 0) return true // returns from hasZeros
    }
    return false
}
```

Note that some inline functions may call the lambdas passed to them as parameters not directly from the function body, but from another execution context, such as a local object or a nested function. In such cases, non-local control flow is also not allowed in the lambdas. To indicate that the lambda parameter of the inline function cannot use non-local returns, mark the lambda parameter with the `crossinline` modifier:

```
inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ...
}
```

Break and continue

Similar to non-local `return`, you can apply `break` and `continue` jump expressions in lambdas passed as arguments to an inline function that encloses a loop:

```
fun processList(elements: List<Int>): Boolean {
    for (element in elements) {
        val variable = element.nullableMethod() ?: run {
            log.warning("Element is null or invalid, continuing...")
            continue
        }
        if (variable == 0) return true
    }
    return false
}
```

Reified type parameters

Sometimes you need to access a type passed as a parameter:

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p ≠ null && !clazz.isInstance(p)) {
        p = p.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T?
}
```

Here, you walk up a tree and use reflection to check whether a node has a certain type. It's all fine, but the call site is not very pretty:

```
treeNode.findParentOfType(MyTreeNode::class.java)
```

A better solution would be to simply pass a type to this function. You can call it as follows:

```
treeNode.findParentOfType<MyTreeNode>()
```

To enable this, inline functions support *reified type parameters*, so you can write something like this:

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p ≠ null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

The code above qualifies the type parameter with the `reified` modifier to make it accessible inside the function, almost as if it were a normal class. Since the function is inlined, no reflection is needed and normal operators like `!is` and `as` are now available for you to use. Also, you can call the function as shown above: `myTree.findParentOfType<MyTreeNodeType>()`.

Though reflection may not be needed in many cases, you can still use it with a reified type parameter:

```
inline fun <reified T> membersOf() = T::class.members

fun main(s: Array<String>) {
    println(membersOf<StringBuilder>().joinToString("\n"))
}
```

Normal functions (not marked as inline) cannot have reified parameters. A type that does not have a run-time representation (for example, a non-reified type parameter or a fictitious type like `Nothing`) cannot be used as an argument for a reified type parameter.

Inline properties

The `inline` modifier can be used on accessors of properties that don't have backing fields. You can annotate individual property accessors:

```
val foo: Foo
    inline get() = Foo()

var bar: Bar
    get() = ...
    inline set(v) { ... }
```

You can also annotate an entire property, which marks both of its accessors as `inline`:

```
inline var bar: Bar
    get() = ...
    set(v) { ... }
```

At the call site, inline accessors are inlined as regular inline functions.

Restrictions for public API inline functions

When an inline function is `public` or `protected` but is not a part of a `private` or `internal` declaration, it is considered a module's public API. It can be called in other modules and is inlined at such call sites as well.

This imposes certain risks of binary incompatibility caused by changes in the module that declares an inline function in case the calling module is not re-compiled after the change.

To eliminate the risk of such incompatibility being introduced by a change in a *non*-public API of a module, public API inline functions are not allowed to use non-public-API declarations, i.e. `private` and `internal` declarations and their parts, in their bodies.

An `internal` declaration can be annotated with `@PublishedApi`, which allows its use in public API inline functions. When an `internal` inline function is marked as `@PublishedApi`, its body is checked too, as if it were public.

Operator overloading

Kotlin allows you to provide custom implementations for the predefined set of operators on types. These operators have predefined symbolic representation (like `+` or `*`) and precedence. To implement an operator, provide a member function or an extension function with a specific name for the corresponding type. This type becomes the left-hand side type for binary operations and the argument type for the unary ones.

To overload an operator, mark the corresponding function with the `operator` modifier:

```
interface IndexedContainer {
    operator fun get(index: Int)
}
```

When overriding your operator overloads, you can omit `operator`:

```
class OrdersList: IndexedContainer {
    override fun get(index: Int) { /*...*/ }
}
```

Unary operations

Unary prefix operators

Expression	Translated to
`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`

This table says that when the compiler processes, for example, an expression `+a`, it performs the following steps:

- Determines the type of `a`, let it be `T`.
- Looks up a function `unaryPlus()` with the `operator` modifier and no parameters for the receiver `T`, that means a member function or an extension function.
- If the function is absent or ambiguous, it is a compilation error.
- If the function is present and its return type is `R`, the expression `+a` has type `R`.

These operations, as well as all the others, are optimized for basic types and do not introduce overhead of function calls for them.

As an example, here's how you can overload the unary minus operator:

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

val point = Point(10, 20)

fun main() {
   println(-point)  // prints "Point(x=-10, y=-20)"
}
```

Increments and decrements

Expression	Translated to
`a++`	`a.inc()` + see below
`a--`	`a.dec()` + see below

The `inc()` and `dec()` functions must return a value, which will be assigned to the variable on which the `++` or `--` operation was used. They shouldn't mutate the object on which the `inc` or `dec` was invoked.

The compiler performs the following steps for resolution of an operator in the *postfix* form, for example `a++`:

- Determines the type of `a`, let it be `T`.
- Looks up a function `inc()` with the `operator` modifier and no parameters, applicable to the receiver of type `T`.
- Checks that the return type of the function is a subtype of `T`.

The effect of computing the expression is:

- Store the initial value of `a` to a temporary storage `a0`.
- Assign the result of `a0.inc()` to `a`.
- Return `a0` as the result of the expression.

For `a--` the steps are completely analogous.

For the *prefix* forms `++a` and `--a` resolution works the same way, and the effect is:

- Assign the result of `a.inc()` to `a`.
- Return the new value of `a` as a result of the expression.

Binary operations

Arithmetic operators

Expression	Translated to
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`
`a * b`	`a.times(b)`
`a / b`	`a.div(b)`
`a % b`	`a.rem(b)`

Expression	Translated to
`a..b`	`a.rangeTo(b)`
`a..<b`	`a.rangeUntil(b)`

For the operations in this table, the compiler just resolves the expression in the *Translated to* column.

Below is an example `Counter` class that starts at a given value and can be incremented using the overloaded `+` operator:

```
data class Counter(val dayIndex: Int) {
    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}
```

in operator

Expression	Translated to
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`

For `in` and `!in` the procedure is the same, but the order of arguments is reversed.

Indexed access operator

Expression	Translated to
`a[i]`	`a.get(i)`
`a[i, j]`	`a.get(i, j)`
`a[i_1, ..., i_n]`	`a.get(i_1, ..., i_n)`
`a[i] = b`	`a.set(i, b)`
`a[i, j] = b`	`a.set(i, j, b)`
`a[i_1, ..., i_n] = b`	`a.set(i_1, ..., i_n, b)`

Square brackets are translated to calls to `get` and `set` with appropriate numbers of arguments.

invoke operator

Expression	Translated to
`a()`	`a.invoke()`
`a(i)`	`a.invoke(i)`
`a(i, j)`	`a.invoke(i, j)`
`a(i_1, ..., i_n)`	`a.invoke(i_1, ..., i_n)`

Parentheses are translated to calls to `invoke` with appropriate number of arguments.

Augmented assignments

Expression	Translated to
`a += b`	`a.plusAssign(b)`
`a -= b`	`a.minusAssign(b)`
`a *= b`	`a.timesAssign(b)`
`a /= b`	`a.divAssign(b)`
`a %= b`	`a.remAssign(b)`

For the assignment operations, for example `a += b`, the compiler performs the following steps:

- If the function from the right column is available:
  - If the corresponding binary function (that means `plus()` for `plusAssign()`) is available too, `a` is a mutable variable, and the return type of `plus` is a subtype of the type of `a`, report an error (ambiguity).
  - Make sure its return type is `Unit`, and report an error otherwise.
  - Generate code for `a.plusAssign(b)`.
- Otherwise, try to generate code for `a = a + b` (this includes a type check: the type of `a + b` must be a subtype of `a`).

Assignments are *NOT* expressions in Kotlin.

Equality and inequality operators

Expression	Translated to
`a == b`	`a?.equals(b) ?: (b === null)`
`a != b`	`!(a?.equals(b) ?: (b === null))`

These operators only work with the function `equals(other: Any?): Boolean`, which you can override to provide a custom equality check implementation. Any other function with the same name (like `equals(other: Foo)`) is ignored.

Kotlin calls `.equals()` when neither operand compares directly to `null` in the `==` expression and the comparison isn't between two floating-point types. Otherwise, Kotlin uses `===` for direct `null` comparisons and compares non-null floating-point values by numeric value.

`===` and `!==` (identity checks) aren't overloadable, so no conventions exist for them.

Comparison operators

Expression	Translated to
`a > b`	`a.compareTo(b) > 0`
`a < b`	`a.compareTo(b) < 0`
`a >= b`	`a.compareTo(b) >= 0`
`a <= b`	`a.compareTo(b) <= 0`

All comparisons are translated into calls to `compareTo`, that is required to return `Int`.

Property delegation operators

`provideDelegate`, `getValue` and `setValue` operator functions are described in Delegated properties.

Infix calls for named functions

You can simulate custom infix operations by using infix function calls.

Experimental

Unused return value checker

This feature is planned to be stabilized and improved in future Kotlin releases. We would appreciate your feedback in our issue tracker YouTrack.

For more information, see the related KEEP proposal.

The unused return value checker allows you to detect *ignored results*. These are values returned from expressions that produce something other than `Unit`, `Nothing`, or `Nothing?` and aren't:

- Stored in a variable or property.
- Returned or thrown.
- Passed as an argument to another function.
- Used as a receiver in a call or safe call.
- Checked in a condition such as `if`, `when`, or `while`.
- Used as the last statement of a lambda.

The checker doesn't report ignored results for increment operations like `++` and `--`, or for boolean shortcuts where the right-hand side exits the current function, for example, `condition || return`.

You can use the unused return value checker to catch bugs where a function call produces a meaningful result, but the result is silently dropped. This helps prevent unexpected behavior and makes such issues easier to track down.

Here's an example where a string is created but never used, so the checker reports it as an ignored result:

```
fun formatGreeting(name: String): String {
    if (name.isBlank()) return "Hello, anonymous user!"
    if (!name.contains(' ')) {
        // The checker reports a warning that this result is ignored:
        // "Unused return value of 'plus'."
        "Hello, " + name.replaceFirstChar(Char::titlecase) + "!"
    }
    val (first, last) = name.split(' ')
    return "Hello, $first! Or should I call you Dr. $last?"
}
```

Configure the unused return value checker

You can control how the compiler reports ignored results with the `-Xreturn-value-checker` compiler option.

It has the following modes:

- `disable` disables the unused return value checker (default).
- `check` enables the checker, and reports warnings for ignored results from marked functions.
- `full` enables the checker, treats all functions in your project as marked, and reports warnings for ignored results.

All marked functions are propagated as such, and ignored results are reported if the checker is enabled in projects that use your code as a dependency.

To use the unused return value checker in your project, add the compiler option to your build configuration file:

Gradle

```
// build.gradle(.kts)
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xreturn-value-checker=check")
    }
}
```

Maven

```
<!-- pom.xml -->
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    ..
    <configuration>
        <args>
            <arg>-Xreturn-value-checker=check</arg>
        </args>
    </configuration>
</plugin>
```

Mark functions to check ignored results

When you set the `-Xreturn-value-checker` compiler option to `check`, the checker reports ignored results only from expressions that are marked, like most functions in the Kotlin standard library.

To mark your own code, use the `@MustUseReturnValues` annotation. You can apply it to a file, class, or function depending on the scope you want the checker to cover.

For example, you can mark an entire file:

```
// Marks all functions and classes in this file so the checker reports unused return values
@file:MustUseReturnValues

package my.project

fun someFunction(): String
```

Or a specific class:

```
// Marks all functions in this class so the checker reports unused return values
@MustUseReturnValues
class Greeter {
    fun greet(name: String): String = "Hello, $name"
}

fun someFunction(): Int = ...
```

You can apply the checker to your entire project by setting the `-Xreturn-value-checker` compiler option to `full`. With this option, you don't have to annotate your code with `@MustUseReturnValues`.

Suppress reports for ignored results

You can suppress reports on specific functions by annotating them with `@IgnorableReturnValue`. Annotate functions where ignoring the result is common and expected, such as `MutableList.add`:

```
@IgnorableReturnValue
fun <T> MutableList<T>.addAndIgnoreResult(element: T): Boolean {
    return add(element)
}
```

You can suppress a warning without annotating the function itself. To do this, assign the result to a special unnamed variable with an underscore syntax (`_`):

```
// Non-ignorable function
fun computeValue(): Int = 42

fun main() {

    // Reports a warning: result is ignored
    computeValue()

    // Suppresses the warning only at this call site with a special unused variable
    val _ = computeValue()
}
```

Ignored results in function overrides

When you override a function, the override inherits the reporting rules defined by the annotations on the base declaration. This also applies when the base declaration is part of the Kotlin standard library or of other library dependencies, so the checker reports ignored results for overrides of functions like `Any.hashCode()`.

Additionally, you can't override a function marked with `@IgnorableReturnValue` with another function that requires its return value to be used. However, you can mark an override with `@IgnorableReturnValue` in a class or interface annotated with `@MustUseReturnValues` when its result can be safely ignored:

```
@MustUseReturnValues
interface Greeter {
    fun greet(name: String): String
}

object SilentGreeter : Greeter {
    @IgnorableReturnValue
    override fun greet(name: String): String = ""
}

fun check(g: Greeter) {
    // Reports a warning: unused return value
    g.greet("John")

    // No warning
    SilentGreeter.greet("John")
}
```

Check for unused results in higher-order functions

Some higher-order functions, such as the `let` scope function, return the result of a lambda. To check for unused lambda results of higher-order functions, add the Experimental `returnsResultOf()` contract to the function's contract.

> Kotlin contracts are Experimental. To opt in, add the `@OptIn(ExperimentalContracts::class)` annotation when declaring a function with a contract.

Here's an example:

```
import kotlin.contracts.ExperimentalContracts
import kotlin.contracts.contract

@OptIn(ExperimentalContracts::class)
inline fun <T, R> T.customLet(block: (T) -> R): R {
    contract {
        returnsResultOf(block)
    }
    return block(this)
}
```

You can then use a function with this contract, such as `.customLet()`, to check if the lambda result is used:

```
fun handleNullablePackageName(packageName: String?, builder: StringBuilder) {
    // The checker doesn't report a warning because the return value of append() can be ignored
    packageName?.customLet { builder.append(it) }

    // The checker reports a warning because the returned string is unused
    packageName?.customLet { "kotlin.$it" }
}
```

> The `returnsResultOf()` contract requires a separate compiler option to opt in. Be aware that using it produces pre-release binaries that Kotlin compiler versions earlier than 2.4.0 can't read.

To opt in for your project, add the following compiler option to your build file:

Gradle

```
// build.gradle(.kts)
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xallow-returns-result-of")
    }
}
```

Maven

```
<!-- pom.xml -->
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-Xallow-returns-result-of</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Interoperability with Java annotations

Some Java libraries use similar mechanisms with different annotations. The unused return value checker treats the following annotations as equivalent to using `@MustUseReturnValues`:

- `com.google.errorprone.annotations.CheckReturnValue`
- `edu.umd.cs.findbugs.annotations.CheckReturnValue`
- `org.jetbrains.annotations.CheckReturnValue`
- `org.springframework.lang.CheckReturnValue`
- `org.jooq.CheckReturnValue`

It also treats `com.google.errorprone.annotations.CanIgnoreReturnValue` as equivalent to using `@IgnorableReturnValue`.

Classes

> Before creating classes, consider using a data class if the purpose is to store data. Alternatively, think about extending an existing class with an extension, rather than creating a new one from scratch.

Like other object-oriented languages, Kotlin uses *classes* to encapsulate data (properties) and behavior (functions) for reusable, structured code.

Classes are blueprints or templates for objects, which you create via constructors. When you create an instance of a class, you are creating a concrete object based on that blueprint.

Kotlin offers concise syntax for declaring classes. To declare a class, use the `class` keyword followed by the class name:

```
class Person { /*...*/ }
```

The class declaration consists of:

- **Class header**, including but not limited to:

- `class` keyword
- Class name
- Type parameters (if any)
- Primary constructor (optional)

- **Class body** (optional), surrounded by curly braces `{}`, and including **class members** such as:
  - Secondary constructors
  - Initializer blocks
  - Functions
  - Properties
  - Nested and inner classes
  - Object declarations

You can keep both the class header and body to a bare minimum. If the class doesn't have a body, you can omit the curly braces `{}`:

```
// Class with primary constructor, but without a body
class Person(val name: String, var age: Int)
```

Here's an example that declares a class with a header and body, then creates an instance from it:

```
// Person class with a primary constructor
// that initializes the name property
class Person(val name: String) {
    // Class body with age property
    var age: Int = 0
}

fun main() {
    // Creates an instance of the Person class by calling the constructor
    val person = Person("Alice")

    // Accesses the instance's properties
    println(person.name)
    // Alice
    println(person.age)
    // 0
}
```

Creating instances

An instance is created when you use the class as a blueprint to build a real object to work with in your program.

To create an instance of a class, use the class name followed by parentheses `()`, similar to calling a function:

```
// Creates an instance of the Person class
val anonymousUser = Person()
```

In Kotlin, you can create instances:

- **Without arguments** (`Person()`): creates an instance using the default values, if they are declared in the class.

- **With arguments** (`Person(value)`): creates an instance by passing specific values.

You can assign the created instance to a mutable (`var`) or read-only (`val`) variable:

```
// Creates an instance using the default value
// and assigns it to a mutable variable
var anonymousUser = Person()

// Creates an instance by passing a specific value
// and assigns it to a read-only variable
val namedUser = Person("Joe")
```

It's possible to create instances wherever you need them, inside the `main()` function, within other functions, or inside another class. Additionally, you can create instances inside another function and call that function from `main()` .

The following code declares a `Person` class with a property for storing a name. It also demonstrates how to create an instance with both the default constructor's value and a specific value:

```
// Class header with a primary constructor
// that initializes name with a default value
class Person(val name: String = "Sebastian")

fun main() {
    // Creates an instance using the default constructor's value
    val anonymousUser = Person()

    // Creates an instance by passing a specific value
    val namedUser = Person("Joe")

    // Accesses the instances' name property
    println(anonymousUser.name)
    // Sebastian
    println(namedUser.name)
    // Joe
}
```

In Kotlin, unlike other object-oriented programming languages, there is no need for the `new` keyword when creating class instances.

For information about creating instances of nested, inner, and anonymous inner classes, see the Nested classes section.

Constructors and initializer blocks

When you create a class instance, you call one of its constructors. A class in Kotlin can have a *primary constructor* and one or more *secondary constructors*.

The primary constructor is the main way to initialize a class. You declare it in the class header. A secondary constructor provides additional initialization logic. You declare it in the class body.

Both primary and secondary constructors are optional, but a class must have at least one constructor.

Primary constructor

The primary constructor sets up the initial state of an instance when it's created.

To declare a primary constructor, place it in the class header after the class name:

```
class Person constructor(name: String) { /*...*/ }
```

If the primary constructor doesn't have any annotations or visibility modifiers, you can omit the `constructor` keyword:

```
class Person(name: String) { /*...*/ }
```

The primary constructor can declare parameters as properties. Use the `val` keyword before the argument name to declare a read-only property and the `var` keyword for a mutable property:

```
class Person(val name: String, var age: Int) { /*...*/ }
```

These constructor parameter properties are stored as part of the instance and are accessible from outside the class.

It's also possible to declare primary constructor parameters that are not properties. These parameters don't have `val` or `var` in front of them, so they are not stored in the instance and are available only within the class body:

```
// Primary constructor parameter that is also a property
class PersonWithProperty(val name: String) {
    fun greet() {
        println("Hello, $name")
    }
}

// Primary constructor parameter only (not stored as a property)
class PersonWithAssignment(name: String) {
    // Must be assigned to a property to be usable later
    val displayName: String = name

    fun greet() {
        println("Hello, $displayName")
    }
}
```

Properties declared in the primary constructor are accessible by member functions of the class:

```
// Class with a primary constructor declaring properties
class Person(val name: String, var age: Int) {
    // Member function accessing class properties
    fun introduce(): String {
        return "Hi, I'm $name and I'm $age years old."
    }
}
```

You can also assign default values to properties in the primary constructor:

```
class Person(val name: String = "John", var age: Int = 30) { /*...*/ }
```

If no value is passed to the constructor during instance creation, properties use their default value:

```
// Class with a primary constructor
// including default values for name and age
class Person(val name: String = "John", var age: Int = 30)

fun main() {
    // Creates an instance using default values
    val person = Person()
    println("Name: ${person.name}, Age: ${person.age}")
    // Name: John, Age: 30
}
```

You can use the primary constructor parameters to initialize additional class properties directly in the class body:

```
// Class with a primary constructor
// including default values for name and age
class Person(
    val name: String = "John",
    var age: Int = 30
) {
    // Initializes the description property
    // from the primary constructor parameters
    val description: String = "Name: $name, Age: $age"
}

fun main() {
    // Creates an instance of the Person class
    val person = Person()
    // Accesses the description property
    println(person.description)
    // Name: John, Age: 30
}
```

As with functions, you can use trailing commas in constructor declarations:

```
class Person(
    val name: String,
    val lastName: String,
    var age: Int,
) { /*...*/ }
```

Initializer blocks

The primary constructor initializes the class and sets its properties. In most cases, you can handle this with simple code.

If you need to perform more complex operations during instance creation, place that logic in *initializer blocks* inside the class body. These blocks run when the primary constructor executes.

Declare initializer blocks with the `init` keyword followed by curly braces `{}`. Write within the curly braces any code that you want to run during initialization:

```
// Class with a primary constructor that initializes name and age
class Person(val name: String, var age: Int) {
    init {
        // Initializer block runs when an instance is created
        println("Person created: $name, age $age.")
    }
}

fun main() {
    // Creates an instance of the Person class
    Person("John", 30)
    // Person created: John, age 30.
}
```

Add as many initializer blocks (`init {}`) as you need. They run in the order in which they appear in the class body, along with property initializers:

```
// Class with a primary constructor that initializes name and age
class Person(val name: String, var age: Int) {
    // First initializer block
    init {
        // Runs first when an instance is created
        println("Person created: $name, age $age.")
    }

    // Second initializer block
    init {
        // Runs after the first initializer block
        if (age < 18) {
            println("$name is a minor.")
        } else {
            println("$name is an adult.")
        }
    }
}

fun main() {
    // Creates an instance of the Person class
    Person("John", 30)
    // Person created: John, age 30.
    // John is an adult.
}
```

You can use primary constructor parameters in initializer blocks. For example, in the code above, the first and second initializers use the `name` and `age` parameters from the primary constructor.

A common use case for `init` blocks is data validation. For example, by calling the `require` function:

```
class Person(val age: Int) {
    init {
        require(age > 0) { "age must be positive" }
    }
}
```

Secondary constructors

In Kotlin, secondary constructors are additional constructors that a class can have beyond its primary constructor. Secondary constructors are useful when you need multiple ways to initialize a class or for Java interoperability.

To declare a secondary constructor, use the `constructor` keyword inside the class body with the constructor parameters within parentheses `()`. Add the constructor logic within curly braces `{}`:

```
// Class header with a primary constructor that initializes name and age
class Person(val name: String, var age: Int) {

    // Secondary constructor that takes age as a
    // String and converts it to an Int
    constructor(name: String, age: String) : this(name, age.toIntOrNull() ?: 0) {
        println("$name created with converted age: ${this.age}")
    }
}

fun main() {
    // Uses the secondary constructor with age as a String
    Person("Bob", "8")
    // Bob created with converted age: 8
}
```

The expression `age.toIntOrNull() ?: 0` uses the Elvis operator. For more information, see Null safety.

In the code above, the secondary constructor delegates to the primary constructor via the `this` keyword, passing `name` and the `age` value converted to an integer.

In Kotlin, secondary constructors must delegate to the primary constructor. This delegation ensures that all primary constructor initialization logic is executed before any secondary constructor logic runs.

Constructor delegation can be:

- **Direct**, where the secondary constructor calls the primary constructor immediately.
- **Indirect**, where one secondary constructor calls another, which in turn delegates to the primary constructor.

Here's an example demonstrating how direct and indirect delegation works:

```
// Class header with a primary constructor that initializes name and age
class Person(
    val name: String,
    var age: Int
) {
    // Secondary constructor with direct delegation
    // to the primary constructor
    constructor(name: String) : this(name, 0) {
        println("Person created with default age: $age and name: $name.")
    }

    // Secondary constructor with indirect delegation:
    // this("Bob") → constructor(name: String) → primary constructor
    constructor() : this("Bob") {
        println("New person created with default age: $age and name: $name.")
    }
}

fun main() {
    // Creates an instance based on the direct delegation
    Person("Alice")
    // Person created with default age: 0 and name: Alice.

    // Creates an instance based on the indirect delegation
    Person()
    // Person created with default age: 0 and name: Bob.
    // New person created with default age: 0 and name: Bob.
}
```

In classes with initializer blocks (`init {}`), the code within these blocks becomes part of the primary constructor. Given that secondary constructors delegate to the primary constructor first, all initializer blocks and property initializers run before the body of the secondary constructor. Even if the class has no primary constructor, the delegation still happens implicitly:

```
// Class header with no primary constructor
class Person {
    // Initializer block runs when an instance is created
    init {
        // Runs before the secondary constructor
        println("1. First initializer block runs")
    }

    // Secondary constructor that takes an integer parameter
    constructor(i: Int) {
        // Runs after the initializer block
        println("2. Person $i is created")
    }
}

fun main() {
    // Creates an instance of the Person class
    Person(1)
    // 1. First initializer block runs
    // 2. Person 1 created
}
```

Classes without constructors

Classes that don't declare any constructors (primary or secondary) have an implicit primary constructor with no parameters:

```
// Class with no explicit constructors
class Person {
    // No primary or secondary constructors declared
}

fun main() {
    // Creates an instance of the Person class
    // using the implicit primary constructor
    val person = Person()
}
```

The visibility of this implicit primary constructor is public, meaning it can be accessed from anywhere. If you don't want your class to have a public constructor, declare an empty primary constructor with non-default visibility:

```
class Person private constructor() { /*...*/ }
```

On the JVM, if all primary constructor parameters have default values, the compiler implicitly provides a parameterless constructor that uses those default values.

This makes it easier to use Kotlin with libraries such as Jackson or Spring Data JPA, which create class instances through parameterless constructors.

In the following example, Kotlin implicitly provides a parameterless constructor `Person()` that uses the default value `""`:

```
class Person(val personName: String = "")
```

Inheritance

Class inheritance in Kotlin allows you to create a new class (derived class) from an existing class (base class), inheriting its properties and functions while adding or modifying behavior.

For detailed information about inheritance hierarchies and how to use the `open` keyword, see the Inheritance section.

Abstract classes

In Kotlin, abstract classes are classes that can't be instantiated directly. They are designed to be inherited by other classes which define their actual behavior. This behavior is called an *implementation*.

An abstract class can declare abstract properties and functions, which must be implemented by subclasses.

Abstract classes can also have constructors. These constructors initialize class properties and enforce required parameters for subclasses. Declare an abstract class using the `abstract` keyword:

```
abstract class Person(val name: String, val age: Int)
```

An abstract class can have both abstract and non-abstract members (properties and functions). To declare a member as abstract, you must use the `abstract` keyword explicitly.

You don't need to annotate abstract classes or functions with the `open` keyword because they are implicitly inheritable by default. For more details about the `open` keyword, see Inheritance.

Abstract members don't have an implementation in the abstract class. You define the implementation in a subclass or inheriting class with an `override` function or property:

```
// Abstract class with a primary constructor that declares name and age
abstract class Person(
    val name: String,
    val age: Int
) {
    // Abstract member
    // Doesn't provide implementation,
    // and it must be implemented by subclasses
    abstract fun introduce()

    // Non-abstract member (has an implementation)
    fun greet() {
        println("Hello, my name is $name.")
    }
}

// Subclass that provides an implementation for the abstract member
class Student(
    name: String,
    age: Int,
    val school: String
) : Person(name, age) {
    override fun introduce() {
        println("I am $name, $age years old, and I study at $school.")
    }
}

fun main() {
    // Creates an instance of the Student class
    val student = Student("Alice", 20, "Engineering University")

    // Calls the non-abstract member
    student.greet()
    // Hello, my name is Alice.

    // Calls the overridden abstract member
    student.introduce()
    // I am Alice, 20 years old, and I study at Engineering University.
}
```

Companion objects

In Kotlin, each class can have a companion object. Companion objects are a type of object declaration that allows you to access its members using the class name without creating a class instance.

Suppose you need to write a function that can be called without creating an instance of a class, but it is still logically connected to the class (such as a factory function). In that case, you can declare it inside a companion object declaration within the class:

```
// Class with a primary constructor that declares the name property
class Person(
    val name: String
) {
    // Class body with a companion object
    companion object {
        fun createAnonymous() = Person("Anonymous")
    }
}

fun main() {
    // Calls the function without creating an instance of the class
    val anonymous = Person.createAnonymous()
    println(anonymous.name)
    // Anonymous
}
```

If you declare a companion object inside your class, you can access its members using only the class name as a qualifier.

For more information, see Companion objects.

Data classes

Data classes in Kotlin are primarily used to hold data. For each data class, the compiler automatically generates additional member functions that allow you to print an instance to readable output, compare instances, copy instances, and more. Data classes are marked with `data`:

```
data class User(val name: String, val age: Int)
```

The compiler automatically derives the following members from all properties declared in the primary constructor:

- `equals()` / `hashCode()` pair.
- `toString()` of the form `"User(name=John, age=42)"`.
- `componentN()` functions corresponding to the properties in their order of declaration.
- `copy()` function.

To ensure consistency and meaningful behavior of the generated code, data classes have to fulfill the following requirements:

- The primary constructor must have at least one parameter.
- All primary constructor parameters must be marked as `val` or `var`.
- Data classes can't be abstract, open, sealed, or inner.

Additionally, the generation of data class members follows these rules with regard to the members' inheritance:

- If there are explicit implementations of `equals()`, `hashCode()`, or `toString()` in the data class body or `final` implementations in a superclass, then these functions are not generated, and the existing implementations are used.
- If a supertype has `componentN()` functions that are `open` and return compatible types, the corresponding functions are generated for the data class and override those of the supertype. If the functions of the supertype cannot be overridden due to incompatible signatures or due to their being final, an error is reported.

- Providing explicit implementations for the `componentN()` and `copy()` functions is not allowed.

Data classes may extend other classes (see Sealed classes for examples).

> On the JVM, if the generated class needs to have a parameterless constructor, default values for the properties have to be specified (see Constructors):
>
> ```
> data class User(val name: String = "", val age: Int = 0)
> ```

Properties declared in the class body

The compiler only uses the properties defined inside the primary constructor for the automatically generated functions. To exclude a property from the generated implementations, declare it inside the class body:

```
data class Person(val name: String) {
    var age: Int = 0
}
```

In the example below, only the `name` property is used by default inside the `toString()`, `equals()`, `hashCode()`, and `copy()` implementations, and there is only one component function, `component1()`. The `age` property is declared inside the class body and is excluded. Therefore, two `Person` objects with the same `name` but different `age` values are considered equal since `equals()` only evaluates properties from the primary constructor:

```
data class Person(val name: String) {
    var age: Int = 0
}
fun main() {
    val person1 = Person("John")
    val person2 = Person("John")
    person1.age = 10
    person2.age = 20

    println("person1 == person2: ${person1 == person2}")
    // person1 == person2: true

    println("person1 with age ${person1.age}: ${person1}")
    // person1 with age 10: Person(name=John)

    println("person2 with age ${person2.age}: ${person2}")
    // person2 with age 20: Person(name=John)
}
```

Copying

Use the `copy()` function to copy an object, allowing you to alter *some* of its properties while keeping the rest unchanged. The implementation of this function for the `User` class above would be as follows:

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

You can then write the following:

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

The `copy()` function creates a *shallow* copy of the instance. In other words, it doesn't copy components recursively. As a result, references to other objects are shared.

For example, if a property holds a mutable list, changes made through the "original" value are also visible through the copy, and changes made through the copy are visible through the original:

```
data class Employee(val name: String, val roles: MutableList<String>)

fun main() {
    val original = Employee("Jamie", mutableListOf("developer"))
    val duplicate = original.copy()

    duplicate.roles.add("team lead")

    println(original)
    // Employee(name=Jamie, roles=[developer, team lead])
    println(duplicate)
    // Employee(name=Jamie, roles=[developer, team lead])
}
```

As you can see, modifying the `duplicate.roles` property also changes the `original.roles` property because both properties share the same list reference.

Data classes and destructuring declarations

Component functions generated for data classes make it possible to use them in destructuring declarations:

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age")
// Jane, 35 years of age
```

Standard data classes

The standard library provides the `Pair` and `Triple` classes. In most cases, though, named data classes are a better design choice because they make the code easier to read by providing meaningful names for the properties.

Extensions

Kotlin *extensions* let you extend a class or an interface with new functionality without using inheritance or design patterns like *Decorator*. They are useful when working with third-party libraries you can't modify directly. Once created, you call these extensions as if they were members of the original class or interface.

The most common forms of extensions are *extension functions* and *extension properties*.

Importantly, extensions don't modify the classes or interfaces they extend. When you define an extension, you don't add new members. You make new functions callable or new properties accessible using the same syntax.

Receivers

Extensions are always called on a receiver. The receiver has to have the same type as the class or interface being extended. To use an extension, prefix it with the receiver followed by a `.` and the function or property name.

For example, the `.appendLine()` extension function from the standard library extends the `StringBuilder` class. So in this case, the receiver is a `StringBuilder` instance, and the *receiver type* is `StringBuilder`:

```
fun main() {
    // builder is an instance of StringBuilder
    val builder = StringBuilder()
        // Calls .appendLine() extension function on builder
        .appendLine("Hello")
        .appendLine()
        .appendLine("World")
    println(builder.toString())
    // Hello
    //
    // World
}
```

Extension functions

Before creating your own extension functions, see if what you are looking for is already available in the Kotlin standard library. The standard library provides many useful extension functions for:

- Operating on collections: `.map()`, `.filter()`, `.reduce()`, `.fold()`, `.groupBy()`.
- Converting to strings: `.joinToString()`.
- Working with null values: `.filterNotNull()`.

To create your own extension function, prefix its name with a receiver type followed by a `.`. In this example, the `.truncate()` function extends the `String` class, so the receiver type is `String`:

```
fun String.truncate(maxLength: Int): String {
    return if (this.length ≤ maxLength) this else take(maxLength - 3) + "..."
}

fun main() {
    val shortUsername = "KotlinFan42"
    val longUsername = "JetBrainsLoverForever"

    println("Short username: ${shortUsername.truncate(15)}")
    // KotlinFan42
    println("Long username:  ${longUsername.truncate(15)}")
    // JetBrainsLov...
}
```

The `.truncate()` function truncates any string that it's called on by the number in the `maxLength` argument and adds an ellipsis `...`. If the string is shorter than `maxLength`, the function returns the original string.

In this example, the `.displayInfo()` function extends the `User` interface:

```
interface User {
    val name: String
    val email: String
}

fun User.displayInfo(): String = "User(name=$name, email=$email)"

// Inherits from and implements the properties of the User interface
class RegularUser(override val name: String, override val email: String) : User

fun main() {
    val user = RegularUser("Alice", "alice@example.com")
    println(user.displayInfo())
    // User(name=Alice, email=alice@example.com)
}
```

The `.displayInfo()` function returns a string containing the `name` and `email` of a `RegularUser` instance. Defining an extension on an interface like this is useful when you want to add functionality to all types that implement an interface only once.

In this example, the `.mostVoted()` function extends the `Map<String, Int>` class:

```
fun Map<String, Int>.mostVoted(): String? {
    return maxByOrNull { (key, value) -> value }?.key
}

fun main() {
    val poll = mapOf(
        "Cats" to 37,
        "Dogs" to 58,
        "Birds" to 22
    )

    println("Top choice: ${poll.mostVoted()}")
    // Dogs
}
```

The `.mostVoted()` function iterates through the key-value pairs of the map it's called on and uses the `maxByOrNull()` function to return the key of the pair containing the highest value. If the map is empty, the `maxByOrNull()` function returns `null`. The `mostVoted()` function uses a safe call `?.` to only access the `key` property when the `maxByOrNull()` function returns a non-null value.

Generic extension functions

To create generic extension functions, declare the generic type parameter before the function name to make it available in the receiver type expression. In this example, the `.endpoints()` function extends `List<T>` where `T` can be any type:

```
fun <T> List<T>.endpoints(): Pair<T, T> {
    return first() to last()
}

fun main() {
    val cities = listOf("Paris", "London", "Berlin", "Prague")
    val temperatures = listOf(21.0, 19.5, 22.3)

    val cityEndpoints = cities.endpoints()
    val tempEndpoints = temperatures.endpoints()

    println("First and last cities: $cityEndpoints")
    // (Paris, Prague)
    println("First and last temperatures: $tempEndpoints")
    // (21.0, 22.3)
}
```

The `.endpoints()` function returns a pair containing the first and last elements of the list that it's called on. Inside the function body, it calls the `first()` and `last()` functions and combines their returned values into a `Pair` using the `to` infix function.

For more information about generics, see generic functions.

Nullable receivers

You can define extension functions with a nullable receiver type, which allows you to call them on a variable even if its value is null. When the receiver is `null`, `this` is also `null`. Make sure to handle nullability correctly within your functions. For example, use `this == null` checks inside function bodies, safe calls `?.`, or the Elvis operator `?:`.

In this example, you can call the `.toString()` function without checking for `null` because the check already happens inside the extension function:

```
fun main() {
        // Extension function on nullable Any
    fun Any?.toString(): String {
        if (this == null) return "null"
        // After null check, `this` is smart-cast to non-nullable Any
        // So this call resolves to the regular toString() function
        return toString()
    }

    val number: Int? = 42
    val nothing: Any? = null

    println(number.toString())
    // 42
    println(nothing.toString())
    // null
    //sampleEnd
}
```

Extension or member functions?

Since extension and member function calls have the same notation, how does the compiler know which one to use? Extension functions are dispatched *statically*, meaning the compiler determines which function to call based on the receiver type at compile time. For example:

```
fun main() {
    open class Shape
    class Rectangle: Shape()

    fun Shape.getName() = "Shape"
    fun Rectangle.getName() = "Rectangle"

    fun printClassName(shape: Shape) {
        println(shape.getName())
    }

    printClassName(Rectangle())
    // Shape
}
```

In this example, the compiler calls the `Shape.getName()` extension function because the parameter `shape` is declared as type `Shape`. Because extension functions are resolved statically, the compiler chooses the function based on the declared type, not the actual instance.

So even though the example passes a `Rectangle` instance, the `.getName()` function resolves to `Shape.getName()` since the variable is declared as type `Shape`.

If a class has a member function and there's an extension function with the same receiver type, the same name, and compatible arguments, the member function takes precedence. For example:

```
fun main() {
    class Example {
        fun printFunctionType() { println("Member function") }
    }

    fun Example.printFunctionType() { println("Extension function") }

    Example().printFunctionType()
    // Member function
}
```

However, extension functions can overload member functions that have the same name but a *different* signature:

```
fun main() {
    class Example {
        fun printFunctionType() { println("Member function") }
    }

    // Same name but different signature
    fun Example.printFunctionType(index: Int) { println("Extension function #$index") }

    Example().printFunctionType(1)
    // Extension function #1
}
```

In this example, since an `Int` is passed to the `.printFunctionType()` function, the compiler chooses the extension function because it matches the signature. The compiler ignores the member function, which takes no arguments.

Anonymous extension functions

You can define extension functions without giving them a name. This is useful when you want to avoid cluttering the global namespace or when you need to pass some extension behavior as a parameter.

For example, suppose you want to extend a data class with a one-time function to calculate shipping, without giving it a name:

```
fun main() {
        data class Order(val weight: Double)
    val calculateShipping = fun Order.(rate: Double): Double = this.weight * rate

    val order = Order(2.5)
    val cost = order.calculateShipping(3.0)
    println("Shipping cost: $cost")
    // Shipping cost: 7.5
}
```

To pass extension behavior as a parameter, use a lambda expression with a type annotation. For example, let's say you want to check if a number is within a range without defining a named function:

```
fun main() {
    val isInRange: Int.(min: Int, max: Int) -> Boolean = { min, max -> this in min..max }

    println(5.isInRange(1, 10))
    // true
    println(20.isInRange(1, 10))
    // false
}
```

In this example, the `isInRange` variable holds a function of type `Int.(min: Int, max: Int) -> Boolean`. The type is an extension function on the `Int` class that takes `min` and `max` parameters and returns a `Boolean`.

The lambda body `{ min, max -> this in min..max }` checks whether the `Int` value the function is called on falls within the range between `min` and `max` parameters. If the check is successful, the lambda returns `true`.

For more information, see Lambda expressions and anonymous functions.

Extension properties

Kotlin supports extension properties, which are useful for performing data transformations or creating UI display helpers without cluttering the class you're working with.

To create an extension property, write the name of the class that you want to extend, followed by a `.` and the name of your property.

For example, suppose you have a data class that represents a user with a first and last name, and you want to create a property that returns an email-style username when accessed. Your code might look like this:

```
data class User(val firstName: String, val lastName: String)

// An extension property to get a username-style email handle
val User.emailUsername: String
    get() = "${firstName.lowercase()}.${lastName.lowercase()}"

fun main() {
    val user = User("Mickey", "Mouse")
    // Calls extension property
    println("Generated email username: ${user.emailUsername}")
    // Generated email username: mickey.mouse
}
```

Since extensions don't actually add members to classes, there's no efficient way for an extension property to have a backing field. That's why initializers are not allowed for extension properties. You can define their behavior only by explicitly providing getters and setters. For example:

```
data class House(val streetName: String)

// Doesn't compile because there is no getter and setter
// var House.number = 1
// Error: Initializers are not allowed for extension properties

// Compiles successfully
val houseNumbers = mutableMapOf<House, Int>()
var House.number: Int
    get() = houseNumbers[this] ?: 1
    set(value) {
        println("Setting house number for ${this.streetName} to $value")
        houseNumbers[this] = value
    }

fun main() {
    val house = House("Maple Street")

    // Shows the default
    println("Default number: ${house.number} ${house.streetName}")
    // Default number: 1 Maple Street

    house.number = 99
    // Setting house number for Maple Street to 99

    // Shows the updated number
    println("Updated number: ${house.number} ${house.streetName}")
    // Updated number: 99 Maple Street
}
```

In this example, the getter uses the Elvis operator to return the house number if it exists in the `houseNumbers` map or `1`. To learn more about how to write getters and setters, see Custom getters and setters.

Companion object extensions

If a class defines a companion object, you can also define extension functions and properties for the companion object. Just like regular members of the companion object, you can call them using only the class name as the qualifier. The compiler names the companion object `Companion` by default:

```
class Logger {
    companion object { }
}

fun Logger.Companion.logStartupMessage() {
    println("Application started.")
}

fun main() {
    Logger.logStartupMessage()
    // Application started.
}
```

Declaring extensions as members

You can declare extensions for one class inside another. Extensions like this have multiple *implicit receivers*. An implicit receiver is an object whose members you can access without qualifying them with `this`:

- The class where you declare the extension is the *dispatch receiver*.
- The extension function's receiver type is the *extension receiver*.

Consider this example where the `Connection` class has an extension function for the `Host` class called `printConnectionString()`:

```
class Host(val hostname: String) {
    fun printHostname() { print(hostname) }
}

class Connection(val host: Host, val port: Int) {
    fun printPort() { print(port) }

    // Host is the extension receiver
    fun Host.printConnectionString() {
        // Calls Host.printHostname()
        printHostname()
        print(":")
        // Calls Connection.printPort()
        // Connection is the dispatch receiver
        printPort()
    }

    fun connect() {
        /*...*/
        // Calls the extension function
        host.printConnectionString()
    }
}

fun main() {
    Connection(Host("kotl.in"), 443).connect()
    // kotl.in:443

    // Triggers an error because the extension function isn't available outside Connection
    // Host("kotl.in").printConnectionString()
    // Unresolved reference 'printConnectionString'.
}
```

This example declares the `printConnectionString()` function inside the `Connection` class, so the `Connection` class is the dispatch receiver. The extension function's receiver type is the `Host` class, so the `Host` class is the extension receiver.

If the dispatch receiver and the extension receiver have members with the same name, the extension receiver's member takes precedence. To access the dispatch receiver explicitly, use the qualified `this` syntax:

```
class Connection {
    fun Host.getConnectionString() {
        // Calls Host.toString()
        toString()
        // Calls Connection.toString()
        this@Connection.toString()
    }
}
```

Overriding member extensions

You can declare member extensions as `open` and override them in subclasses, which is useful when you want to customize the extension's behavior for each subclass. The compiler handles each receiver type differently:

Receiver type	Resolution time	Dispatch type
Dispatch receiver	Runtime	Virtual
Extension receiver	Compile time	Static

Consider this example, where the `User` class is `open` and the `Admin` class inherits from it. The `NotificationSender` class defines `sendNotification()` extension functions for both `User` and `Admin` classes, and the `SpecialNotificationSender` class overrides them:

```
open class User

class Admin : User()

open class NotificationSender {
    open fun User.sendNotification() {
        println("Sending user notification from normal sender")
    }

    open fun Admin.sendNotification() {
        println("Sending admin notification from normal sender")
    }

    fun notify(user: User) {
        user.sendNotification()
    }
}

class SpecialNotificationSender : NotificationSender() {
    override fun User.sendNotification() {
        println("Sending user notification from special sender")
    }

    override fun Admin.sendNotification() {
        println("Sending admin notification from special sender")
    }
}

fun main() {
    // Dispatch receiver is NotificationSender
    // Extension receiver is User
    // Resolves to User.sendNotification() in NotificationSender
    NotificationSender().notify(User())
    // Sending user notification from normal sender

    // Dispatch receiver is SpecialNotificationSender
    // Extension receiver is User
    // Resolves to User.sendNotification() in SpecialNotificationSender
    SpecialNotificationSender().notify(User())
    // Sending user notification from special sender

    // Dispatch receiver is SpecialNotificationSender
    // Extension receiver is User NOT Admin
    // The notify() function declares user as type User
    // Statically resolves to User.sendNotification() in SpecialNotificationSender
    SpecialNotificationSender().notify(Admin())
    // Sending user notification from special sender
}
```

The dispatch receiver is resolved at runtime using virtual dispatch, which makes the behavior in the `main()` function easier to follow. What may surprise you is that when you call the `notify()` function on an `Admin` instance, the compiler chooses the extension based on the declared type: `user: User`, because it resolves the extension receiver statically.

Extensions and visibility modifiers

Extensions use the same visibility modifiers as regular functions declared in the same scope, including extensions declared as members of other classes.

For example, an extension declared at the top level of a file can access other `private` top-level declarations in the same file:

```
// File: StringUtils.kt

private fun removeWhitespace(input: String): String {
    return input.replace("\\s".toRegex(), "")
}

fun String.cleaned(): String {
    return removeWhitespace(this)
}

fun main() {
    val rawEmail = "  user @example. com  "
    val cleaned = rawEmail.cleaned()
    println("Raw:     '$rawEmail'")
    // Raw:     '  user @example. com  '
    println("Cleaned: '$cleaned'")
    // Cleaned: 'user@example.com'
    println("Looks like an email: ${cleaned.contains("@") && cleaned.contains(".")}")
    // Looks like an email: true
}
```

And if an extension is declared outside its receiver type, it can't access the receiver's `private` or `protected` members:

```
class User(private val password: String) {
    fun isLoggedIn(): Boolean = true
    fun passwordLength(): Int = password.length
}

// Extension declared outside the class
fun User.isSecure(): Boolean {
    // Can't access password because it's private:
    // return password.length >= 8

    // Instead, we rely on public members:
    return passwordLength() >= 8 && isLoggedIn()
}

fun main() {
    val user = User("supersecret")
    println("Is user secure: ${user.isSecure()}")
    // Is user secure: true
}
```

If an extension is marked as `internal`, it's only accessible within its module:

```
// Networking module
// JsonParser.kt
internal fun String.parseJson(): Map<String, Any> {
    return mapOf("fakeKey" to "fakeValue")
}
```

Scope of extensions

In most cases, you define extensions on the top level, directly under packages:

```
package org.example.declarations

fun List<String>.getLongestString() { /*...*/}
```

To use an extension outside its declaring package, import it at the call site:

```
package org.example.usage

import org.example.declarations.getLongestString

fun main() {
    val list = listOf("red", "green", "blue")
    list.getLongestString()
}
```

For more information, see Imports.

Interfaces

Interfaces in Kotlin can contain declarations of abstract methods, as well as method implementations. What makes them different from abstract classes is that interfaces cannot store state. They can have properties, but these need to be abstract or provide accessor implementations.

An interface is defined using the keyword `interface`:

```
interface MyInterface {
    fun bar()
    fun foo() {
      // optional body
    }
}
```

Implementing interfaces

A class or object can implement one or more interfaces:

```
class Child : MyInterface {
    override fun bar() {
        // body
    }
}
```

Properties in interfaces

You can declare properties in interfaces. A property declared in an interface can either be abstract or provide implementations for accessors. Properties declared in interfaces can't have backing fields, and therefore accessors declared in interfaces can't reference them:

```
interface MyInterface {
    val prop: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

Interfaces Inheritance

An interface can derive from other interfaces, meaning it can both provide implementations for their members and declare new functions and properties. Quite naturally, classes implementing such an interface are only required to define the missing implementations:

```
interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String get() = "$firstName $lastName"
}

data class Employee(
    // implementing 'name' is not required
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person
```

Resolving overriding conflicts

When you declare many types in your supertype list, you may inherit more than one implementation of the same method:

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }

    override fun bar() {
        super<B>.bar()
    }
}
```

Interfaces *A* and *B* both declare functions *foo()* and *bar()*. Both of them implement *foo()*, but only *B* implements *bar()* (*bar()* is not marked as abstract in *A*, because this is the default for interfaces if the function has no body). Now, if you derive a concrete class *C* from *A*, you have to override *bar()* and provide an implementation.

However, if you derive *D* from *A* and *B*, you need to implement all the methods that you have inherited from multiple interfaces, and you need to specify how exactly *D* should implement them. This rule applies both to methods for which you've inherited a single implementation (*bar()*) and to those for which you've inherited multiple implementations (*foo()*).

JVM default method generation for interface functions

On the JVM, functions declared in interfaces are compiled to default methods. You can control this behavior using the `-jvm-default` compiler option with the following values:

- `enable` (default): generates default implementations in interfaces and includes bridge functions in subclasses and `DefaultImpls` classes. Use this mode to maintain binary compatibility with older Kotlin versions.
- `no-compatibility` : generates only default implementations in interfaces. This mode skips compatibility bridges and `DefaultImpls` classes, making it suitable for new Kotlin code.
- `disable` : skips default methods and generates only compatibility bridges and `DefaultImpls` classes.

To configure the `-jvm-default` compiler option, set the `jvmDefault` property in your Gradle Kotlin DSL:

```
kotlin {
    compilerOptions {
        jvmDefault = JvmDefaultMode.NO_COMPATIBILITY
    }
}
```

Delegation

The Delegation pattern has proven to be a good alternative to implementation inheritance, and Kotlin supports it natively requiring zero boilerplate code.

A class `Derived` can implement an interface `Base` by delegating all of its public members to a specified object:

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val base = BaseImpl(10)
    Derived(base).print()
}
```

The `by` -clause in the supertype list for `Derived` indicates that `b` will be stored internally in objects of `Derived` and the compiler will generate all the methods of `Base` that forward to `b`.

Overriding a member of an interface implemented by delegation

Overrides work as you expect: the compiler will use your `override` implementations instead of those in the delegate object. If you want to add `override fun printMessage() { print("abc") }` to `Derived`, the program would print *abc* instead of *10* when `printMessage` is called:

```
interface Base {
    fun printMessage()
    fun printMessageLine()
}

class BaseImpl(val x: Int) : Base {
    override fun printMessage() { print(x) }
    override fun printMessageLine() { println(x) }
}

class Derived(b: Base) : Base by b {
    override fun printMessage() { print("abc") }
}

fun main() {
    val base = BaseImpl(10)
    Derived(base).printMessage()
    Derived(base).printMessageLine()
}
```

Note, however, that members overridden in this way do not get called from the members of the delegate object, which can only access its own implementations of the interface members:

```
interface Base {
    val message: String
    fun print()
}

class BaseImpl(x: Int) : Base {
    override val message = "BaseImpl: x = $x"
    override fun print() { println(message) }
}

class Derived(b: Base) : Base by b {
    // This property is not accessed from b's implementation of `print`
    override val message = "Message of Derived"
}

fun main() {
    val b = BaseImpl(10)
    val derived = Derived(b)
    derived.print()
    println(derived.message)
}
```

Learn more about delegated properties.

Inheritance

> Before creating an inheritance hierarchy with classes, consider using abstract classes or interfaces. You can inherit from abstract classes and interfaces by default. They are designed so that other classes can inherit their members and implement them.

All classes in Kotlin have a common superclass, `Any`, which is the default superclass for a class with no supertypes declared:

```
class Example // Implicitly inherits from Any
```

`Any` has three methods: `equals()`, `hashCode()`, and `toString()`. Thus, these methods are defined for all Kotlin classes.

By default, Kotlin classes are final – they can't be inherited. To make a class inheritable, mark it with the `open` keyword:

```
open class Base // Class is open for inheritance
```

For more information, see Open keyword.

To declare an explicit supertype, place the type after a colon in the class header:

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

If the derived class has a primary constructor, the base class can (and must) be initialized in that primary constructor according to its parameters.

If the derived class has no primary constructor, then each secondary constructor has to initialize the base type using the `super` keyword or it has to delegate to another constructor which does. Note that in this case, different secondary constructors can call different constructors of the base type:

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

Open keyword

In Kotlin, the `open` keyword indicates that a class or a member (function or property) can be overridden in subclasses. By default, Kotlin classes and their members are *final*, meaning they cannot be inherited from (for classes) or overridden (for members) unless you explicitly mark them as `open`:

```
// Base class with the open keyword to allow inheritance
open class Person(
    val name: String
) {
    // Open function that can be overridden in a subclass
    open fun introduce() {
        println("Hello, my name is $name.")
    }
}

// Subclass inheriting from Person and overriding the introduce() function
class Student(
    name: String,
    val school: String
) : Person(name) {
    override fun introduce() {
        println("Hi, I'm $name, and I study at $school.")
    }
}
```

If you override a member of a base class, the overriding member is also open by default. If you want to change this and forbid the subclasses of your class from overriding your implementation, you can explicitly mark the overriding member as `final`:

```
// Base class with the open keyword to allow inheritance
open class Person(val name: String) {
    // Open function that can be overridden in a subclass
    open fun introduce() {
        println("Hello, my name is $name.")
    }
}

//  A subclass that inherits from Person and overrides the introduce() function
class Student(name: String, val school: String) : Person(name) {
    // The final keyword prevents further overrides in subclasses
    final override fun introduce() {
        println("Hi, I'm $name, and I study at $school.")
    }
}
```

Overriding methods

Kotlin requires explicit modifiers for overridable members and overrides:

```
open class Shape {
    open fun draw() { /*...*/ }
    fun fill() { /*...*/ }
}

class Circle() : Shape() {
    override fun draw() { /*...*/ }
}
```

The `override` modifier is required for `Circle.draw()`. If it's missing, the compiler will complain. If there is no `open` modifier on a function, like `Shape.fill()`, declaring a method with the same signature in a subclass is not allowed, either with `override` or without it. The `open` modifier has no effect when added to members of a final class – a class without an `open` modifier.

A member marked `override` is itself open, so it may be overridden in subclasses. If you want to prohibit re-overriding, use `final`:

```
open class Rectangle() : Shape() {
    final override fun draw() { /*...*/ }
}
```

Overriding properties

The overriding mechanism works on properties in the same way that it does on methods. Properties declared on a superclass that are then redeclared on a derived class must be prefaced with `override`, and they must have a compatible type. Each declared property can be overridden by a property with an initializer or by a property with a `get` method:

```
open class Shape {
    open val vertexCount: Int = 0
}

class Rectangle : Shape() {
    override val vertexCount = 4
}
```

You can also override a `val` property with a `var` property, but not vice versa. This is allowed because a `val` property essentially declares a `get` method, and overriding it as a `var` additionally declares a `set` method in the derived class.

Note that you can use the `override` keyword as part of the property declaration in a primary constructor:

```
interface Shape {
    val vertexCount: Int
}

class Rectangle(override val vertexCount: Int = 4) : Shape // Always has 4 vertices

class Polygon : Shape {
    override var vertexCount: Int = 0  // Can be set to any number later
}
```

Derived class initialization order

During the construction of a new instance of a derived class, the base class initialization is done as the first step (preceded only by evaluation of the arguments for the base class constructor), which means that it happens before the initialization logic of the derived class is run.

```
open class Base(val name: String) {

    init { println("Initializing a base class") }

    open val size: Int =
        name.length.also { println("Initializing size in the base class: $it") }
}

class Derived(
    name: String,
    val lastName: String,
) : Base(name.replaceFirstChar { it.uppercase() }.also { println("Argument for the base class: $it")
}) {

    init { println("Initializing a derived class") }

    override val size: Int =
        (super.size + lastName.length).also { println("Initializing size in the derived class: $it") }
}

fun main() {
    println("Constructing the derived class(\"hello\", \"world\")")
    Derived("hello", "world")
}
```

This means that when the base class constructor is executed, the properties declared or overridden in the derived class have not yet been initialized. Using any of those properties in the base class initialization logic (either directly or indirectly through another overridden `open` member implementation) may lead to incorrect behavior or a runtime failure. When designing a base class, you should therefore avoid using `open` members in the constructors, property initializers, or `init` blocks.

Calling the superclass implementation

Code in a derived class can call its superclass functions and property accessor implementations using the `super` keyword:

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

class FilledRectangle : Rectangle() {
    override fun draw() {
        super.draw()
        println("Filling the rectangle")
    }

    val fillColor: String get() = super.borderColor
}
```

Inside an inner class, accessing the superclass of the outer class is done using the `super` keyword qualified with the outer class name: `super@Outer`:

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

class FilledRectangle: Rectangle() {
    override fun draw() {
        val filler = Filler()
        filler.drawAndFill()
    }

    inner class Filler {
        fun fill() { println("Filling") }
        fun drawAndFill() {
            super@FilledRectangle.draw() // Calls Rectangle's implementation of draw()
            fill()
            println("Drawn a filled rectangle with color ${super@FilledRectangle.borderColor}") //
Uses Rectangle's implementation of borderColor's get()
        }
    }
}

fun main() {
    val fr = FilledRectangle()
        fr.draw()
}
```

Overriding rules

In Kotlin, implementation inheritance is regulated by the following rule: if a class inherits multiple implementations of the same member from its immediate superclasses, it must override this member and provide its own implementation (perhaps, using one of the inherited ones).

To denote the supertype from which the inherited implementation is taken, use `super` qualified by the supertype name in angle brackets, such as `super<Base>`:

```
open class Rectangle {
    open fun draw() { /* ... */ }
}

interface Polygon {
    fun draw() { /* ... */ } // interface members are 'open' by default
}

class Square() : Rectangle(), Polygon {
    // The compiler requires draw() to be overridden:
    override fun draw() {
        super<Rectangle>.draw() // call to Rectangle.draw()
        super<Polygon>.draw() // call to Polygon.draw()
    }
}
```

It's fine to inherit from both `Rectangle` and `Polygon`, but both of them have their implementations of `draw()`, so you need to override `draw()` in `Square` and provide a separate implementation for it to eliminate the ambiguity.

Object declarations and expressions

In Kotlin, objects allow you to define a class and create an instance of it in a single step. This is useful when you need either a reusable singleton instance or a one-time object. To handle these scenarios, Kotlin provides two key approaches: *object declarations* for creating singletons and *object expressions* for creating anonymous, one-time objects.

> A singleton ensures that a class has only one instance and provides a global point of access to it.

Object declarations and object expressions are best used for scenarios when:

- **Using singletons for shared resources:** You need to ensure that only one instance of a class exists throughout the application. For example, managing a database connection pool.
- **Creating factory methods:** You need a convenient way to create instances efficiently. Companion objects allow you to define class-level functions and properties tied to a class, simplifying the creation and management of these instances.
- **Modifying existing class behavior temporarily:** You want to modify the behavior of an existing class without the need to create a new subclass. For example, adding temporary functionality to an object for a specific operation.
- **Type-safe design is required:** You require one-time implementations of interfaces or abstract classes using object expressions. This can be useful for scenarios like a button click handler.

Object declarations

You can create single instances of objects in Kotlin using object declarations, which always have a name following the `object` keyword. This allows you to define a class and create an instance of it in a single step, which is useful for implementing singletons:

```
// Declares a Singleton object to manage data providers
object DataProviderManager {
    private val providers = mutableListOf<DataProvider>()

    // Registers a new data provider
    fun registerDataProvider(provider: DataProvider) {
        providers.add(provider)
    }

    // Retrieves all registered data providers
    val allDataProviders: Collection<DataProvider>
        get() = providers
}

// Example data provider interface
interface DataProvider {
    fun provideData(): String
}

// Example data provider implementation
class ExampleDataProvider : DataProvider {
    override fun provideData(): String {
        return "Example data"
    }
}

fun main() {
    // Creates an instance of ExampleDataProvider
    val exampleProvider = ExampleDataProvider()

    // To refer to the object, use its name directly
    DataProviderManager.registerDataProvider(exampleProvider)

    // Retrieves and prints all data providers
    println(DataProviderManager.allDataProviders.map { it.provideData() })
    // [Example data]
}
```

The initialization of an object declaration is thread-safe and done on first access.

To refer to the `object`, use its name directly:

```
DataProviderManager.registerDataProvider(exampleProvider)
```

Object declarations can also have supertypes, similar to how anonymous objects can inherit from existing classes or implement interfaces:

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ... }

    override fun mouseEntered(e: MouseEvent) { ... }
}
```

Like variable declarations, object declarations are not expressions, so they cannot be used on the right-hand side of an assignment statement:

```
// Syntax error: An object expression cannot bind a name.
val myObject = object MySingleton {
    val name = "Singleton"
}
```

Object declarations cannot be local, which means they cannot be nested directly inside a function. However, they can be nested within other object declarations or non-inner classes.

Data objects

When printing a plain object declaration in Kotlin, the string representation contains both its name and the hash of the `object`:

```
object MyObject

fun main() {
    println(MyObject)
    // MyObject@hashcode
}
```

However, by marking an object declaration with the `data` modifier, you can instruct the compiler to return the actual name of the object when calling `toString()`, the same way it works for data classes:

```
data object MyDataObject {
    val number: Int = 3
}

fun main() {
    println(MyDataObject)
    // MyDataObject
}
```

Additionally, the compiler generates several functions for your `data object`:

- `toString()` returns the name of the data object
- `equals()` / `hashCode()` enables equality checks and hash-based collections

> You can't provide a custom `equals` or `hashCode` implementation for a `data object`.

The `equals()` function for a `data object` ensures that all objects that have the type of your `data object` are considered equal. In most cases, you will only have a single instance of your `data object` at runtime, since a `data object` declares a singleton. However, in the edge case where another object of the same type is generated at runtime (for example, by using platform reflection with `java.lang.reflect` or a JVM serialization library that uses this API under the hood), this ensures that the objects are treated as being equal.

> Make sure that you only compare `data objects` structurally (using the `==` operator) and never by reference (using the `===` operator). This helps you to avoid pitfalls when more than one instance of a data object exists at runtime.

```
import java.lang.reflect.Constructor

data object MySingleton

fun main() {
    val evilTwin = createInstanceViaReflection()

    println(MySingleton)
    // MySingleton

    println(evilTwin)
    // MySingleton

    // Even when a library forcefully creates a second instance of MySingleton,
    // its equals() function returns true:
    println(MySingleton == evilTwin)
    // true

    // Don't compare data objects using ===
    println(MySingleton === evilTwin)
    // false
}

fun createInstanceViaReflection(): MySingleton {
    // Kotlin reflection does not permit the instantiation of data objects.
    // This creates a new MySingleton instance "by force" (using Java platform reflection)
    // Don't do this yourself!
    return (MySingleton.javaClass.declaredConstructors[0].apply { isAccessible = true } as
Constructor<MySingleton>).newInstance()
}
```

The generated `hashCode()` function has a behavior that is consistent with the `equals()` function, so that all runtime instances of a `data object` have the same hash code.

Differences between data objects and data classes

While `data object` and `data class` declarations are often used together and have some similarities, there are some functions that are not generated for a `data object`:

- No `copy()` function. Because a `data object` declaration is intended to be used as singletons, no `copy()` function is generated. Singletons restrict the instantiation of a class to a single instance, which would be violated by allowing copies of the instance to be created.
- No `componentN()` function. Unlike a `data class`, a `data object` does not have any data properties. Since attempting to destructure such an object without data properties wouldn't make sense, no `componentN()` functions are generated.

Use data objects with sealed hierarchies

Data object declarations are particularly useful for sealed hierarchies like sealed classes or sealed interfaces. They allow you to maintain symmetry with any data classes you may have defined alongside the object.

In this example, declaring `EndOfFile` as a `data object` instead of a plain `object` means that it will get the `toString()` function without the need to override it manually:

```
sealed interface ReadResult
data class Number(val number: Int) : ReadResult
data class Text(val text: String) : ReadResult
data object EndOfFile : ReadResult

fun main() {
    println(Number(7))
    // Number(number=7)
    println(EndOfFile)
    // EndOfFile
}
```

Companion objects

Companion objects allow you to define class-level functions and properties. This makes it easy to create factory methods, hold constants, and access shared utilities.

An object declaration inside a class can be marked with the `companion` keyword:

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

Members of the `companion object` can be called simply by using the class name as the qualifier:

```
class User(val name: String) {
    // Defines a companion object that acts as a factory for creating User instances
    companion object Factory {
        fun create(name: String): User = User(name)
    }
}

fun main(){
    // Calls the companion object's factory method using the class name as the qualifier.
    // Creates a new User instance
    val userInstance = User.create("John Doe")
    println(userInstance.name)
    // John Doe
}
```

The name of the `companion object` can be omitted, in which case the name `Companion` is used:

```
class User(val name: String) {
    // Defines a companion object without a name
    companion object { }
}

// Accesses the companion object
val companionUser = User.Companion
```

Class members can access `private` members of their corresponding `companion object`:

```
class User(val name: String) {
    companion object {
        private val defaultGreeting = "Hello"
    }

    fun sayHi() {
        println(defaultGreeting)
    }
}
User("Nick").sayHi()
// Hello
```

When a class name is used by itself, it acts as a reference to the companion object of the class, regardless of whether the companion object is named or not:

```
class User1 {
    // Defines a named companion object
    companion object Named {
        fun show(): String = "User1's Named Companion Object"
    }
}

// References the companion object of User1 using the class name
val reference1 = User1

class User2 {
    // Defines an unnamed companion object
    companion object {
        fun show(): String = "User2's Companion Object"
    }
}

// References the companion object of User2 using the class name
val reference2 = User2

fun main() {
    // Calls the show() function from the companion object of User1
    println(reference1.show())
    // User1's Named Companion Object

    // Calls the show() function from the companion object of User2
    println(reference2.show())
    // User2's Companion Object
}
```

Although members of companion objects in Kotlin look like static members from other languages, they are actually instance members of the companion object, meaning they belong to the object itself. This allows companion objects to implement interfaces:

```
interface Factory<T> {
    fun create(name: String): T
}

class User(val name: String) {
    // Defines a companion object that implements the Factory interface
    companion object : Factory<User> {
        override fun create(name: String): User = User(name)
    }
}

fun main() {
    // Uses the companion object as a Factory
    val userFactory: Factory<User> = User
    val newUser = userFactory.create("Example User")
    println(newUser.name)
    // Example User
}
```

However, on the JVM, you can have members of companion objects generated as real static methods and fields if you use the `@JvmStatic` annotation. See the Java interoperability section for more detail.

Object expressions

Object expressions declare a class and create an instance of that class, but without naming either of them. These classes are useful for one-time use. They can either be created from scratch, inherit from existing classes, or implement interfaces. Instances of these classes are also called *anonymous objects* because they are defined by an expression, not a name.

Create anonymous objects from scratch

Object expressions start with the `object` keyword.

If the object doesn't extend any classes or implement interfaces, you can define an object's members directly inside curly braces after the `object` keyword:

```
fun main() {
    val helloWorld = object {
        val hello = "Hello"
        val world = "World"
        // Object expressions extend the Any class, which already has a toString() function,
        // so it must be overridden
        override fun toString() = "$hello $world"
    }

    print(helloWorld)
    // Hello World
}
```

Inherit anonymous objects from supertypes

To create an anonymous object that inherits from some type (or types), specify this type after `object` and a colon `:`. Then implement or override the members of this class as if you were inheriting from it:

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { /*...*/ }

    override fun mouseEntered(e: MouseEvent) { /*...*/ }
})
```

If a supertype has a constructor, pass the appropriate constructor parameters to it. Multiple supertypes can be specified, separated by commas, after the colon:

```
// Creates an open class BankAccount with a balance property
open class BankAccount(initialBalance: Int) {
    open val balance: Int = initialBalance
}

// Defines an interface Transaction with an execute() function
interface Transaction {
    fun execute()
}

// A function to perform a special transaction on a BankAccount
fun specialTransaction(account: BankAccount) {
    // Creates an anonymous object that inherits from the BankAccount class and implements the
Transaction interface
    // The balance of the provided account is passed to the BankAccount superclass constructor
    val temporaryAccount = object : BankAccount(account.balance), Transaction {

        override val balance = account.balance + 500  // Temporary bonus

        // Implements the execute() function from the Transaction interface
        override fun execute() {
            println("Executing special transaction. New balance is $balance.")
        }
    }
    // Executes the transaction
    temporaryAccount.execute()
}
fun main() {
    // Creates a BankAccount with an initial balance of 1000
    val myAccount = BankAccount(1000)
    // Performs a special transaction on the created account
    specialTransaction(myAccount)
    // Executing special transaction. New balance is 1500.
}
```

Use anonymous objects as return and value types

When you return an anonymous object from a local or `private` function or property, all the members of that anonymous object are accessible through that function or property:

```
class UserPreferences {
    private fun getPreferences() = object {
        val theme: String = "Dark"
        val fontSize: Int = 14
    }

    fun printPreferences() {
        val preferences = getPreferences()
        println("Theme: ${preferences.theme}, Font Size: ${preferences.fontSize}")
    }
}

fun main() {
    val userPreferences = UserPreferences()
    userPreferences.printPreferences()
    // Theme: Dark, Font Size: 14
}
```

This allows you to return an anonymous object with specific properties, offering a simple way to encapsulate data or behavior without creating a separate class.

If a function or property that returns an anonymous object has `public`, `protected`, or `internal` visibility, its actual type is:

- `Any` if the anonymous object doesn't have a declared supertype.
- The declared supertype of the anonymous object, if there is exactly one such type.
- The explicitly declared type if there is more than one declared supertype.

In all these cases, members added in the anonymous object are not accessible. Overridden members are accessible if they are declared in the actual type of the function or property. For example:

```
interface Notification {
    // Declares notifyUser() in the Notification interface
    fun notifyUser()
}

interface DetailedNotification

class NotificationManager {
    // The return type is Any. The message property is not accessible.
    // When the return type is Any, only members of the Any class are accessible.
    fun getNotification() = object {
        val message: String = "General notification"
    }

    // The return type is Notification because the anonymous object implements only one interface
    // The notifyUser() function is accessible because it is part of the Notification interface
    // The message property is not accessible because it is not declared in the Notification interface
    fun getEmailNotification() = object : Notification {
        override fun notifyUser() {
            println("Sending email notification")
        }
        val message: String = "You've got mail!"
    }

    // The return type is DetailedNotification. The notifyUser() function and the message property are
not accessible
    // Only members declared in the DetailedNotification interface are accessible
    fun getDetailedNotification(): DetailedNotification = object : Notification, DetailedNotification
{
        override fun notifyUser() {
            println("Sending detailed notification")
        }
        val message: String = "Detailed message content"
    }
}
fun main() {
    // This produces no output
    val notificationManager = NotificationManager()

    // The message property is not accessible here because the return type is Any
    // This produces no output
    val notification = notificationManager.getNotification()

    // The notifyUser() function is accessible
    // The message property is not accessible here because the return type is Notification
    val emailNotification = notificationManager.getEmailNotification()
    emailNotification.notifyUser()
    // Sending email notification

    // The notifyUser() function and message property are not accessible here because the return type
is DetailedNotification
    // This produces no output
    val detailedNotification = notificationManager.getDetailedNotification()
}
```

Access variables from anonymous objects

Code within the body of object expressions can access variables from the enclosing scope:

```
import java.awt.event.MouseAdapter
import java.awt.event.MouseEvent

fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    // MouseAdapter provides default implementations for mouse event functions
    // Simulates MouseAdapter handling mouse events
    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // The clickCount and enterCount variables are accessible within the object expression
}
```

Behavior difference between object declarations and expressions

There are differences in the initialization behavior between object declarations and object expressions:

- Object expressions are executed (and initialized) *immediately*, where they are used.
- Object declarations are initialized *lazily*, when accessed for the first time.
- A companion object is initialized when the corresponding class is loaded (resolved) that matches the semantics of a Java static initializer.

This expressions

The `this` expression refers to the current receiver (the object with which the function works). How you use `this` depends on the context:

- In a member of a class, `this` refers to the current object of that class.

  For example, in the following code, `this.name` means the `name` property of the current `Language` object:

  ```
  class Language(val name: String) {
      fun printName() {
          println(this.name)
      }
  }

  fun main() {
      val language = Language("Kotlin")
      language.printName()
      // Kotlin
  }
  ```

- In an extension function or a function literal with receiver, `this` refers to the receiver.

In the following example, `lastCharacter()` is called on the string `"Kotlin"`. Therefore, `"Kotlin"` is the receiver. Inside the extension function, `this` refers to `"Kotlin"`. This means that `this.length` is the length of `"Kotlin"`:

```
fun String.lastCharacter(): Char {
    println(this.length)
    // 6
    return this[this.length - 1]
}

fun main() {
    println("Kotlin".lastCharacter())
    // n
}
```

In simple cases, you don't need to write `this` explicitly. Kotlin resolves it from the current scope. Learn more about Implicit this.

Qualified `this`

Your code can have several receivers available at the same time when receiver scopes are nested. Kotlin can use any available receiver to access its members implicitly, but receivers from inner scopes have higher priority. To explicitly refer to a particular receiver, use a qualified `this`. It is especially useful when multiple receivers have members with the same name, and you need to access a member of an outer receiver.

To use a qualified `this`, add a label qualifier:

```
this@label
```

The label tells the compiler which receiver to access. You can use the name of an enclosing class or extension function. For example, `this@foo` refers to the receiver of an enclosing extension function named `foo`.

Access an outer class from an inner class

In an inner class, unqualified `this` refers to the inner class instance. To access the outer class object, use qualified `this`:

```
class User(val name: String) {
    inner class Age(val value: Int) {
        fun printInfo() {

            // Refers to the value property of the current Age object
            println(this.value)
            // 22

            // Refers to the name property of the outer User object
            println(this@User.name)
            // Jane Doe
        }
    }
}

fun main() {
    val user = User("Jane Doe")
    val age = user.Age(22)
    age.printInfo()
}
```

Only inner classes hold a reference to an outer class instance. Regular nested classes don't have access to `this` from the outer class.

Access a class from an extension function

If you declare an extension function inside a class, two receivers are available:

- The extension receiver: the value on which you call the extension function.
- The dispatch receiver: the current object of the class where you declare the extension function.

Use qualified `this` to specify which receiver you need:

```
class User(val name: String) {
    val prefix = "Name"

    fun String.formatName(): String {
        return "${this@User.prefix}: ${this.uppercase()}"
    }

    fun printName() {
        println(name.formatName())
    }
}

fun main() {
    val user = User("Jane Doe")
    user.printName()
    // Name: JANE DOE
}
```

Here:

- `this` refers to the extension receiver (`String`), so `this.uppercase()` converts the string to uppercase.
- `this@User` refers to the current `User` object.
- `this@User.prefix` accesses the `prefix` property of the current `User` object.

Access from a lambda

Unlike a regular lambda, a lambda with receiver introduces a receiver into its scope. As a result, `this` inside the lambda refers to the lambda's receiver, not to one from an enclosing scope. If a lambda with receiver is nested inside another receiver scope, add a label to the lambda and use a qualified `this` to refer explicitly to the lambda's receiver or to a receiver from an enclosing scope:

```
class User(val name: String) {
    fun printWithPrefix() {
        val printString: String.() -> Unit = stringLabel@ {
            println("${this@stringLabel}: ${this@User.name}")
        }

        printString("User")
    }
}

fun main() {
    val user = User("Jane Doe")
    user.printWithPrefix()
    // User: Jane Doe
}
```

Here:

- `stringLabel@` labels the lambda.
- `this@stringLabel` refers to the string receiver of the lambda.
- `this@User` refers to the current `User` object.

The label doesn't call anything or change how the lambda works. It only helps you refer to the lambda's receiver.

Access an anonymous object or its outer class

An anonymous object has its own receiver scope. Inside the object body, unqualified `this` refers to the anonymous object itself. However, since anonymous objects don't have class names, you can't use them as `this` qualifiers. Therefore, you can refer to the anonymous object only with unqualified `this`:

```
interface UserPrinter {
    fun print()
}

fun main() {
    val printer = object : UserPrinter {
        val prefix = "User"

        override fun print() {
            // `this` refers to the anonymous object
            // `this.prefix` accesses its `prefix` property
            println(this.prefix)
        }
    }
    printer.print()
    // User
}
```

If you declare an anonymous object inside another class, you can use qualified `this` to access the enclosing object:

```
interface UserPrinter {
    fun print()
}

class User(val name: String) {
    fun createPrinter(): UserPrinter {
        return object : UserPrinter {
            override fun print() {
                // `this@User` refers to the enclosing `User` object
                // `this@User.name` accesses its `name` property
                println(this@User.name)
            }
        }
    }
}

fun main() {
    val printer = User("Jane Doe").createPrinter()
    printer.print()
    // Jane Doe
}
```

Implicit `this`

When you call a member function on `this`, you can omit the `this.` qualifier. However, if another callable with the same name is available in a closer lexical scope, Kotlin resolves the unqualified call to that callable instead of the member function. To explicitly call the member function, use the `this.` qualifier:

```
fun main() {
    class A {
        fun printLine() {
            println("Member function")
        }

        fun invokePrintLine() {
            fun printLine() {
                println("Local function")
            }

            printLine()
            // Local function

            this.printLine()
            // Member function
        }
    }

    A().invokePrintLine()
}
```

Sealed classes and interfaces

Sealed classes and interfaces provide controlled inheritance of your class hierarchies. All direct subclasses of a sealed class are known at compile time. No other subclasses may appear outside the module and package within which the sealed class is defined. The same logic applies to sealed interfaces and their implementations: once a module with a sealed interface is compiled, no new implementations can be created.

Direct subclasses are classes that immediately inherit from their superclass.

Indirect subclasses are classes that inherit from more than one level down from their superclass.

When you combine sealed classes and interfaces with the `when` expression, you can cover the behavior of all possible subclasses and ensure that no new subclasses are created to affect your code adversely.

Sealed classes are best used for scenarios when:

- **Limited class inheritance is desired:** You have a predefined, finite set of subclasses that extend a class, all of which are known at compile time.
- **Type-safe design is required:** Safety and pattern matching are crucial in your project. Particularly for state management or handling complex conditional logic. For an example, check out Use sealed classes with when expressions.
- **Working with closed APIs:** You want robust and maintainable public APIs for libraries that ensure that third-party clients use the APIs as intended.

For more detailed practical applications, see Use case scenarios.

Java 15 introduced a similar concept, where sealed classes use the `sealed` keyword along with the `permits` clause to define restricted hierarchies.

Declare a sealed class or interface

To declare a sealed class or interface, use the `sealed` modifier:

```
// Create a sealed interface
sealed interface Error

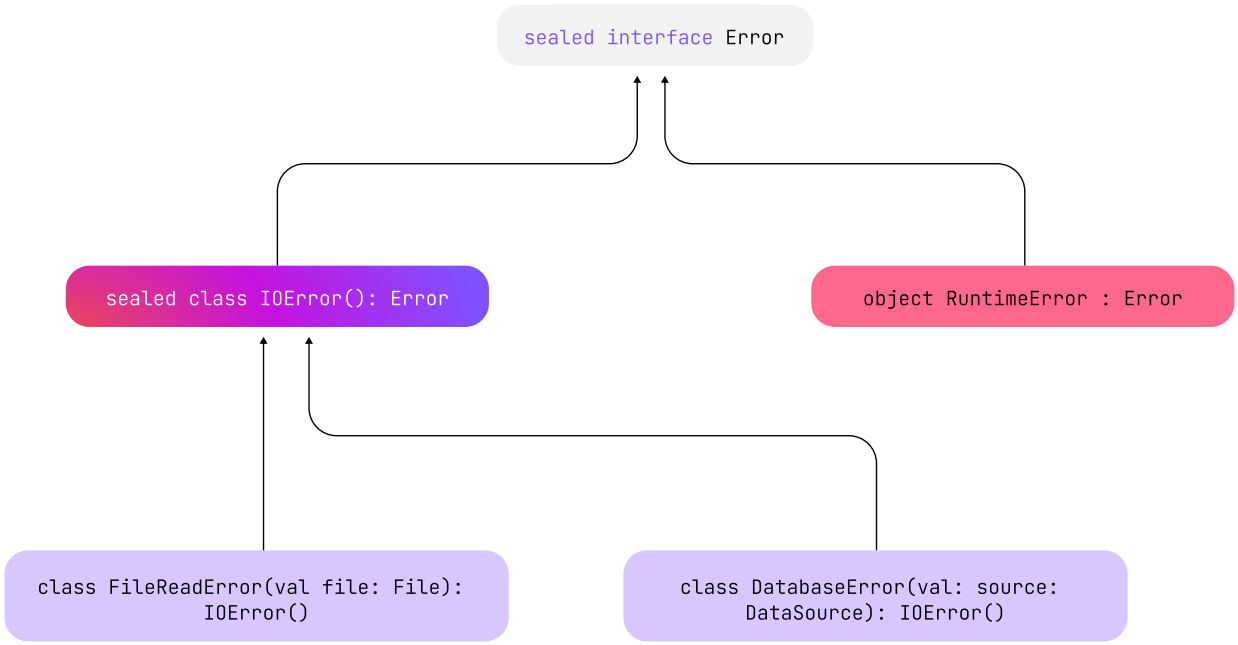
// Create a sealed class that implements sealed interface Error
sealed class IOError(): Error

// Define subclasses that extend sealed class 'IOError'
class FileReadError(val file: File): IOError()
class DatabaseError(val source: DataSource): IOError()

// Create a singleton object implementing the 'Error' sealed interface
object RuntimeError : Error
```

This example could represent a library's API that contains error classes to let library users handle errors that it can throw. If the hierarchy of such error classes includes interfaces or abstract classes visible in the public API, then nothing prevents other developers from implementing or extending them in the client code. Since the library doesn't know about errors declared outside of it, it can't treat them consistently with its own classes. However, with a **sealed** hierarchy of error classes, library authors can be sure that they know all the possible error types and that other error types can't appear later.

The hierarchy of the example looks like this:



Hierarchy illustration of sealed classes and interfaces

Constructors

A sealed class itself is always an abstract class, and as a result, can't be instantiated directly. However, it may contain or inherit constructors. These constructors aren't for creating instances of the sealed class itself but for its subclasses. Consider the following example with a sealed class called `Error` and its several subclasses, which we instantiate:

```
sealed class Error(val message: String) {
    class NetworkError : Error("Network failure")
    class DatabaseError : Error("Database cannot be reached")
    class UnknownError : Error("An unknown error has occurred")
}

fun main() {
    val errors = listOf(Error.NetworkError(), Error.DatabaseError(), Error.UnknownError())
    errors.forEach { println(it.message) }
}
// Network failure
// Database cannot be reached
// An unknown error has occurred
```

You can use `enum` classes within your sealed classes to use enum constants to represent states and provide additional detail. Each enum constant exists only as a **single** instance, while subclasses of a sealed class may have **multiple** instances. In the example, the `sealed class Error` along with its several subclasses, employs an `enum` to denote error severity. Each subclass constructor initializes the `severity` and can alter its state:

```
enum class ErrorSeverity { MINOR, MAJOR, CRITICAL }

sealed class Error(val severity: ErrorSeverity) {
    class FileReadError(val file: File): Error(ErrorSeverity.MAJOR)
    class DatabaseError(val source: DataSource): Error(ErrorSeverity.CRITICAL)
    object RuntimeError : Error(ErrorSeverity.CRITICAL)
    // Additional error types can be added here
}
```

Constructors of sealed classes can have one of two visibilities: `protected` (by default) or `private`:

```
sealed class IOError {
    // A sealed class constructor has protected visibility by default. It's visible inside this class
and its subclasses
    constructor() { /*...*/ }

    // Private constructor, visible inside this class only.
    // Using a private constructor in a sealed class allows for even stricter control over
instantiation, enabling specific initialization procedures within the class.
    private constructor(description: String): this() { /*...*/ }

    // This will raise an error because public and internal constructors are not allowed in sealed
classes
    // public constructor(code: Int): this() {}
}
```

Inheritance

Direct subclasses of sealed classes and interfaces must be declared in the same package. They may be top-level or nested inside any number of other named classes, named interfaces, or named objects. Subclasses can have any visibility as long as they are compatible with normal inheritance rules in Kotlin, including those for overriding properties.

Subclasses of sealed classes must have a properly qualified name. They can't be local or anonymous objects.

`enum` classes can't extend a sealed class, or any other class. However, they can implement sealed interfaces:

```
sealed interface Error

// enum class extending the sealed interface Error
enum class ErrorType : Error {
    FILE_ERROR, DATABASE_ERROR
}
```

These restrictions don't apply to indirect subclasses. If a direct subclass of a sealed class is not marked as sealed, it can be extended in any way that its modifiers allow:

```
// Sealed interface 'Error' has implementations only in the same package and module
sealed interface Error

// Sealed class 'IOError' extends 'Error' and is extendable only within the same package
sealed class IOError(): Error

// Open class 'CustomError' extends 'Error' and can be extended anywhere it's visible
open class CustomError(): Error
```

Inheritance in multiplatform projects

There is one more inheritance restriction in multiplatform projects: direct subclasses of sealed classes must reside in the same source set. It applies to sealed classes without the expected and actual modifiers.

If a sealed class is declared as `expect` in a common source set and have `actual` implementations in platform source sets, both `expect` and `actual` versions can have subclasses in their source sets. Moreover, if you use a hierarchical structure, you can create subclasses in any source set between the `expect` and `actual` declarations.

Learn more about the hierarchical structure of multiplatform projects.

Use sealed classes with when expression

The key benefit of using sealed classes comes into play when you use them in a `when` expression. The `when` expression, used with a sealed class, allows the Kotlin compiler to check exhaustively that all possible cases are covered. In such cases, you don't need to add an `else` clause:

```
// Sealed class and its subclasses
sealed class Error {
    class FileReadError(val file: String): Error()
    class DatabaseError(val source: String): Error()
    object RuntimeError : Error()
}

// Function to log errors
fun log(e: Error) = when(e) {
    is Error.FileReadError -> println("Error while reading file ${e.file}")
    is Error.DatabaseError -> println("Error while reading from database ${e.source}")
    Error.RuntimeError -> println("Runtime error")
    // No `else` clause is required because all the cases are covered
}

// List all errors
fun main() {
    val errors = listOf(
        Error.FileReadError("example.txt"),
        Error.DatabaseError("usersDatabase"),
        Error.RuntimeError
    )

    errors.forEach { log(it) }
}
```

To reduce repetition in `when` expressions, try out context-sensitive resolution (currently in preview). This feature allows you to omit the type name when matching sealed class members if the expected type is known.

For more information, see Preview of context-sensitive resolution or the related KEEP proposal.

When using sealed classes with `when` expressions, you can also add guard conditions to include additional checks in a single branch. For more information, see Guard conditions in when expressions.

In multiplatform projects, if you have a sealed class with a `when` expression as an expected declaration in your common code, you still need an `else` branch. This is because subclasses of `actual` platform implementations may extend sealed classes that aren't known in the common code.

Use case scenarios

Let's explore some practical scenarios where sealed classes and interfaces can be particularly useful.

State management in UI applications

You can use sealed classes to represent different UI states in an application. This approach allows for structured and safe handling of UI changes. This example demonstrates how to manage various UI states:

```
sealed class UIState {
    data object Loading : UIState()
    data class Success(val data: String) : UIState()
    data class Error(val exception: Exception) : UIState()
}

fun updateUI(state: UIState) {
    when (state) {
        is UIState.Loading -> showLoadingIndicator()
        is UIState.Success -> showData(state.data)
        is UIState.Error -> showError(state.exception)
    }
}
```

Payment method handling

In practical business applications, handling various payment methods efficiently is a common requirement. You can use sealed classes with `when` expressions to implement such business logic. By representing different payment methods as subclasses of a sealed class, it establishes a clear and manageable structure for processing transactions:

```
sealed class Payment {
    data class CreditCard(val number: String, val expiryDate: String) : Payment()
    data class PayPal(val email: String) : Payment()
    data object Cash : Payment()
}

fun processPayment(payment: Payment) {
    when (payment) {
        is Payment.CreditCard → processCreditCardPayment(payment.number, payment.expiryDate)
        is Payment.PayPal → processPayPalPayment(payment.email)
        is Payment.Cash → processCashPayment()
    }
}
```

`Payment` is a sealed class that represents different payment methods in an e-commerce system: `CreditCard`, `PayPal`, and `Cash`. Each subclass can have its specific properties, like `number` and `expiryDate` for `CreditCard`, and `email` for `PayPal`.

The `processPayment()` function demonstrates how to handle different payment methods. This approach ensures that all possible payment types are considered, and the system remains flexible for new payment methods to be added in the future.

API request-response handling

You can use sealed classes and sealed interfaces to implement a user authentication system that handles API requests and responses. The user authentication system has login and logout functionalities. The `ApiRequest` sealed interface defines specific request types: `LoginRequest` for login, and `LogoutRequest` for logout operations. The sealed class, `ApiResponse`, encapsulates different response scenarios: `UserSuccess` with user data, `UserNotFound` for absent users, and `Error` for any failures. The `handleRequest` function processes these requests in a type-safe manner using a `when` expression, while `getUserById` simulates user retrieval:

```kotlin
// Import necessary modules
import io.ktor.server.application.*
import io.ktor.server.resources.*

import kotlinx.serialization.*

// Define the sealed interface for API requests using Ktor resources
@Resource("api")
sealed interface ApiRequest

@Serializable
@Resource("login")
data class LoginRequest(val username: String, val password: String) : ApiRequest


@Serializable
@Resource("logout")
object LogoutRequest : ApiRequest

// Define the ApiResponse sealed class with detailed response types
sealed class ApiResponse {
    data class UserSuccess(val user: UserData) : ApiResponse()
    data object UserNotFound : ApiResponse()
    data class Error(val message: String) : ApiResponse()
}

// User data class to be used in the success response
data class UserData(val userId: String, val name: String, val email: String)

// Function to validate user credentials (for demonstration purposes)
fun isValidUser(username: String, password: String): Boolean {
    // Some validation logic (this is just a placeholder)
    return username == "validUser" && password == "validPass"
}

// Function to handle API requests with detailed responses
fun handleRequest(request: ApiRequest): ApiResponse {
    return when (request) {
        is LoginRequest -> {
            if (isValidUser(request.username, request.password)) {
                ApiResponse.UserSuccess(UserData("userId", "userName", "userEmail"))
            } else {
                ApiResponse.Error("Invalid username or password")
            }
        }
        is LogoutRequest -> {
            // Assuming logout operation always succeeds for this example
            ApiResponse.UserSuccess(UserData("userId", "userName", "userEmail")) // For demonstration
        }
    }
}

// Function to simulate a getUserById call
fun getUserById(userId: String): ApiResponse {
    return if (userId == "validUserId") {
        ApiResponse.UserSuccess(UserData("validUserId", "John Doe", "john@example.com"))
    } else {
        ApiResponse.UserNotFound
    }
    // Error handling would also result in an Error response.
}

// Main function to demonstrate the usage
```

```
fun main() {
    val loginResponse = handleRequest(LoginRequest("user", "pass"))
    println(loginResponse)

    val logoutResponse = handleRequest(LogoutRequest)
    println(logoutResponse)

    val userResponse = getUserById("validUserId")
    println(userResponse)

    val userNotFoundResponse = getUserById("invalidId")
    println(userNotFoundResponse)
}
```

What's next

Learn more about inheritance in Kotlin.

Enum classes

The most basic use case for enum classes is the implementation of type-safe enums:

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

Each enum constant is an object. Enum constants are separated by commas.

Since each enum is an instance of the enum class, it can be initialized as:

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

Anonymous classes

Enum constants can declare their own anonymous classes with their corresponding methods, as well as with overriding base methods.

```
enum class ProtocolState {
    WAITING {
        override fun signal() = TALKING
    },

    TALKING {
        override fun signal() = WAITING
    };

    abstract fun signal(): ProtocolState
}
```

If the enum class defines any members, separate the constant definitions from the member definitions with a semicolon.

Implementing interfaces in enum classes

An enum class can implement an interface (but it cannot derive from a class), providing either a common implementation of interface members for all the entries, or separate implementations for each entry within its anonymous class. This is done by adding the interfaces you want to implement to the enum class declaration as follows:

```
import java.util.function.BinaryOperator
import java.util.function.IntBinaryOperator

enum class IntArithmetics : BinaryOperator<Int>, IntBinaryOperator {
    PLUS {
        override fun apply(t: Int, u: Int): Int = t + u
    },
    TIMES {
        override fun apply(t: Int, u: Int): Int = t * u
    };

    override fun applyAsInt(t: Int, u: Int) = apply(t, u)
}

fun main() {
    val a = 13
    val b = 31
    for (f in IntArithmetics.entries) {
        println("$f($a, $b) = ${f.apply(a, b)}")
    }
}
```

All enum classes implement the Comparable interface by default. Constants in the enum class are defined in the natural order. For more information, see Ordering.

Working with enum constants

Enum classes in Kotlin have synthetic properties and methods for listing the defined enum constants and getting an enum constant by its name. The signatures of these methods are as follows (assuming the name of the enum class is `EnumClass`):

```
EnumClass.valueOf(value: String): EnumClass
EnumClass.entries: EnumEntries<EnumClass> // specialized List<EnumClass>
```

Below is an example of them in action:

```
enum class RGB { RED, GREEN, BLUE }

fun main() {
    for (color in RGB.entries) println(color.toString()) // prints RED, GREEN, BLUE
    println("The first color is: ${RGB.valueOf("RED")}") // prints "The first color is: RED"
}
```

The `valueOf()` method throws an `IllegalArgumentException` if the specified name does not match any of the enum constants defined in the class.

Prior to the introduction of `entries` in Kotlin 1.9.0, the `values()` function was used to retrieve an array of enum constants.

Every enum constant also has properties: `name` and `ordinal`, for obtaining its name and position (starting from 0) in the enum class declaration:

```
enum class RGB { RED, GREEN, BLUE }

fun main() {
        println(RGB.RED.name)    // prints RED
    println(RGB.RED.ordinal) // prints 0
    //sampleEnd
}
```

> To reduce repetition when working with enum entries, try out context-sensitive resolution (currently in preview). This feature allows you to omit the enum class name when the expected type is known, such as in `when` expressions or when assigning to a typed variable.
>
> For more information, see Preview of context-sensitive resolution or the related KEEP proposal.

You can access the constants in an enum class in a generic way using the `enumValues<T>()` and `enumValueOf<T>()` functions. In Kotlin 2.0.0, the `enumEntries<T>()` function is introduced as a replacement for the `enumValues<T>()` function. The `enumEntries<T>()` function returns a list of all enum entries for the given enum type `T`.

The `enumValues<T>()` function is still supported, but we recommend that you use the `enumEntries<T>()` function instead because it has less performance impact. Every time you call `enumValues<T>()` a new array is created, whereas whenever you call `enumEntries<T>()` the same list is returned each time, which is far more efficient.

For example:

```
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    println(enumEntries<T>().joinToString { it.name })
}

printAllValues<RGB>()
// RED, GREEN, BLUE
```

> For more information about inline functions and reified type parameters, see Inline functions.

Inline value classes

Sometimes it is useful to wrap a value in a class to create a more domain-specific type. However, it introduces runtime overhead due to additional heap allocations. Moreover, if the wrapped type is primitive, the performance hit is significant, because primitive types are usually heavily optimized by the runtime, while their wrappers don't get any special treatment.

To solve such issues, Kotlin introduces a special kind of class called an *inline class*. Inline classes are a subset of value-based classes. They don't have an identity and can only hold values.

To declare an inline class, use the `value` modifier before the name of the class:

```
value class Password(private val s: String)
```

To declare an inline class for the JVM backend, use the `value` modifier along with the `@JvmInline` annotation before the class declaration:

```
// For JVM backends
@JvmInline
value class Password(private val s: String)
```

An inline class must have a single property initialized in the primary constructor. At runtime, instances of the inline class will be represented using this single property (see details about runtime representation below):

```
// No actual instantiation of class 'Password' happens
// At runtime 'securePassword' contains just 'String'
val securePassword = Password("Don't try this in production")
```

This is the main feature of inline classes, which inspired the name *inline*: data of the class is *inlined* into its usages (similar to how the content of inline functions is inlined to call sites).

Members

Inline classes support some functionality of regular classes. In particular, they are allowed to declare properties and functions, have an `init` block and secondary constructors:

```
@JvmInline
value class Person(private val fullName: String) {
    init {
        require(fullName.isNotEmpty()) {
            "Full name shouldn't be empty"
        }
    }

    constructor(firstName: String, lastName: String) : this("$firstName $lastName") {
        require(lastName.isNotBlank()) {
            "Last name shouldn't be empty"
        }
    }

    val length: Int
        get() = fullName.length

    fun greet() {
        println("Hello, $fullName")
    }
}

fun main() {
    val name1 = Person("Kotlin", "Mascot")
    val name2 = Person("Kodee")
    name1.greet() // the `greet()` function is called as a static method
    println(name2.length) // property getter is called as a static method
}
```

Inline class properties cannot have backing fields. They can only have simple computable properties (no `lateinit` /delegated properties).

Inheritance

Inline classes are allowed to inherit from interfaces:

```
interface Printable {
    fun prettyPrint(): String
}

@JvmInline
value class Name(val s: String) : Printable {
    override fun prettyPrint(): String = "Let's $s!"
}

fun main() {
    val name = Name("Kotlin")
    println(name.prettyPrint()) // Still called as a static method
}
```

It is forbidden for inline classes to participate in a class hierarchy. This means that inline classes cannot extend other classes and are always `final`.

Representation

In generated code, the Kotlin compiler keeps a *wrapper* for each inline class. Inline class instances can be represented at runtime either as wrappers or as the underlying type. This is similar to how `Int` can be represented either as a primitive `int` or as the wrapper `Integer`.

The Kotlin compiler will prefer using underlying types instead of wrappers to produce the most performant and optimized code. However, sometimes it is necessary to keep wrappers around. As a rule of thumb, inline classes are boxed whenever they are used as another type.

```
interface I

@JvmInline
value class Foo(val i: Int) : I

fun asInline(f: Foo) {}
fun <T> asGeneric(x: T) {}
fun asInterface(i: I) {}
fun asNullable(i: Foo?) {}

fun <T> id(x: T): T = x

fun main() {
    val f = Foo(42)

    asInline(f)    // unboxed: used as Foo itself
    asGeneric(f)   // boxed: used as generic type T
    asInterface(f) // boxed: used as type I
    asNullable(f)  // boxed: used as Foo?, which is different from Foo

    // below, 'f' first is boxed (while being passed to 'id') and then unboxed (when returned from
'id')
    // In the end, 'c' contains unboxed representation (just '42'), as 'f'
    val c = id(f)
}
```

Because inline classes may be represented both as the underlying value and as a wrapper, referential equality is pointless for them and is therefore prohibited.

Inline classes can also have a generic type parameter as the underlying type. In this case, the compiler maps it to `Any?` or, generally, to the upper bound of the type parameter.

```
@JvmInline
value class UserId<T>(val value: T)

fun compute(s: UserId<String>) {} // compiler generates fun compute-<hashcode>(s: Any?)
```

Mangling

Since inline classes are compiled to their underlying type, it may lead to various obscure errors, for example, unexpected platform signature clashes:

```
@JvmInline
value class UInt(val x: Int)

// Represented as 'public final void compute(int x)' on the JVM
fun compute(x: Int) { }

// Also represented as 'public final void compute(int x)' on the JVM!
fun compute(x: UInt) { }
```

To mitigate such issues, functions using inline classes are *mangled* by adding some stable hashcode to the function name. Therefore, `fun compute(x: UInt)` will be represented as `public final void compute-<hashcode>(int x)`, which solves the clash problem.

Calling from Java code

You can call functions that accept inline classes from Java code. To do so, you should manually disable mangling: add the `@JvmName` annotation before the function declaration:

```
@JvmInline
value class UInt(val x: Int)

fun compute(x: Int) { }

@JvmName("computeUInt")
fun compute(x: UInt) { }
```

By default, Kotlin compiles inline classes using **unboxed representations**, which makes them difficult to access from Java. To learn how to compile inline classes into **boxed representations** that are accessible from Java, see the guide to Calling Kotlin from Java.

Inline classes vs type aliases

At first sight, inline classes seem very similar to type aliases. Indeed, both seem to introduce a new type and both will be represented as the underlying type at runtime.

However, the crucial difference is that type aliases are *assignment-compatible* with their underlying type (and with other type aliases with the same underlying type), while inline classes are not.

In other words, inline classes introduce a truly *new* type, contrary to type aliases which only introduce an alternative name (alias) for an existing type:

```
typealias NameTypeAlias = String

@JvmInline
value class NameInlineClass(val s: String)

fun acceptString(s: String) {}
fun acceptNameTypeAlias(n: NameTypeAlias) {}
fun acceptNameInlineClass(p: NameInlineClass) {}

fun main() {
    val nameAlias: NameTypeAlias = ""
    val nameInlineClass: NameInlineClass = NameInlineClass("")
    val string: String = ""

    acceptString(nameAlias) // OK: pass alias instead of underlying type
    acceptString(nameInlineClass) // Not OK: can't pass inline class instead of underlying type

    // And vice versa:
    acceptNameTypeAlias(string) // OK: pass underlying type instead of alias
    acceptNameInlineClass(string) // Not OK: can't pass underlying type instead of inline class
}
```

Inline classes and delegation

Implementation by delegation to inlined value of inlined class is allowed with interfaces:

```
interface MyInterface {
    fun bar()
    fun foo() = "foo"
}

@JvmInline
value class MyInterfaceWrapper(val myInterface: MyInterface) : MyInterface by myInterface

fun main() {
    val my = MyInterfaceWrapper(object : MyInterface {
        override fun bar() {
            // body
        }
    })
    println(my.foo()) // prints "foo"
}
```

Nested and inner classes

Classes can be nested in other classes:

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

You can also use interfaces with nesting. All combinations of classes and interfaces are possible: You can nest interfaces in classes, classes in interfaces, and interfaces in interfaces.

```
interface OuterInterface {
    class InnerClass
    interface InnerInterface
}

class OuterClass {
    class InnerClass
    interface InnerInterface
}
```

Inner classes

A nested class marked as `inner` can access the members of its outer class. Inner classes carry a reference to an object of an outer class:

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

See Qualified `this` expressions to learn about disambiguation of `this` in inner classes.

Anonymous inner classes

Anonymous inner class instances are created using an object expression:

```
window.addMouseListener(object : MouseAdapter() {

    override fun mouseClicked(e: MouseEvent) { ... }

    override fun mouseEntered(e: MouseEvent) { ... }
})
```

On the JVM, if the object is an instance of a functional Java interface (that means a Java interface with a single abstract method), you can create it using a lambda expression prefixed with the type of the interface:

```
val listener = ActionListener { println("clicked") }
```

Functional (SAM) interfaces

An interface with only one abstract member function is called a *functional interface*, or a *Single Abstract Method (SAM) interface*. The functional interface can have several non-abstract member functions but only one abstract member function.

To declare a functional interface in Kotlin, use the `fun` modifier.

```
fun interface KRunnable {
    fun invoke()
}
```

SAM conversions

For functional interfaces, you can use SAM conversions that help make your code more concise and readable by using lambda expressions.

Instead of creating a class that implements a functional interface manually, you can use a lambda expression. With a SAM conversion, Kotlin can convert any lambda expression whose signature matches the signature of the interface's single method into the code, which dynamically instantiates the interface implementation.

For example, consider the following Kotlin functional interface:

```
fun interface IntPredicate {
    fun accept(i: Int): Boolean
}
```

If you don't use a SAM conversion, you will need to write code like this:

```
// Creating an instance of a class
val isEven = object : IntPredicate {
    override fun accept(i: Int): Boolean {
        return i % 2 == 0
    }
}
```

By leveraging Kotlin's SAM conversion, you can write the following equivalent code instead:

```
// Creating an instance using lambda
val isEven = IntPredicate { it % 2 == 0 }
```

A short lambda expression replaces all the unnecessary code.

```
fun interface IntPredicate {
    fun accept(i: Int): Boolean
}

val isEven = IntPredicate { it % 2 == 0 }

fun main() {
    println("Is 7 even? - ${isEven.accept(7)}")
}
```

You can also use SAM conversions for Java interfaces.

Migration from an interface with constructor function to a functional interface

Starting from 1.6.20, Kotlin supports callable references to functional interface constructors, which adds a source-compatible way to migrate from an interface with a constructor function to a functional interface. Consider the following code:

```
interface Printer {
    fun print()
}

fun Printer(block: () -> Unit): Printer = object : Printer {
    override fun print() = block()
}
```

With callable references to functional interface constructors enabled, this code can be replaced with just a functional interface declaration:

```
fun interface Printer {
    fun print()
}
```

Its constructor will be created implicitly, and any code using the `::Printer` function reference will compile. For example:

```
documentsStorage.addPrinter(::Printer)
```

Preserve the binary compatibility by marking the legacy function `Printer` with the `@Deprecated` annotation with `DeprecationLevel.HIDDEN`:

```
@Deprecated(message = "Your message about the deprecation", level = DeprecationLevel.HIDDEN)
fun Printer(...) {...}
```

Functional interfaces vs. type aliases

You can also simply rewrite the above using a type alias for a functional type:

```
typealias IntPredicate = (i: Int) -> Boolean

val isEven: IntPredicate = { it % 2 == 0 }

fun main() {
    println("Is 7 even? - ${isEven(7)}")
}
```

However, functional interfaces and type aliases serve different purposes. Type aliases are just names for existing types – they don't create a new type, while functional interfaces do. You can provide extensions that are specific to a particular functional interface to be inapplicable for plain functions or their type aliases.

Type aliases can have only one member, while functional interfaces can have multiple non-abstract member functions and one abstract member function. Functional interfaces can also implement and extend other interfaces.

Functional interfaces are more flexible and provide more capabilities than type aliases, but they can be more costly both syntactically and at runtime because they can require conversions to a specific interface. When you choose which one to use in your code, consider your needs:

- If your API needs to accept a function (any function) with some specific parameter and return types – use a simple functional type or define a type alias to give a shorter name to the corresponding functional type.
- If your API accepts a more complex entity than a function – for example, it has non-trivial contracts and/or operations on it that can't be expressed in a functional type's signature – declare a separate functional interface for it.

Properties

In Kotlin, properties let you store and manage data without writing functions to access or change the data. You can use properties in classes, interfaces, objects, companion objects, and even outside these structures as top-level properties.

Every property has a name, a type, and an automatically generated `get()` function called a getter. You can use the getter to read the property's value. If the property is mutable, it also has a `set()` function called a setter, which allows you to change the property's value.

Getters and setters are called *accessors*.

Declaring properties

Properties can be mutable (`var`) or read-only (`val`). You can declare them as a top-level property in a `.kt` file. Think of a top-level property as a global variable that belongs to a package:

```
// File: Constants.kt
package my.app

val pi = 3.14159
var counter = 0
```

You can also declare properties inside a class, interface, or object:

```
// Class with properties
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
}

// Interface with a property
interface ContactInfo {
    val email: String
}

// Object with properties
object Company {
    var name: String = "Detective Inc."
    val country: String = "UK"
}

// Class implementing the interface
class PersonContact : ContactInfo {
    override val email: String = "sherlock@example.com"
}
```

To use a property, refer to it by its name:

```
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
}

interface ContactInfo {
    val email: String
}

object Company {
    var name: String = "Detective Inc."
    val country: String = "UK"
}

class PersonContact : ContactInfo {
    override val email: String = "sherlock@example.com"
}

fun copyAddress(address: Address): Address {
    val result = Address()
    // Accesses properties in the result instance
    result.name = address.name
    result.street = address.street
    result.city = address.city
    return result
}

fun main() {
    val sherlockAddress = Address()
    val copy = copyAddress(sherlockAddress)
    // Accesses properties in the copy instance
    println("Copied address: ${copy.name}, ${copy.street}, ${copy.city}")
    // Copied address: Holmes, Sherlock, Baker, London

    // Accesses properties in the Company object
    println("Company: ${Company.name} in ${Company.country}")
    // Company: Detective Inc. in UK

    val contact = PersonContact()
    // Access properties in the contact instance
    println("Email: ${contact.email}")
    // Email: sherlock@email.com
}
```

In Kotlin, we recommend initializing properties when you declare them to keep your code safe and easy to read. However, you can initialize them later in special cases.

Declaring the property type is optional if the compiler can infer it from the initializer or the getter's return type:

```
var initialized = 1 // The inferred type is Int
var allByDefault    // ERROR: Property must be initialized.
```

Custom getters and setters

By default, Kotlin automatically generates getters and setters. You can define your own custom accessors when you need extra logic, such as validation, formatting, or calculations based on other properties.

A custom getter runs every time the property is accessed:

```
class Rectangle(val width: Int, val height: Int) {
    val area: Int
        get() = this.width * this.height
}
fun main() {
    val rectangle = Rectangle(3, 4)
    println("Width=${rectangle.width}, height=${rectangle.height}, area=${rectangle.area}")
}
```

You can omit the type if the compiler can infer it from the getter:

```
val area get() = this.width * this.height
```

A custom setter runs every time you assign a value to the property, except during initialization. By convention, the name of the setter parameter is `value`, but you can choose a different name:

```
class Point(var x: Int, var y: Int) {
    var coordinates: String
        get() = "$x,$y"
        set(value) {
            val parts = value.split(",")
            x = parts[0].toInt()
            y = parts[1].toInt()
        }
}

fun main() {
    val location = Point(1, 2)
    println(location.coordinates)
    // 1,2

    location.coordinates = "10,20"
    println("${location.x}, ${location.y}")
    // 10, 20
}
```

Changing visibility or adding annotations

In Kotlin, you can change accessor visibility or add annotations without replacing the default implementation. You don't have to make these changes within a body `{}`.

To change the visibility of an accessor, use the modifier before the `get` or `set` keyword:

```
class BankAccount(initialBalance: Int) {
    var balance: Int = initialBalance
        // Only the class can modify the balance
        private set

    fun deposit(amount: Int) {
        if (amount > 0) balance += amount
    }

    fun withdraw(amount: Int) {
        if (amount > 0 && amount ⩽ balance) balance -= amount
    }
}

fun main() {
    val account = BankAccount(100)
    println("Initial balance: ${account.balance}")
    // 100

    account.deposit(50)
    println("After deposit: ${account.balance}")
    // 150

    account.withdraw(70)
    println("After withdrawal: ${account.balance}")
    // 80

    // account.balance = 1000
    // Error: cannot assign because setter is private
}
```

To annotate an accessor, use the annotation before the `get` or `set` keyword:

```
// Defines an annotation that can be applied to a getter
@Target(AnnotationTarget.PROPERTY_GETTER)
annotation class Inject

class Service {
    var dependency: String = "Default Service"
        // Annotates the getter
        @Inject get
}

fun main() {
    val service = Service()
    println(service.dependency)
    // Default service
    println(service::dependency.getter.annotations)
    // [@Inject()]
    println(service::dependency.setter.annotations)
    // []
}
```

This example uses reflection to show which annotations are present on the getter and setter.

Backing fields

The compiler automatically generates backing fields for properties when a value needs to be stored in memory.

For example, the compiler creates a backing field when you use the default `get()` and `set()` functions because they read and write the stored value:

```
var count = 0
```

You can access backing fields by using the `field` keyword in a custom `get()` or `set()` function. For example, you can add extra logic to a getter or setter, or trigger an additional action when a property changes.

In this example, the `score` property uses the backing field inside the `set()` function so that updating the value also triggers a log event:

```
class Scoreboard {
    var score: Int = 0
        set(value) {
            field = value
            // Adds logging when updating the value
            println("Score updated to $field")
        }
}

fun main() {
    val board = Scoreboard()
    board.score = 10
    // Score updated to 10
    board.score = 20
    // Score updated to 20
}
```

Backing fields aren't created by default for all properties because they might not need them. For example, the `isEmpty` property doesn't have a backing field because the value is calculated from the `size` property each time you access it:

```
val isEmpty: Boolean
    get() = this.size == 0
```

Explicit backing fields

Sometimes you might need more flexibility. For example, if you have an API where you want to be able to modify the property internally but not externally. In such cases, you can use an *explicit backing field*.

In the following example, the `ShoppingCart` class has an `items` property that represents everything in the shopping cart. The class exposes the `items` property as a read-only list of strings, but internally it stores the data in a mutable list with an explicit backing field:

```
class ShoppingCart {
    // Public read-only view with explicit backing field
    val items: List<String>
        field = mutableListOf()

    fun addItem(item: String) {
        items.add(item)
    }

    fun removeItem(item: String) {
        items.remove(item)
    }
}

fun main() {
    val cart = ShoppingCart()
    cart.addItem("Apple")
    cart.addItem("Banana")

    println(cart.items)
    // [Apple, Banana]

    cart.removeItem("Apple")
    println(cart.items)
    // [Banana]
}
```

In this example, the compiler infers the type of the backing field from the `mutableListOf()` call: `MutableList<String>`. You can also declare the type of the backing field explicitly:

```
val items: List<String>
    // Explicit backing field with explicit type
    field: MutableList<String> = mutableListOf()
```

In the example of the `ShoppingCart` class, the compiler smart casts the `items` property to the `MutableList<String>` type, so the class can add and remove items from the cart through the `add()` and `remove()` functions. Outside the class, the compiler uses the public property type `List<String>`, so API users can only read what's in the `items` list.

Limitations

To use explicit backing fields, their properties and the backing fields themselves must follow certain rules. Properties can have explicit backing fields only if they:

- Don't have a custom getter.
- Are read-only (`val`).
- Aren't `open` .
- Aren't a delegated property.
- Aren't compile-time constants.

In addition, the backing field type must be a subtype of the property's type and have `private` visibility.

You can work around these restrictions by using backing properties instead.

Backing properties

If explicit backing fields don't fit your use case, you can try using a coding pattern called a *backing property*.

For example, if your property needs a custom getter:

```
class UserDirectory {
    private val _users = mutableListOf(
        "sarah",
        "mike",
        "emma"
    )

    val users: List<String>
        get() = _users.sorted()

    fun addUser(username: String) {
        _users.add(username)
    }
}

fun main() {
    val directory = UserDirectory()

    directory.addUser("alex")
    println(directory.users)
    // [alex, emma, mike, sarah]
}
```

Use a leading underscore when naming backing properties to follow Kotlin coding conventions.

In this example, the `UserDirectory` class has a read-only `users` property that lists every user in the directory. The `_users` variable is the private backing property containing the real list. The getter for the public `users` property sorts the entries before returning them.

Compile-time constants

If the value of a read-only property is known at compile time, mark it as a *compile-time constant* using the `const` modifier. Compile-time constants are inlined at compile time, so each reference is replaced with its actual value. They are accessed more efficiently because no getter is called:

```
// File: AppConfig.kt
package com.example

// Compile-time constant
const val MAX_LOGIN_ATTEMPTS = 3
```

Compile-time constants must meet the following requirements:

- They must be either a top-level property, or a member of an `object` declaration or a companion object.
- They must be initialized with a value of type `String` or a primitive type.
- They can't have a custom getter.

Compile-time constants still have a backing field, so you can interact with them using reflection.

You can also use these properties in annotations:

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"

@Deprecated(SUBSYSTEM_DEPRECATED) fun processLegacyOrders() { ... }
```

Late-initialized properties and variables

Normally, you must initialize properties in the constructor. However, this isn't always convenient. For example, you might initialize properties through dependency injection or inside the setup method of a unit test.

To handle these situations, mark the property with the `lateinit` modifier:

```
public class OrderServiceTest {
    lateinit var orderService: OrderService

    @SetUp fun setup() {
        orderService = OrderService()
    }

    @Test fun processesOrderSuccessfully() {
        // Calls orderService directly without checking for null
        // or initialization
        orderService.processOrder()
    }
}
```

You can use the `lateinit` modifier on `var` properties declared as:

- Top-level properties.
- Local variables.
- Properties inside the body of a class.

For class properties:

- You can't declare them in the primary constructor.
- They must not have a custom getter or setter.

In all cases, the property or variable must be non-nullable and must not be a primitive type.

If you access a `lateinit` property before initializing it, Kotlin throws a specific exception that identifies the uninitialized property being accessed:

```
class ReportGenerator {
    lateinit var report: String

    fun printReport() {
        // Throws an exception as it's accessed before
        // initialization
        println(report)
    }
}

fun main() {
    val generator = ReportGenerator()
    generator.printReport()
    // Exception in thread "main" kotlin.UninitializedPropertyAccessException: lateinit property
report has not been initialized
}
```

To check whether a `lateinit var` has already been initialized, use the `isInitialized` property on the reference to that property:

```
class WeatherStation {
    lateinit var latestReading: String

    fun printReading() {
        // Checks whether the property is initialized
        if (this::latestReading.isInitialized) {
            println("Latest reading: $latestReading")
        } else {
            println("No reading available")
        }
    }
}

fun main() {
    val station = WeatherStation()

    station.printReading()
    // No reading available
    station.latestReading = "22°C, sunny"
    station.printReading()
    // Latest reading: 22°C, sunny
}
```

You can only use `isInitialized` on a property if you can already access that property in your code. The property must be declared in the same class, in an outer class, or as a top-level property in the same file.

Overriding properties

See Overriding properties.

Delegated properties

To reuse logic and reduce code duplication, you can delegate the responsibility of getting and setting a property to a separate object.

Delegating accessor behavior keeps the property's accessor logic centralized, making it easier to reuse. This approach is useful when implementing behaviors like:

- Computing a value lazily.
- Reading from a map by a given key.
- Accessing a database.
- Notifying a listener when a property is accessed.

You can implement these common behaviors in libraries yourself or use existing delegates provided by external libraries. For more information, see delegated properties.

Delegated properties

With some common kinds of properties, even though you can implement them manually every time you need them, it is more helpful to implement them once, add them to a library, and reuse them later. For example:

- *Lazy* properties: the value is computed only on first access.
- *Observable* properties: listeners are notified about changes to this property.
- Storing properties in a *map* instead of a separate field for each property.

To cover these (and other) cases, Kotlin supports *delegated properties*:

```
class Example {
    var p: String by Delegate()
}
```

The syntax is: `val/var <property name>: <Type> by <expression>`. The expression after `by` is a *delegate*, because the `get()` (and `set()`) that correspond to the property will be delegated to its `getValue()` and `setValue()` methods. Property delegates don't have to implement an interface, but they have to provide a `getValue()` function (and `setValue()` for `var`s).

For example:

```
import kotlin.reflect.KProperty

class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, thank you for delegating '${property.name}' to me!"
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
        println("$value has been assigned to '${property.name}' in $thisRef.")
    }
}
```

When you read from `p`, which delegates to an instance of `Delegate`, the `getValue()` function from `Delegate` is called. Its first parameter is the object you read `p` from, and the second parameter holds a description of `p` itself (for example, you can take its name).

```
val e = Example()
println(e.p)
```

This prints:

```
Example@33a17727, thank you for delegating 'p' to me!
```

Similarly, when you assign to `p`, the `setValue()` function is called. The first two parameters are the same, and the third holds the value being assigned:

```
e.p = "NEW"
```

This prints:

```
NEW has been assigned to 'p' in Example@33a17727.
```

The specification of the requirements for the delegated object can be found below.

You can declare a delegated property inside a function or code block; it doesn't have to be a member of a class. Below you can find an example.

Standard delegates

The Kotlin standard library provides factory methods for several useful kinds of delegates.

Lazy properties

`lazy()` is a function that takes a lambda and returns an instance of `Lazy<T>`, which can serve as a delegate for implementing a lazy property. The first call to `get()` executes the lambda passed to `lazy()` and remembers the result. Subsequent calls to `get()` simply return the remembered result.

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

By default, the evaluation of lazy properties is *synchronized*: the value is computed only in one thread, but all threads will see the same value. If the synchronization of the initialization delegate is not required to allow multiple threads to execute it simultaneously, pass `LazyThreadSafetyMode.PUBLICATION` as a parameter to `lazy()`.

If you're sure that the initialization will always happen in the same thread as the one where you use the property, you can use `LazyThreadSafetyMode.NONE`. It doesn't incur any thread-safety guarantees and related overhead.

Observable properties

`Delegates.observable()` takes two arguments: the initial value and a handler for modifications.

The handler is called every time you assign to the property (*after* the assignment has been performed). It has three parameters: the property being assigned to, the old value, and the new value:

```
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new →
        println("$old → $new")
    }
}

fun main() {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

If you want to intercept assignments and *veto* them, use `vetoable()` instead of `observable()`. The handler passed to `vetoable` will be called *before* the assignment of a new property value.

Delegating to another property

A property can delegate its getter and setter to another property. Such delegation is available for both top-level and class properties (member and extension). The delegate property can be:

- A top-level property
- A member or an extension property of the same class
- A member or an extension property of another class

To delegate a property to another property, use the `::` qualifier in the delegate name, for example, `this::delegate` or `MyClass::delegate`.

```
var topLevelInt: Int = 0
class ClassWithDelegate(val anotherClassInt: Int)

class MyClass(var memberInt: Int, val anotherClassInstance: ClassWithDelegate) {
    var delegatedToMember: Int by this::memberInt
    var delegatedToTopLevel: Int by ::topLevelInt

    val delegatedToAnotherClass: Int by anotherClassInstance::anotherClassInt
}
var MyClass.extDelegated: Int by ::topLevelInt
```

This may be useful, for example, when you want to rename a property in a backward-compatible way: introduce a new property, annotate the old one with the `@Deprecated` annotation, and delegate its implementation.

```
class MyClass {
   var newName: Int = 0
   @Deprecated("Use 'newName' instead", ReplaceWith("newName"))
   var oldName: Int by this::newName
}
fun main() {
   val myClass = MyClass()
   // Notification: 'oldName: Int' is deprecated.
   // Use 'newName' instead
   myClass.oldName = 42
   println(myClass.newName) // 42
}
```

Storing properties in a map

One common use case is storing the values of properties in a map. This comes up often in applications for things like parsing JSON or performing other dynamic tasks. In this case, you can use the map instance itself as the delegate for a delegated property.

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
```

In this example, the constructor takes a map:

```
val user = User(mapOf(
    "name" to "John Doe",
    "age"  to 25
))
```

Delegated properties take values from this map through string keys, which are associated with the names of properties:

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}

fun main() {
    val user = User(mapOf(
        "name" to "John Doe",
        "age"  to 25
    ))
    println(user.name) // Prints "John Doe"
    println(user.age)  // Prints 25
}
```

This also works for `var`'s properties if you use a `MutableMap` instead of a read-only `Map`:

```
class MutableUser(val map: MutableMap<String, Any?>) {
    var name: String by map
    var age: Int     by map
}
```

Local delegated properties

You can declare local variables as delegated properties. For example, you can make a local variable lazy:

```
fun example(computeFoo: () -> Foo) {
    val memoizedFoo by lazy(computeFoo)

    if (someCondition && memoizedFoo.isValid()) {
        memoizedFoo.doSomething()
    }
}
```

The `memoizedFoo` variable will be computed on first access only. If `someCondition` fails, the variable won't be computed at all.

Property delegate requirements

For a *read-only* property (`val`), a delegate should provide an operator function `getValue()` with the following parameters:

- `thisRef` must be the same type as, or a supertype of, the *property owner* (for extension properties, it should be the type being extended).
- `property` must be of type `KProperty<*>` or its supertype.

`getValue()` must return the same type as the property (or its subtype).

```
class Resource

class Owner {
    val valResource: Resource by ResourceDelegate()
}

class ResourceDelegate {
    operator fun getValue(thisRef: Owner, property: KProperty<*>): Resource {
        return Resource()
    }
}
```

For a *mutable* property (`var`), a delegate has to additionally provide an operator function `setValue()` with the following parameters:

- `thisRef` must be the same type as, or a supertype of, the *property owner* (for extension properties, it should be the type being extended).
- `property` must be of type `KProperty<*>` or its supertype.
- `value` must be of the same type as the property (or its supertype).

```
class Resource

class Owner {
    var varResource: Resource by ResourceDelegate()
}

class ResourceDelegate(private var resource: Resource = Resource()) {
    operator fun getValue(thisRef: Owner, property: KProperty<*>): Resource {
        return resource
    }
    operator fun setValue(thisRef: Owner, property: KProperty<*>, value: Any?) {
        if (value is Resource) {
            resource = value
        }
    }
}
```

`getValue()` and/or `setValue()` functions can be provided either as member functions of the delegate class or as extension functions. The latter is handy when you need to delegate a property to an object that doesn't originally provide these functions. Both of the functions need to be marked with the `operator` keyword.

You can create delegates as anonymous objects without creating new classes, by using the interfaces `ReadOnlyProperty` and `ReadWriteProperty` from the Kotlin standard library. They provide the required methods: `getValue()` is declared in `ReadOnlyProperty` ; `ReadWriteProperty` extends it and adds `setValue()` . This means you can pass a `ReadWriteProperty` whenever a `ReadOnlyProperty` is expected.

```
fun resourceDelegate(resource: Resource = Resource()): ReadWriteProperty<Any?, Resource> =
    object : ReadWriteProperty<Any?, Resource> {
        var curValue = resource
        override fun getValue(thisRef: Any?, property: KProperty<*>): Resource = curValue
        override fun setValue(thisRef: Any?, property: KProperty<*>, value: Resource) {
            curValue = value
        }
    }

val readOnlyResource: Resource by resourceDelegate()  // ReadWriteProperty as val
var readWriteResource: Resource by resourceDelegate()
```

Translation rules for delegated properties

Under the hood, the Kotlin compiler generates auxiliary properties for some kinds of delegated properties and then delegates to them.

> For optimization purposes, the compiler *does not* generate auxiliary properties in several cases. Learn about the optimization on the example of delegating to another property.

For example, for the property `prop` it generates the hidden property `prop$delegate`, and the code of the accessors simply delegates to this additional property:

```
class C {
    var prop: Type by MyDelegate()
}

// this code is generated by the compiler instead:
class C {
    private val prop$delegate = MyDelegate()
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

The Kotlin compiler provides all the necessary information about `prop` in the arguments: the first argument `this` refers to an instance of the outer class `C`, and `this::prop` is a reflection object of the `KProperty` type describing `prop` itself.

Optimized cases for delegated properties

The `$delegate` field will be omitted if a delegate is:

- A referenced property:

```
class C<Type> {
    private var impl: Type = ...
    var prop: Type by ::impl
}
```

- A named object:

```
object NamedObject {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String = ...
}

val s: String by NamedObject
```

- A final `val` property with a backing field and a default getter in the same module:

```
val impl: ReadOnlyProperty<Any?, String> = ...

class A {
    val s: String by impl
}
```

- A constant expression, enum entry, `this`, `null`. The example of `this`:

```
class A {
    operator fun getValue(thisRef: Any?, property: KProperty<*>) ...

    val s by this
}
```

Translation rules when delegating to another property

When delegating to another property, the Kotlin compiler generates immediate access to the referenced property. This means that the compiler doesn't generate the field `prop$delegate`. This optimization helps save memory.

Take the following code, for example:

```
class C<Type> {
    private var impl: Type = ...
    var prop: Type by ::impl
}
```

Property accessors of the `prop` variable invoke the `impl` variable directly, skipping the delegated property's `getValue` and `setValue` operators, and thus the `KProperty` reference object is not needed.

For the code above, the compiler generates the following code:

```
class C<Type> {
    private var impl: Type = ...

    var prop: Type
        get() = impl
        set(value) {
            impl = value
        }

    fun getProp$delegate(): Type = impl // This method is needed only for reflection
}
```

Providing a delegate

By defining the `provideDelegate` operator, you can extend the logic for creating the object to which the property implementation is delegated. If the object used on the right-hand side of `by` defines `provideDelegate` as a member or extension function, that function will be called to create the property delegate instance.

One of the possible use cases of `provideDelegate` is to check the consistency of the property upon its initialization.

For example, to check the property name before binding, you can write something like this:

```
class ResourceDelegate<T> : ReadOnlyProperty<MyUI, T> {
    override fun getValue(thisRef: MyUI, property: KProperty<*>): T { ... }
}

class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(
            thisRef: MyUI,
            prop: KProperty<*>
    ): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        // create delegate
        return ResourceDelegate()
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ... }
}

class MyUI {
    fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ... }

    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

The parameters of `provideDelegate` are the same as those of `getValue`:

- `thisRef` must be the same type as, or a supertype of, the *property owner* (for extension properties, it should be the type being extended);
- `property` must be of type `KProperty<*>` or its supertype.

The `provideDelegate` method is called for each property during the creation of the `MyUI` instance, and it performs the necessary validation right away.

Without this ability to intercept the binding between the property and its delegate, to achieve the same functionality you'd have to pass the property name explicitly, which isn't very convenient:

```
// Checking the property name without "provideDelegate" functionality
class MyUI {
    val image by bindResource(ResourceID.image_id, "image")
    val text by bindResource(ResourceID.text_id, "text")
}

fun <T> MyUI.bindResource(
        id: ResourceID<T>,
        propertyName: String
): ReadOnlyProperty<MyUI, T> {
    checkProperty(this, propertyName)
    // create delegate
}
```

In the generated code, the `provideDelegate` method is called to initialize the auxiliary `prop$delegate` property. Compare the generated code for the property declaration `val prop: Type by MyDelegate()` with the generated code above (when the `provideDelegate` method is not present):

```
class C {
    var prop: Type by MyDelegate()
}

// this code is generated by the compiler
// when the 'provideDelegate' function is available:
class C {
    // calling "provideDelegate" to create the additional "delegate" property
    private val prop$delegate = MyDelegate().provideDelegate(this, this::prop)
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

Note that the `provideDelegate` method affects only the creation of the auxiliary property and doesn't affect the code generated for the getter or the setter.

With the `PropertyDelegateProvider` interface from the standard library, you can create delegate providers without creating new classes.

```
val provider = PropertyDelegateProvider { thisRef: Any?, property ->
    ReadOnlyProperty<Any?, Int> {_, property -> 42 }
}
val delegate: Int by provider
```

Null safety

Null safety is a Kotlin feature designed to significantly reduce the risk of null references, also known as The Billion-Dollar Mistake.

One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference results in a null reference exception. In Java, this would be the equivalent of a `NullPointerException`, or an *NPE* for short.

Kotlin explicitly supports nullability as part of its type system, meaning you can explicitly declare which variables or properties are allowed to be `null`. Also, when you declare non-null variables, the compiler enforces that these variables cannot hold a `null` value, preventing an NPE.

Kotlin's null safety ensures safer code by catching potential null-related issues at compile time rather than runtime. This feature improves code robustness, readability, and maintainability by explicitly expressing `null` values, making the code easier to understand and manage.

The only possible causes of an NPE in Kotlin are:

- An explicit call to `throw NullPointerException()`.
- Usage of the not-null assertion operator `!!`.
- Data inconsistency during initialization, such as when:
  - An uninitialized `this` available in a constructor is used somewhere else (a "leaking `this`").
  - A superclass constructor calling an open member whose implementation in the derived class uses an uninitialized state.
- Java interoperation:
  - Attempts to access a member of a `null` reference of a platform type.

- Nullability issues with generic types. For example, a piece of Java code adding `null` into a Kotlin `MutableList<String>`, which would require `MutableList<String?>` to handle it properly.
- Other issues caused by external Java code.

> Besides NPE, another exception related to null safety is `UninitializedPropertyAccessException`. Kotlin throws this exception when you try to access a property that has not been initialized, ensuring that non-nullable properties are not used until they are ready. This typically happens with `lateinit` properties.

Nullable types and non-nullable types

In Kotlin, the type system distinguishes between types that can hold `null` (nullable types) and those that cannot (non-nullable types). For example, a regular variable of type `String` cannot hold `null`:

```
fun main() {
    // Assigns a non-null string to a variable
    var a: String = "abc"
    // Attempts to re-assign null to the non-nullable variable
    a = null
    print(a)
    // Null can not be a value of a non-null type String
}
```

You can safely call a method or access a property on `a`. It's guaranteed not to cause an NPE because `a` is a non-nullable variable. The compiler ensures that `a` always holds a valid `String` value, so there's no risk of accessing its properties or methods when it's `null`:

```
fun main() {
    // Assigns a non-null string to a variable
    val a: String = "abc"
    // Returns the length of a non-nullable variable
    val l = a.length
    print(l)
    // 3
}
```

To allow `null` values, declare a variable with a `?` sign right after the variable type. For example, you can declare a nullable string by writing `String?`. This expression makes `String` a type that can accept `null`:

```
fun main() {
    // Assigns a nullable string to a variable
    var b: String? = "abc"
    // Successfully re-assigns null to the nullable variable
    b = null
    print(b)
    // null
}
```

If you try accessing `length` directly on `b`, the compiler reports an error. This is because `b` is declared as a nullable variable and can hold `null` values. Attempting to access properties on nullables directly leads to an NPE:

```
fun main() {
    // Assigns a nullable string to a variable
    var b: String? = "abc"
    // Re-assigns null to the nullable variable
    b = null
    // Tries to directly return the length of a nullable variable
    val l = b.length
    print(l)
    // Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type
String?
}
```

In the example above, the compiler requires you to use safe calls to check for nullability before accessing properties or performing operations. There are several ways to handle nullables:

- Check for `null` with the `if` conditional
- Safe call operator `?.`
- Elvis operator `?:`
- Not-null assertion operator `!!`
- Nullable receiver
- `let` function
- Safe casts `as?`
- Collections of a nullable type

Read the next sections for details and examples of `null` handling tools and techniques.

Check for null with the if conditional

When working with nullable types, you need to handle nullability safely to avoid an NPE. One way to handle this is checking for nullability explicitly with the `if` conditional expression.

For example, check whether `b` is `null` and then access `b.length`:

```
fun main() {
    // Assigns null to a nullable variable
    val b: String? = null
    // Checks for nullability first and then accesses length
    val l = if (b ≠ null) b.length else -1
    print(l)
    // -1
}
```

In the example above, the compiler performs a smart cast to change the type from nullable `String?` to non-nullable `String`. It also tracks the information about the check you performed and allows the call to `length` inside the `if` conditional.

More complex conditions are supported as well:

```
fun main() {
    // Assigns a nullable string to a variable
    val b: String? = "Kotlin"

    // Checks for nullability first and then accesses length
    if (b ≠ null && b.length > 0) {
        print("String of length ${b.length}")
        // String of length 6
    } else {
        // Provides alternative if the condition is not met
        print("Empty string")
    }
}
```

Note that the example above only works when the compiler can guarantee that `b` doesn't change between the check and its usage, same as the smart cast prerequisites.

Safe call operator

The safe call operator `?.` allows you to handle nullability safely in a shorter form. Instead of throwing an NPE, if the object is `null`, the `?.` operator simply returns `null`:

```
fun main() {
    // Assigns a nullable string to a variable
    val a: String? = "Kotlin"
    // Assigns null to a nullable variable
    val b: String? = null

    // Checks for nullability and returns length or null
    println(a?.length)
    // 6
    println(b?.length)
    // null
}
```

The `b?.length` expression checks for nullability and returns `b.length` if `b` is non-null, or `null` otherwise. The type of this expression is `Int?`.

You can use the `?.` operator with both `var` and `val` variables in Kotlin:

- A nullable `var` can hold a `null` (for example, `var nullableValue: String? = null`) or a non-null value (for example, `var nullableValue: String? = "Kotlin"`). If it's a non-null value, you can change it to `null` at any point.
- A nullable `val` can hold a `null` (for example, `val nullableValue: String? = null`) or a non-null value (for example, `val nullableValue: String? = "Kotlin"`). If it's a non-null value, you cannot change it to `null` subsequently.

Safe calls are useful in chains. For example, Bob is an employee who may be assigned to a department (or not). That department may, in turn, have another employee as a department head. To obtain the name of Bob's department head (if there is one), you write the following:

```
bob?.department?.head?.name
```

This chain returns `null` if any of its properties are `null`.

You can also place a safe call on the left side of an assignment:

```
person?.department?.head = managersPool.getManager()
```

In the example above, if one of the receivers in the safe call chain is `null`, the assignment is skipped, and the expression on the right is not evaluated at all. For example, if either `person` or `person.department` is `null`, the function is not called. Here's the equivalent of the same safe call but with the `if` conditional:

```
if (person ≠ null && person.department ≠ null) {
    person.department.head = managersPool.getManager()
}
```

Elvis operator

When working with nullable types, you can check for `null` and provide an alternative value. For example, if `b` is not `null`, access `b.length`. Otherwise, return an alternative value:

```
fun main() {
    // Assigns null to a nullable variable
    val b: String? = null
    // Checks for nullability. If not null, returns length. If null, returns 0
    val l: Int = if (b ≠ null) b.length else 0
    println(l)
    // 0
}
```

Instead of writing the complete `if` expression, you can handle this in a more concise way with the Elvis operator `?:`:

```
fun main() {
    // Assigns null to a nullable variable
    val b: String? = null
    // Checks for nullability. If not null, returns length. If null, returns a non-null value
    val l = b?.length ?: 0
    println(l)
    // 0
}
```

If the expression to the left of `?:` is not `null`, the Elvis operator returns it. Otherwise, the Elvis operator returns the expression to the right. The expression on the right-hand side is evaluated only if the left-hand side is `null`.

Since `throw` and `return` are expressions in Kotlin, you can also use them on the right-hand side of the Elvis operator. This can be handy, for example, when checking function arguments:

```
fun foo(node: Node): String? {
    // Checks for getParent(). If not null, it's assigned to parent. If null, returns null
    val parent = node.getParent() ?: return null
    // Checks for getName(). If not null, it's assigned to name. If null, throws exception
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // ...
}
```

Not-null assertion operator

The not-null assertion operator `!!` converts any value to a non-nullable type.

When you apply the `!!` operator to a variable whose value is not `null`, it's safely handled as a non-nullable type, and the code executes normally. However, if the value is `null`, the `!!` operator forces it to be treated as non-nullable, which results in an NPE.

When `b` is not `null` and the `!!` operator makes it return its non-null value (which is a `String` in this example), it accesses `length` correctly:

```
fun main() {
    // Assigns a nullable string to a variable
    val b: String? = "Kotlin"
    // Treats b as non-null and accesses its length
    val l = b!!.length
    println(l)
    // 6
}
```

When `b` is `null` and the `!!` operator makes it return its non-null value, and an NPE occurs:

```
fun main() {
    // Assigns null to a nullable variable
    val b: String? = null
    // Treats b as non-null and tries to access its length
    val l = b!!.length
    println(l)
    // Exception in thread "main" java.lang.NullPointerException
}
```

The `!!` operator is particularly useful when you are confident that a value is not `null` and there's no chance of getting an NPE, but the compiler cannot guarantee this due to certain rules. In such cases, you can use the `!!` operator to explicitly tell the compiler that the value is not `null`.

Nullable receiver

You can use extension functions with a nullable receiver type, allowing these functions to be called on variables that might be `null`.

By defining an extension function on a nullable receiver type, you can handle `null` values within the function itself instead of checking for `null` at every place where you call the function.

For example, the `.toString()` extension function can be called on a nullable receiver. When invoked on a `null` value, it safely returns the string `"null"` without throwing an exception:

```
fun main() {
    // Assigns null to a nullable Person object stored in the person variable
    val person: Person? = null

    // Applies .toString to the nullable person variable and prints a string
    println(person.toString())
    // null
}

// Defines a simple Person class
data class Person(val name: String)
```

In the example above, even though `person` is `null`, the `.toString()` function safely returns the string `"null"`. This can be helpful for debugging and logging.

If you expect the `.toString()` function to return a nullable string (either a string representation or `null`), use the safe-call operator `?.`. The `?.` operator calls `.toString()` only if the object is not `null`, otherwise it returns `null`:

```
fun main() {
    // Assigns a nullable Person object to a variable
    val person1: Person? = null
    val person2: Person? = Person("Alice")

    // Prints "null" if person is null; otherwise prints the result of person.toString()
    println(person1?.toString())
    // null
    println(person2?.toString())
    // Person(name=Alice)
}

// Defines a Person class
data class Person(val name: String)
```

The `?.` operator allows you to safely handle potential `null` values while still accessing properties or functions of objects that might be `null`.

Let function

To handle `null` values and perform operations only on non-null types, you can use the safe call operator `?.` together with the `let` function.

This combination is useful for evaluating an expression, check the result for `null`, and execute code only if it's not `null`, avoiding manual null checks:

```
fun main() {
    // Declares a list of nullable strings
    val listWithNulls: List<String?> = listOf("Kotlin", null)

    // Iterates over each item in the list
    for (item in listWithNulls) {
        // Checks if the item is null and only prints non-null values
        item?.let { println(it) }
        //Kotlin
    }
}
```

Safe casts

The regular Kotlin operator for type casts is the `as` operator. However, regular casts can result in an exception if the object is not of the target type.

You can use the `as?` operator for safe casts. It tries to cast a value to the specified type and returns `null` if the value is not of that type:

```
fun main() {
    // Declares a variable of type Any, which can hold any type of value
    val a: Any = "Hello, Kotlin!"

    // Safe casts to Int using the 'as?' operator
    val aInt: Int? = a as? Int
    // Safe casts to String using the 'as?' operator
    val aString: String? = a as? String

    println(aInt)
    // null
    println(aString)
    // "Hello, Kotlin!"
}
```

The code above prints `null` because `a` is not an `Int`, so the cast fails safely. It also prints `"Hello, Kotlin!"` because it matches the `String?` type, so the safe cast succeeds.

Collections of a nullable type

If you have a collection of nullable elements and want to keep only the non-null ones, use the `filterNotNull()` function:

```
fun main() {
    // Declares a list containing some null and non-null integer values
    val nullableList: List<Int?> = listOf(1, 2, null, 4)

    // Filters out null values, resulting in a list of non-null integers
    val intList: List<Int> = nullableList.filterNotNull()

    println(intList)
    // [1, 2, 4]
}
```

What's next?

- Learn how to handle nullability in Java and Kotlin.
- Learn about generic types that are definitely non-nullable.

Equality

In Kotlin, there are two types of equality:

- *Structural* equality (`==`) - a check for the `equals()` function
- *Referential* equality (`===`) - a check for two references pointing to the same object

Structural equality

Structural equality verifies if two objects have the same content or structure. Structural equality is checked by the `==` operation and its negated counterpart `!=`. By convention, an expression like `a == b` is translated to:

```
a?.equals(b) ?: (b === null)
```

If `a` is not `null`, it calls the `equals(Any?)` function. Otherwise (`a` is `null`), it checks that `b` is referentially equal to `null`:

```
fun main() {
    var a = "hello"
    var b = "hello"
    var c = null
    var d = null
    var e = d

    println(a == b)
    // true
    println(a == c)
    // false
    println(c == e)
    // true
}
```

Note that there's no point in optimizing your code when comparing to `null` explicitly: `a == null` will be automatically translated to `a === null`.

In Kotlin, the `equals()` function is inherited by all classes from the `Any` class. By default, the `equals()` function implements referential equality. However, classes in Kotlin can override the `equals()` function to provide a custom equality logic and, in this way, implement structural equality.

Value classes and data classes are two specific Kotlin types that automatically override the `equals()` function. That's why they implement structural equality by default.

However, in the case of data classes, if the `equals()` function is marked as `final` in the parent class, its behavior remains unchanged.

Distinctly, non-data classes (those not declared with the `data` modifier) do not override the `equals()` function by default. Instead, non-data classes implement referential equality behavior inherited from the `Any` class. To implement structural equality, non-data classes require a custom equality logic to override the `equals()` function.

To provide a custom equals check implementation, override the `equals(other: Any?): Boolean` function:

```
class Point(val x: Int, val y: Int) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is Point) return false

        // Compares properties for structural equality
        return this.x == other.x && this.y == other.y
    }
}
```



When overriding the equals() function, you should also override the hashCode() function to keep consistency between equality and hashing and ensure the proper behavior of these functions.

Functions with the same name and other signatures (like `equals(other: Foo)`) don't affect equality checks with the operators `==` and `!=`.

Structural equality has nothing to do with comparison defined by the `Comparable<...>` interface, so only a custom `equals(Any?)` implementation may affect the behavior of the operator.

Referential equality

Referential equality verifies the memory addresses of two objects to determine if they are the same instance.

Referential equality is checked by the `===` operation and its negated counterpart `!==`. `a === b` evaluates to true if and only if `a` and `b` point to the same object:

```
fun main() {
    var a = "Hello"
    var b = a
    var c = "world"
    var d = "world"

    println(a === b)
    // true
    println(a === c)
    // false
    println(c === d)
    // true

}
```

For values represented by primitive types at runtime (for example, `Int`), the `===` equality check is equivalent to the `==` check.

> The referential equality is implemented differently in Kotlin/JS. For more information about equality, see the Kotlin/JS documentation.

Floating-point numbers equality

When the operands of an equality check are statically known to be `Float` or `Double` (nullable or not), the check follows the IEEE 754 Standard for Floating-Point Arithmetic.

The behavior is different for operands that are not statically typed as floating-point numbers. In these cases, structural equality is implemented. As a result, checks with operands not statically typed as floating-point numbers differ from the IEEE standard. In this scenario:

- `NaN` is equal to itself
- `NaN` is greater than any other element (including `POSITIVE_INFINITY`)
- `-0.0` is not equal to `0.0`

For more information, see Floating-point numbers comparison.

Array equality

To compare whether two arrays have the same elements in the same order, use `contentEquals()`.

For more information, see Compare arrays.

Generics: in, out, where

Classes in Kotlin can have type parameters, just like in Java:

```
class Box<T>(t: T) {
    var value = t
}
```

To create an instance of such a class, simply provide the type arguments:

```
val box: Box<Int> = Box<Int>(1)
```

If the compiler can infer the type arguments, for example, from the constructor arguments, you don't need to specify them explicitly:

```
val box = Box(1) // 1 has type Int, so the compiler figures out that it is Box<Int>
```

Variance

One of the trickiest aspects of Java's type system is the wildcard types (see Java Generics FAQ). Kotlin doesn't have these. Instead, Kotlin has declaration-site variance and type projections.

Variance and wildcards in Java

Let's think about why Java needs these mysterious wildcards. First, generic types in Java are *invariant*, meaning that `List<String>` is *not* a subtype of `List<Object>`. If `List` were not *invariant*, it would have been no better than Java's arrays, as the following code would have compiled but caused an exception at runtime:

```
// Java
List<String> strs = new ArrayList<String>();

// Java reports a type mismatch here at compile-time.
List<Object> objs = strs;

// What if it didn't?
// We would be able to put an Integer into a list of Strings.
objs.add(1);

// And then at runtime, Java would throw
// a ClassCastException: Integer cannot be cast to String
String s = strs.get(0);
```

Java prohibits such things to guarantee runtime safety. But this has implications. For example, consider the `addAll()` method from the `Collection` interface. What's the signature of this method? Intuitively, you'd write it this way:

```
// Java
interface Collection<E> ... {
    void addAll(Collection<E> items);
}
```

But then, you would not be able to do the following (which is perfectly safe):

```java
// Java

// The following would not compile with the naive declaration of addAll:
// Collection<String> is not a subtype of Collection<Object>
void copyAll(Collection<Object> to, Collection<String> from) {
    to.addAll(from);
}
```

That's why the actual signature of `addAll()` is the following:

```java
// Java
interface Collection<E> ... {
    void addAll(Collection<? extends E> items);
}
```

The *wildcard type argument* `? extends E` indicates that this method accepts a collection of objects of `E` *or a subtype of* `E`, not just `E` itself. This means that you can safely *read* `E`'s from items (elements of this collection are instances of a subclass of E), but *cannot write* to it as you don't know what objects comply with that unknown subtype of `E`. In return for this limitation, you get the desired behavior: `Collection<String>` *is* a subtype of `Collection<? extends Object>`. In other words, the wildcard with an *extends*-bound (*upper* bound) makes the type *covariant*.

The key to understanding why this works is rather simple: if you can only *take* items from a collection, then using a collection of `String`s and reading `Object`s from it is fine. Conversely, if you can only *put* items into the collection, it's okay to take a collection of `Object`s and put `String`s into it: in Java there is `List<? super String>`, which accepts `String`s or any of its supertypes.

The latter is called *contravariance*, and you can only call methods that take `String` as an argument on `List<? super String>` (for example, you can call `add(String)` or `set(int, String)`). If you call something that returns `T` in `List<T>`, you don't get a `String`, but rather an `Object`.

Joshua Bloch, in his book Effective Java, 3rd Edition, explains the problem well (Item 31: "Use bounded wildcards to increase API flexibility"). He gives the name *Producers* to objects you only *read from* and *Consumers* to those you only *write to*. He recommends:

> "For maximum flexibility, use wildcard types on input parameters that represent producers or consumers."

He then proposes the following mnemonic: *PECS* stands for *Producer-Extends, Consumer-Super.*

If you use a producer-object, say, `List<? extends Foo>`, you are not allowed to call `add()` or `set()` on this object, but this does not mean that it is *immutable*: for example, nothing prevents you from calling `clear()` to remove all the items from the list, since `clear()` does not take any parameters at all.

The only thing guaranteed by wildcards (or other types of variance) is *type safety*. Immutability is a completely different story.

Declaration-site variance

Let's suppose that there is a generic interface `Source<T>` that does not have any methods that take `T` as a parameter, only methods that return `T`:

```java
// Java
interface Source<T> {
    T nextT();
}
```

Then, it would be perfectly safe to store a reference to an instance of `Source<String>` in a variable of type `Source<Object>` - there are no consumer-methods to call. But Java does not know this, and still prohibits it:

```java
// Java
void demo(Source<String> strs) {
    Source<Object> objects = strs; // !!! Not allowed in Java
    // ...
}
```

To fix this, you should declare objects of type `Source<? extends Object>`. Doing so is meaningless, because you can call all the same methods on such a variable as before, so there's no value added by the more complex type. But the compiler does not know that.

In Kotlin, there is a way to explain this sort of thing to the compiler. This is called *declaration-site variance*: you can annotate the *type parameter* `T` of `Source` to make sure that it is only *returned* (produced) from members of `Source<T>`, and never consumed. To do this, use the `out` modifier:

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // This is OK, since T is an out-parameter
    // ...
}
```

The general rule is this: when a type parameter `T` of a class `C` is declared `out`, it may occur only in the *out*-position in the members of `C`, but in return `C<Base>` can safely be a supertype of `C<Derived>`.

In other words, you can say that the class `C` is *covariant* in the parameter `T`, or that `T` is a *covariant* type parameter. You can think of `C` as being a *producer* of `T`'s, and NOT a *consumer* of `T`'s.

The `out` modifier is called a *variance annotation*, and since it is provided at the type parameter declaration site, it provides *declaration-site variance*. This is in contrast with Java's *use-site variance* where wildcards in the type usages make the types covariant.

In addition to `out`, Kotlin provides a complementary variance annotation: `in`. It makes a type parameter *contravariant*, meaning it can only be consumed and never produced. A good example of a contravariant type is `Comparable`:

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 has type Double, which is a subtype of Number
    // Thus, you can assign x to a variable of type Comparable<Double>
    val y: Comparable<Double> = x // OK!
}
```

The words *in* and *out* seem to be self-explanatory (as they've already been used successfully in C# for quite some time), and so the mnemonic mentioned above is not really needed. It can in fact be rephrased at a higher level of abstraction:

The Existential **Transformation: Consumer in, Producer out!**:-)

Type projections

Use-site variance: type projections

It is very easy to declare a type parameter `T` as `out` and avoid trouble with subtyping on the use site, but some classes *can't* actually be restricted to only return `T`'s! A good example of this is `Array`:

```
class Array<T>(val size: Int) {
    operator fun get(index: Int): T { ... }
    operator fun set(index: Int, value: T) { ... }
}
```

This class can be neither co- nor contravariant in `T`. And this imposes certain inflexibilities. Consider the following function:

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

This function is supposed to copy items from one array to another. Let's try to apply it in practice:

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3) { "" }
copy(ints, any)
//   ^ type is Array<Int> but Array<Any> was expected
```

Here you run into the same familiar problem: `Array<T>` is *invariant* in `T`, and so neither `Array<Int>` nor `Array<Any>` is a subtype of the other. Why not? Again, this is because `copy` could have an unexpected behavior, for example, it may attempt to write a `String` to `from`, and if you actually pass an array of `Int` there, a `ClassCastException` will be thrown later.

To prohibit the `copy` function from *writing* to `from`, you can do the following:

```
fun copy(from: Array<out Any>, to: Array<Any>) { ... }
```

This is *type projection*, which means that `from` is not a simple array, but is rather a restricted (*projected*) one. You can only call methods that return the type parameter `T`, which in this case means that you can only call `get()`. This is our approach to *use-site variance*, and it corresponds to Java's `Array<? extends Object>` while being slightly simpler.

You can project a type with `in` as well:

```
fun fill(dest: Array<in String>, value: String) { ... }
```

`Array<in String>` corresponds to Java's `Array<? super String>`. This means that you can pass an array of `String`, `CharSequence`, or `Object` to the `fill()` function.

Star-projections

Sometimes you want to say that you know nothing about the type argument, but you still want to use it in a safe way. The safe way here is to define such a projection of the generic type, that every concrete instantiation of that generic type will be a subtype of that projection.

Kotlin provides so-called *star-projection* syntax for this:

- For `Foo<out T : TUpper>`, where `T` is a covariant type parameter with the upper bound `TUpper`, `Foo<*>` is equivalent to `Foo<out TUpper>`. This means that when the `T` is unknown, you can safely *read* values of `TUpper` from `Foo<*>`.
- For `Foo<in T>`, where `T` is a contravariant type parameter, `Foo<*>` is equivalent to `Foo<in Nothing>`. This means there is nothing you can *write* to `Foo<*>` in a safe way when `T` is unknown.
- For `Foo<T : TUpper>`, where `T` is an invariant type parameter with the upper bound `TUpper`, `Foo<*>` is equivalent to `Foo<out TUpper>` for reading values and to `Foo<in Nothing>` for writing values.

If a generic type has several type parameters, each of them can be projected independently. For example, if the type is declared as `interface Function<in T, out U>` you could use the following star-projections:

- `Function<*, String>` means `Function<in Nothing, String>`.
- `Function<Int, *>` means `Function<Int, out Any?>`.
- `Function<*, *>` means `Function<in Nothing, out Any?>`.

Star-projections are very much like Java's raw types, but safe.

Captured types

When you use a type projection, such as `out T` or `in T`, the compiler internally represents the unknown concrete type as a *captured type*. A captured type is an unknown type with known upper and lower bounds.

Captured types are non-denotable, so you can't write them directly in Kotlin code. Instead, you can most frequently see captured types in compiler diagnostics, such as `CapturedType(out X)`. For example, the following type mismatch diagnostic contains a captured type:

```
val array: Array<out CharSequence> = arrayOf("str")
val item: Int = array.get(0)
// Initializer type mismatch: expected 'Int', actual 'CapturedType(out CharSequence)'
```

The compiler uses the captured type's upper bound for read operations and its lower bound to determine which values are type-safe for write operations:

```
// The projected type is Array<out CharSequence>
val array: Array<out CharSequence> = arrayOf("Kotlin")

// The get() read operation uses the captured type's upper bound, CharSequence
val item = array.get(0)

// The set() write operation uses the captured type's lower bound, Nothing,
// which results in an error
array.set(0, "New value")
// Receiver type 'Array<out CharSequence>' contains out projection
// which prohibits the use of 'fun set(index: Int, value: T): Unit'
```

In this example:

- The variable `array` has the projected type `Array<out CharSequence>`. The compiler represents the projected type argument `out CharSequence` as a captured type with `CharSequence` as its upper bound and `Nothing` as its lower bound.
- For the `get()` operation, the compiler approximates the captured type to its upper bound, `CharSequence`, and infers `CharSequence` as the type of `item`.
- For the `set()` operation, the captured type has `Nothing` as its lower bound. Since `Nothing` has no instances, writing a value to the projected type isn't type-safe and results in an error.

Generic functions

Classes aren't the only declarations that can have type parameters. Functions can, too. Type parameters are placed *before* the name of the function:

```
fun <T> singletonList(item: T): List<T> {
    // ...
}

fun <T> T.basicToString(): String { // extension function
    // ...
}
```

To call a generic function, specify the type arguments at the call site *after* the name of the function:

```
val l = singletonList<Int>(1)
```

Type arguments can be omitted if they can be inferred from the context, so the following example works as well:

```
val l = singletonList(1)
```

Generic constraints

The set of all possible types that can be substituted for a given type parameter may be restricted by *generic constraints*.

Upper bounds

The most common type of constraint is an *upper bound*, which corresponds to Java's `extends` keyword:

```
fun <T : Comparable<T>> sort(list: List<T>) {  ... }
```

The type specified after a colon is the *upper bound*, indicating that only a subtype of `Comparable<T>` can be substituted for `T`. For example:

```
sort(listOf(1, 2, 3)) // OK. Int is a subtype of Comparable<Int>
sort(listOf(HashMap<Int, String>())) // Error: HashMap<Int, String> is not a subtype of
Comparable<HashMap<Int, String>>
```

The default upper bound (if there was none specified) is `Any?`. Only one upper bound can be specified inside the angle brackets. If the same type parameter needs more than one upper bound, you need a separate *where*-clause:

```
fun <T> copyWhenGreater(list: List<T>, threshold: T): List<String>
    where T : CharSequence,
          T : Comparable<T> {
    return list.filter { it > threshold }.map { it.toString() }
}
```

The passed type must satisfy all conditions of the `where` clause simultaneously. In the above example, the `T` type must implement *both* `CharSequence` and `Comparable`.

Definitely non-nullable types

To make interoperability with generic Java classes and interfaces easier, Kotlin supports declaring a generic type parameter as **definitely non-nullable**.

To declare a generic type `T` as definitely non-nullable, declare the type with `& Any`. For example: `T & Any`.

A definitely non-nullable type must have a nullable upper bound.

The most common use case for declaring definitely non-nullable types is when you want to override a Java method that contains `@NotNull` as an argument. For example, consider the `load()` method:

```
import org.jetbrains.annotations.*;

public interface Game<T> {
    public T save(T x) {}
    @NotNull
    public T load(@NotNull T x) {}
}
```

To override the `load()` method in Kotlin successfully, you need `T1` to be declared as definitely non-nullable:

```
interface ArcadeGame<T1> : Game<T1> {
    override fun save(x: T1): T1
    // T1 is definitely non-nullable
    override fun load(x: T1 & Any): T1 & Any
}
```

When working only with Kotlin, it's unlikely that you will need to declare definitely non-nullable types explicitly because Kotlin's type inference takes care of this for you.

Type erasure

The type safety checks that Kotlin performs for generic declaration usages are done at compile time. At runtime, the instances of generic types do not hold any information about their actual type arguments. The type information is said to be *erased*. For example, the instances of `Foo<Bar>` and `Foo<Baz?>` are erased to just `Foo<*>`.

Generics type checks and casts

Due to the type erasure, there is no general way to check whether an instance of a generic type was created with certain type arguments at runtime, and the compiler prohibits such `is`-checks such as `ints is List<Int>` or `list is T` (type parameter). However, you can check an instance against a star-projected type:

```
if (something is List<*>) {
    something.forEach { println(it) } // The items are typed as `Any?`
}
```

Similarly, when you already have the type arguments of an instance checked statically (at compile time), you can make an `is`-check or a cast that involves the non-generic part of the type. Note that angle brackets are omitted in this case:

```
fun handleStrings(list: MutableList<String>) {
    if (list is ArrayList) {
        // `list` is smart-cast to `ArrayList<String>`
    }
}
```

The same syntax but with the type arguments omitted can be used for casts that do not take type arguments into account: `list as ArrayList`.

The type arguments of generic function calls are also only checked at compile time. Inside the function bodies, the type parameters cannot be used for type checks, and type casts to type parameters (`foo as T`) are unchecked. The only exclusion is inline functions with reified type parameters, which have their actual type arguments inlined at each call site. This enables type checks and casts for the type parameters. However, the restrictions described above still apply for instances of generic types used inside checks or casts. For example, in the type check `arg is T`, if `arg` is an instance of a generic type itself, its type arguments are still erased.

```
inline fun <reified A, reified B> Pair<*, *>.asPairOf(): Pair<A, B>? {
    if (first !is A || second !is B) return null
    return first as A to second as B
}

val somePair: Pair<Any?, Any?> = "items" to listOf(1, 2, 3)


val stringToSomething = somePair.asPairOf<String, Any>()
val stringToInt = somePair.asPairOf<String, Int>()
val stringToList = somePair.asPairOf<String, List<*>>()
val stringToStringList = somePair.asPairOf<String, List<String>>() // Compiles but breaks type safety!
// Expand the sample for more details


fun main() {
    println("stringToSomething = " + stringToSomething)
    println("stringToInt = " + stringToInt)
    println("stringToList = " + stringToList)
    println("stringToStringList = " + stringToStringList)
    //println(stringToStringList?.second?.forEach() {it.length}) // This will throw ClassCastException
as list items are not String
}
```

Unchecked casts

Type casts to generic types with concrete type arguments such as `foo as List<String>` cannot be checked at runtime.
These unchecked casts can be used when type safety is implied by the high-level program logic but cannot be inferred directly by the compiler. See the example below.

```
fun readDictionary(file: File): Map<String, *> = file.inputStream().use {
    TODO("Read a mapping of strings to arbitrary elements.")
}

// We saved a map with `Int`s into this file
val intsFile = File("ints.dictionary")

// Warning: Unchecked cast: `Map<String, *>` to `Map<String, Int>`
val intsDictionary: Map<String, Int> = readDictionary(intsFile) as Map<String, Int>
```

A warning appears for the cast in the last line. The compiler can't fully check it at runtime and provides no guarantee that the values in the map are `Int`.

To avoid unchecked casts, you can redesign the program structure. In the example above, you could use the `DictionaryReader<T>` and `DictionaryWriter<T>` interfaces with type-safe implementations for different types. You can introduce reasonable abstractions to move unchecked casts from the call site to the implementation details. Proper use of generic variance can also help.

For generic functions, using reified type parameters makes casts like `arg as T` checked, unless `arg`'s type has *its own* type arguments that are erased.

An unchecked cast warning can be suppressed by annotating the statement or the declaration where it occurs with `@Suppress("UNCHECKED_CAST")`:

```
inline fun <reified T> List<*>.asListOfType(): List<T>? =
    if (all { it is T })
        @Suppress("UNCHECKED_CAST")
        this as List<T> else
        null
```

On the JVM: array types (`Array<Foo>`) retain information about the erased type of their elements, and type casts to an array type are partially checked: the nullability and actual type arguments of the element type are still erased. For example, the cast `foo as Array<List<String>?>` will succeed if `foo` is an array holding any `List<*>`, whether it is nullable or not.

Underscore operator for type arguments

The underscore operator `_` can be used for type arguments. Use it to automatically infer a type of the argument when other types are explicitly specified:

```
abstract class SomeClass<T> {
    abstract fun execute() : T
}

class SomeImplementation : SomeClass<String>() {
    override fun execute(): String = "Test"
}

class OtherImplementation : SomeClass<Int>() {
    override fun execute(): Int = 42
}

object Runner {
    inline fun <reified S: SomeClass<T>, T> run() : T {
        return S::class.java.getDeclaredConstructor().newInstance().execute()
    }
}

fun main() {
    // T is inferred as String because SomeImplementation derives from SomeClass<String>
    val s = Runner.run<SomeImplementation, _>()
    assert(s == "Test")

    // T is inferred as Int because OtherImplementation derives from SomeClass<Int>
    val n = Runner.run<OtherImplementation, _>()
    assert(n == 42)
}
```

Asynchronous programming techniques

For decades, as developers we are confronted with a problem to solve - how to prevent our applications from blocking. Whether we're developing desktop, mobile, or even backend applications, we want to avoid making the user wait or, even worse, create bottlenecks that prevent the application from scaling.

There have been many approaches to solving this problem, including:

- Threading
- Callbacks
- Futures, promises, and others
- Reactive Extensions
- Coroutines

Before explaining what coroutines are, let's briefly review some of the other solutions.

Threading

Threads are by far probably the most well-known approach to avoid applications from blocking.

```
fun postItem(item: Item) {
    val token = preparePost()
    val post = submitPost(token, item)
    processPost(post)
}

fun preparePost(): Token {
    // makes a request and consequently blocks the main thread
    return token
}
```

Let's assume in the code above that `preparePost` is a long-running process and consequently would block the user interface. What we can do is launch it in a separate thread. This would then allow us to avoid the UI from blocking. This is a very common technique, but has a series of drawbacks:

- Threads aren't cheap. Threads require context switches which are costly.
- Threads aren't infinite. The number of threads that can be launched is limited by the underlying operating system. In backend applications, this could cause a major bottleneck.
- Threads aren't always available. Some platforms, such as JavaScript, do not even support threads.
- Threads aren't easy. Debugging threads and avoiding race conditions are common problems we suffer in multithreaded programming.

Callbacks

With callbacks, the idea is to pass one function as a parameter to another function, and have this one invoked once the process has completed.

```
fun postItem(item: Item) {
    preparePostAsync { token →
        submitPostAsync(token, item) { post →
            processPost(post)
        }
    }
}

fun preparePostAsync(callback: (Token) → Unit) {
    // make request and return immediately
    // arrange callback to be invoked later
}
```

This in principle feels like a much more elegant solution, but once again has several issues:

- Difficulty of nested callbacks. Usually a function that is used as a callback, often ends up needing its own callback. This leads to a series of nested callbacks which lead to incomprehensible code. The pattern is often referred to as callback hell, or the pyramid of doom due to the triangular shape that indentations from these deeply nested callbacks create.
- Error handling is complicated. The nesting model makes error handling and propagation of these somewhat more complicated.

Callbacks are quite common in event-loop architectures such as JavaScript, but even there, generally people have moved away to using other approaches such as promises or reactive extensions.

Futures, promises, and others

The idea behind futures or promises (other terms may be used depending on the language or platform), is that when we make a call, we're *promised* that at some point the call will return a `Promise` object, which we can then operate on.

```
fun postItem(item: Item) {
    preparePostAsync()
        .thenCompose { token →
            submitPostAsync(token, item)
        }
        .thenAccept { post →
            processPost(post)
        }

}

fun preparePostAsync(): Promise<Token> {
    // makes request and returns a promise that is completed later
    return promise
}
```

This approach requires a series of changes in how we program, in particular:

- Different programming model. Similar to callbacks, the programming model moves away from a top-down imperative approach to a compositional model with chained calls. Traditional program structures such as loops, exception handling, etc. usually are no longer valid in this model.
- Different APIs. Usually there's a need to learn a completely new API such as `thenCompose` or `thenAccept`, which can also vary across platforms.
- Specific return type. The return type moves away from the actual data that we need and instead returns a new type `Promise` which has to be introspected.
- Error handling can be complicated. The propagation and chaining of errors aren't always straightforward.

Reactive extensions

Reactive Extensions (Rx) were introduced to C# by Erik Meijer. While it was definitely used on the .NET platform, it really didn't reach mainstream adoption until Netflix ported it over to Java, naming it RxJava. From then on, numerous ports have been provided for a variety of platforms including JavaScript (RxJS).

The idea behind Rx is to move towards what's called `observable streams` whereby we now think of data as streams (infinite amounts of data) and these streams can be observed. In practical terms, Rx is simply the Observer Pattern with a series of extensions which allow us to operate on the data.

In approach it's quite similar to Futures, but one can think of a Future as returning a discrete element, whereas Rx returns a stream. However, similar to the previous, it also introduces a complete new way of thinking about our programming model, famously phrased as

```
"everything is a stream, and it's observable"
```

This implies a different way to approach problems and quite a significant shift from what we're used to when writing synchronous code. One benefit as opposed to Futures is that given it's ported to so many platforms, generally we can find a consistent API experience no matter what we use, be it C#, Java, JavaScript, or any other language where Rx is available.

In addition, Rx does introduce a somewhat nicer approach to error handling.

Coroutines

Kotlin's approach to working with asynchronous code is using coroutines, which is the idea of suspendable computations, i.e. the idea that a function can suspend its execution at some point and resume later on.

However, one of the benefits of coroutines is that when it comes to the developer, writing non-blocking code is essentially the same as writing blocking code. The programming model itself doesn't really change.

Take for instance the following code:

```
fun postItem(item: Item) {
    launch {
        val token = preparePost()
        val post = submitPost(token, item)
        processPost(post)
    }
}

suspend fun preparePost(): Token {
    // makes a request and suspends the coroutine
    return suspendCoroutine { /* ... */ }
}
```

This code will launch a long-running operation without blocking the main thread. The `preparePost` is what's called a `suspendable function`, thus the keyword `suspend` prefixing it. What this means as stated above, is that the function will execute, pause execution and resume at some point in time.

- The function signature remains exactly the same. The only difference is `suspend` being added to it. The return type, however, is the type we want to be returned.
- The code is still written as if we were writing synchronous code, top-down, without the need of any special syntax, beyond the use of a function called `launch` which essentially kicks off the coroutine (covered in other tutorials).
- The programming model and APIs remain the same. We can continue to use loops, exception handling, etc. and there's no need to learn a complete set of new APIs.
- It is platform independent. Whether we're targeting JVM, JavaScript or any other platform, the code we write is the same. Under the covers, the compiler takes care of adapting it to each platform.

Coroutines are not a new concept, let alone invented by Kotlin. They've been around for decades and are popular in some other programming languages such as Go. What is important to note though is that the way they're implemented in Kotlin, most of the functionality is delegated to libraries. In fact, beyond the `suspend` keyword, no other keywords are added to the language. This is somewhat different from languages such as C# that have `async` and `await` as part of the syntax. With Kotlin, these are just library functions.

For more information, see the Coroutines reference.

Coroutines

Applications often need to perform multiple tasks at the same time, such as responding to user input, loading data, or updating the screen. To support this, they rely on concurrency, which allows operations to run independently without blocking each other.

The most common way to run tasks concurrently is by using threads, which are independent paths of execution managed by the operating system. However, threads are relatively heavy, and creating many of them can lead to performance issues.

To support efficient concurrency, Kotlin uses asynchronous programming built around *coroutines*, which let you write asynchronous code in a natural, sequential style using suspending functions. Coroutines are lightweight alternatives to threads. They can suspend without blocking system resources and are resource-friendly, making them better suited for fine-grained concurrency.

Most coroutine features are provided by the `kotlinx.coroutines` library, which includes tools for launching coroutines, handling concurrency, working with asynchronous streams, and more.

If you're new to coroutines in Kotlin, start with the Coroutine basics guide before diving into more complex topics. This guide introduces the key concepts of suspending functions, coroutine builders, and structured concurrency through simple examples:

Get started with coroutines

Check out the KotlinConf app for a sample project to see how coroutines are used in practice.

Coroutine concepts

The `kotlinx.coroutines` library provides the core building blocks for running tasks concurrently, structuring coroutine execution, and managing shared state.

Suspending functions and coroutine builders

Coroutines in Kotlin are built on suspending functions, which allow code to pause and resume without blocking a thread. The `suspend` keyword marks functions that can perform long-running operations asynchronously.

To launch new coroutines, use coroutine builders like `.launch()` and `.async()`. These builders are extension functions on `CoroutineScope`, which defines the coroutine's lifecycle and provides the coroutine context.

You can learn more about these builders in Coroutine basics and Composing suspend functions.

Coroutine context and behavior

Launching a coroutine from a `CoroutineScope` creates a context that governs its execution. Builder functions like `.launch()` and `.async()` automatically create a set of elements that define how the coroutine behaves:

- The `Job` interface tracks the coroutine's lifecycle and enables structured concurrency.
- `CoroutineDispatcher` controls where the coroutine runs, such as on a background thread or the main thread in UI applications.
- `CoroutineExceptionHandler` handles uncaught exceptions.

These, along with other possible elements, make up the *coroutine context*, which is inherited by default from the coroutine's parent. This context forms a hierarchy that enables structured concurrency, where related coroutines can be canceled together or handle exceptions as a group.

Asynchronous flow and shared mutable state

Kotlin provides several ways for coroutines to communicate. Use one of the following options based on how you want to share values between coroutines:

- `Flow` produces values only when a coroutine actively collects them.
- `Channel` allows multiple coroutines to send and receive values, with each value delivered to exactly one coroutine.
- `SharedFlow` continuously shares every value with all active collecting coroutines.

When multiple coroutines need to access or update the same data, they *share mutable state*. Without coordination, this can lead to race conditions, where operations interfere with each other in unpredictable ways. To safely manage shared mutable state, use `StateFlow` to wrap the shared data. Then, you can update it from one coroutine and collect its latest value from others.

For more information, see Flows, Channels, and the Coroutines and channels tutorial.

What's next

- Learn the fundamentals of coroutines, suspending functions, and builders in the Coroutine basics guide.
- Explore how to combine suspending functions and build coroutine pipelines in Composing suspending functions.
- Learn how to debug coroutines using built-in tools in IntelliJ IDEA.
- For flow-specific debugging, see the Debug Kotlin Flow using IntelliJ IDEA tutorial.
- Read the Guide to UI programming with coroutines to learn about coroutine-based UI development.
- Review best practices for using coroutines in Android.
- Check out the `kotlinx.coroutines` API reference.

Reflection

Reflection is a set of language and library features that allows you to introspect the structure of your program at runtime. Functions and properties are first-class citizens in Kotlin, and the ability to introspect them (for example, learning the name or the type of a property or function at runtime) is essential when using a functional or reactive style.

Kotlin/JS provides limited support for reflection features. Learn more about reflection in Kotlin/JS.

JVM dependency

On the JVM platform, the Kotlin compiler distribution includes the runtime component required for using the reflection features as a separate artifact, `kotlin-reflect.jar`. This is done to reduce the required size of the runtime library for applications that do not use reflection features.

To use reflection in a Gradle or Maven project, add the dependency on `kotlin-reflect`:

- In Gradle:

Kotlin

```
dependencies {
    implementation(kotlin("reflect"))
}
```

Groovy

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-reflect:2.4.20"
}
```

- In Maven:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-reflect</artifactId>
    </dependency>
</dependencies>
```

If you don't use Gradle or Maven, make sure you have `kotlin-reflect.jar` in the classpath of your project. In other supported cases (IntelliJ IDEA projects that use the command-line compiler), it is added by default. In the command-line compiler, you can use the `-no-reflect` compiler option to exclude `kotlin-reflect.jar` from the classpath.

Class references

The most basic reflection feature is getting the runtime reference to a Kotlin class. To obtain the reference to a statically known Kotlin class, you can use the *class literal* syntax:

```
val c = MyClass::class
```

The reference is a KClass type value.

> On JVM: a Kotlin class reference is not the same as a Java class reference. To obtain a Java class reference, use the `.java` property on a `KClass` instance.

Bound class references

You can get the reference to the class of a specific object with the same `::class` syntax by using the object as a receiver:

```
val widget: Widget = ...
assert(widget is GoodWidget) { "Bad widget: ${widget::class.qualifiedName}" }
```

You will obtain the reference to the exact class of an object, for example, `GoodWidget` or `BadWidget`, regardless of the type of the receiver expression (`Widget`).

Callable references

References to functions, properties, and constructors can also be called or used as instances of function types.

The common supertype for all callable references is `KCallable<out R>`, where `R` is the return value type. It is the property type for properties, and the constructed type for constructors.

Function references

When you have a named function declared as below, you can call it directly (`isOdd(5)`):

```
fun isOdd(x: Int) = x % 2 != 0
```

Alternatively, you can use the function as a function type value, that is, pass it to another function. To do so, use the `::` operator:

```
fun isOdd(x: Int) = x % 2 != 0

fun main() {
    val numbers = listOf(1, 2, 3)
    println(numbers.filter(::isOdd))
}
```

Here `::isOdd` is a value of function type `(Int) -> Boolean`.

Function references belong to one of the `KFunction<out R>` subtypes, depending on the parameter count. For instance, `KFunction3<T1, T2, T3, R>`.

`::` can be used with overloaded functions when the expected type is known from the context. For example:

```
fun main() {
    fun isOdd(x: Int) = x % 2 != 0
    fun isOdd(s: String) = s == "brillig" || s == "slithy" || s == "tove"

    val numbers = listOf(1, 2, 3)
    println(numbers.filter(::isOdd)) // refers to isOdd(x: Int)
}
```

Alternatively, you can provide the necessary context by storing the method reference in a variable with an explicitly specified type:

```
val predicate: (String) -> Boolean = ::isOdd   // refers to isOdd(x: String)
```

If you need to use a member of a class or an extension function, it needs to be qualified: `String::toCharArray`.

Even if you initialize a variable with a reference to an extension function, the inferred function type will have no receiver, but it will have an additional parameter accepting a receiver object. To have a function type with a receiver instead, specify the type explicitly:

```
val isEmptyStringList: List<String>.() -> Boolean = List<String>::isEmpty
```

Example: function composition

Consider the following function:

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

It returns a composition of two functions passed to it: `compose(f, g) = f(g(*))`. You can apply this function to callable references:

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}

fun isOdd(x: Int) = x % 2 != 0

fun main() {
    fun length(s: String) = s.length

    val oddLength = compose(::isOdd, ::length)
    val strings = listOf("a", "ab", "abc")

    println(strings.filter(oddLength))
}
```

Property references

To access properties as first-class objects in Kotlin, use the `::` operator:

```
val x = 1

fun main() {
    println(::x.get())
    println(::x.name)
}
```

The expression `::x` evaluates to a `KProperty0<Int>` type property object. You can read its value using `get()` or retrieve the property name using the `name` property. For more information, see the docs on the `KProperty` class.

For a mutable property such as `var y = 1`, `::y` returns a value with the `KMutableProperty0<Int>` type which has a `set()` method:

```
var y = 1

fun main() {
    ::y.set(2)
    println(y)
}
```

A property reference can be used where a function with a single generic parameter is expected:

```
fun main() {
    val strs = listOf("a", "bc", "def")
    println(strs.map(String::length))
}
```

To access a property that is a member of a class, qualify it as follows:

```
fun main() {
    class A(val p: Int)
    val prop = A::p
    println(prop.get(A(1)))
}
```

For an extension property:

```
val String.lastChar: Char
    get() = this[length - 1]

fun main() {
    println(String::lastChar.get("abc"))
}
```

Interoperability with Java reflection

On the JVM platform, the standard library contains extensions for reflection classes that provide a mapping to and from Java reflection objects (see package `kotlin.reflect.jvm`). For example, to find a backing field or a Java method that serves as a getter for a Kotlin property, you can write something like this:

```
import kotlin.reflect.jvm.*

class A(val p: Int)

fun main() {
    println(A::p.javaGetter) // prints "public final int A.getP()"
    println(A::p.javaField)  // prints "private final int A.p"
}
```

To get the Kotlin class that corresponds to a Java class, use the `.kotlin` extension property:

```
fun getKClass(o: Any): KClass<Any> = o.javaClass.kotlin
```

Constructor references

Constructors can be referenced just like methods and properties. You can use them wherever the program expects a function type object that takes the same parameters as the constructor and returns an object of the appropriate type. Constructors are referenced by using the `::` operator and adding the class name. Consider the following function that expects a function parameter with no parameters and return type `Foo`:

```
class Foo

fun function(factory: () -> Foo) {
    val x: Foo = factory()
}
```

Using `::Foo`, the zero-argument constructor of the class `Foo`, you can call it like this:

```
function(::Foo)
```

Callable references to constructors are typed as one of the `KFunction<out R>` subtypes depending on the parameter count.

Bound function and property references

You can refer to an instance method of a particular object:

```
fun main() {
    val numberRegex = "\\d+".toRegex()
    println(numberRegex.matches("29"))

    val isNumber = numberRegex::matches
    println(isNumber("29"))
}
```

Instead of calling the method `matches` directly, the example uses a reference to it. Such a reference is bound to its receiver. It can be called directly (like in the example above) or used whenever a function type expression is expected:

```
fun main() {
    val numberRegex = "\\d+".toRegex()
    val strings = listOf("abc", "124", "a70")
    println(strings.filter(numberRegex::matches))
}
```

Compare the types of the bound and the unbound references. The bound callable reference has its receiver "attached" to it, so the type of the receiver is no longer a parameter:

```
val isNumber: (CharSequence) -> Boolean = numberRegex::matches

val matches: (Regex, CharSequence) -> Boolean = Regex::matches
```

A property reference can be bound as well:

```
fun main() {
    val prop = "abc"::length
    println(prop.get())
}
```

You don't need to specify `this` as the receiver: `this::foo` and `::foo` are equivalent.

Bound constructor references

A bound callable reference to a constructor of an inner class can be obtained by providing an instance of the outer class:

```
class Outer {
    inner class Inner
}

val o = Outer()
val boundInnerCtor = o::Inner
```

Destructuring declarations

Sometimes it is convenient to *destructure* an object into a number of variables, for example:

```
val (name, age) = person
```

This syntax is called a *destructuring declaration*. A destructuring declaration creates multiple variables at once. You have declared two new variables: `name` and `age`, and can use them independently:

```
println(name)
println(age)
```

A destructuring declaration is compiled down to the following code:

```
val name = person.component1()
val age = person.component2()
```

The `component1()` and `component2()` functions are another example of the *principle of conventions* widely used in Kotlin (see operators like `+` and `*`, `for`-loops as an example). Anything can be on the right-hand side of a destructuring declaration, as long as the required number of component functions can be called on it. And, of course, there can be `component3()` and `component4()` and so on.

> The `componentN()` functions need to be marked with the `operator` keyword to allow using them in a destructuring declaration.

Destructuring declarations also work in `for`-loops:

```
for ((a, b) in collection) { ... }
```

Variables `a` and `b` get the values returned by `component1()` and `component2()` called on elements of the collection.

Example: returning two values from a function

Assume that you need to return two things from a function - for example, a result object and a status of some sort. A compact way of doing this in Kotlin is to declare a data class and return its instance:

```
data class Result(val result: Int, val status: Status)
fun function(...): Result {
    // computations

    return Result(result, status)
}

// Now, to use this function:
val (result, status) = function(...)
```

Since data classes automatically declare `componentN()` functions, destructuring declarations work here.

> You could also use the standard class `Pair` and have `function()` return `Pair<Int, Status>`, but it's often better to have your data named properly.

Example: destructuring declarations and maps

Probably the nicest way to traverse a map is this:

```
for ((key, value) in map) {
    // do something with the key and the value
}
```

To make this work, you should

- Present the map as a sequence of values by providing an `iterator()` function.
- Present each of the elements as a pair by providing functions `component1()` and `component2()`.

And indeed, the standard library provides such extensions:

```
operator fun <K, V> Map<K, V>.iterator(): Iterator<Map.Entry<K, V>> = entrySet().iterator()
operator fun <K, V> Map.Entry<K, V>.component1() = getKey()
operator fun <K, V> Map.Entry<K, V>.component2() = getValue()
```

So you can freely use destructuring declarations in `for`-loops with maps (as well as collections of data class instances or similar).

Underscore for unused variables

If you don't need a variable in the destructuring declaration, you can place an underscore instead of its name:

```
val (_, status) = getResult()
```

The `componentN()` operator functions are not called for the components that are skipped in this way.

Destructuring in lambdas

You can use the destructuring declarations syntax for lambda parameters. If a lambda has a parameter of the `Pair` type (or `Map.Entry`, or any other type that has the appropriate `componentN` functions), you can introduce several new parameters instead of one by putting them in parentheses:

```
map.mapValues { entry -> "${entry.value}!" }
map.mapValues { (key, value) -> "$value!" }
```

Note the difference between declaring two parameters and declaring a destructuring pair instead of a parameter:

```
{ a -> ... } // one parameter
{ a, b -> ... } // two parameters
{ (a, b) -> ... } // a destructured pair
{ (a, b), c -> ... } // a destructured pair and another parameter
```

If a component of the destructured parameter is unused, you can replace it with the underscore to avoid inventing its name:

```
map.mapValues { (_, value) -> "$value!" }
```

You can specify the type for the whole destructured parameter or for a specific component separately:

```
map.mapValues { (_, value): Map.Entry<Int, String> -> "$value!" }

map.mapValues { (_, value: String) -> "$value!" }
```

Experimental

Name-based destructuring

Kotlin supports *name-based destructuring declarations*, where variables match properties by name instead of the position defined by `componentN()` functions in *position-based* destructuring.

> For more information about name-based destructuring, see the feature's KEEP.

In position-based destructuring, variables correspond to the order of `componentN()` functions, for example:

```
data class User(val username: String, val email: String)

fun main() {
    val user = User("alice", "alice@example.com")

    val (email, username) = user

    println(email)
    // alice

    println(username)
    // alice@example.com
}
```

In this example, because destructuring relies on the order of `componentN()` functions, `email` receives the value of `username`, and `username` receives the value of `email`.

With name-based destructuring, property names determine which values are extracted rather than the position of `componentN()` functions:

```
fun main() {
    val user = User("alice", "alice@example.com")

    // Uses name-based destructuring with explicit form
    (val mail = email, val name = username) = user

    println(name)
    // alice

    println(mail)
    // alice@example.com
}
```

Name-based destructuring is Experimental. When you enable this feature, it also introduces a new syntax for position-based destructuring using square brackets. Use this syntax for types where the order of elements matters, such as lists and other ordered collections, as well as unnamed tuples like `Pair` or `Triple`:

```
val point = Pair(10, 20)

// Uses position-based destructuring
val [x, y] = point
```

You can control how the compiler interprets destructuring declarations with the `-Xname-based-destructuring` compiler option.

It has the following modes:

- `only-syntax` enables the explicit form of name-based destructuring without changing the behavior of existing destructuring declarations.
- `name-mismatch` reports warnings when position-based destructuring in data classes uses variable names that don't match the property names.
- `complete` enables short-form name-based destructuring with parentheses and continues supporting position-based destructuring with square bracket syntax.

> Before enabling `complete` mode, review and resolve the warnings reported in `name-mismatch` mode. These warnings show which destructuring declarations the compiler interprets differently in `complete` mode and include suggestions for rewriting those declarations accordingly.

If you use `complete` mode, the short-form destructuring syntax with parentheses matches variables to property names instead of relying on position:

```
val (email, username) = user
```

To enable name-based destructuring in your project, add the compiler option to your build configuration file:

Gradle

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xname-based-destructuring=only-syntax")
    }
}
```

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-Xname-based-destructuring=only-syntax</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Backend development with Kotlin

Kotlin is a great fit for developing server-side applications. With Kotlin, you can write concise and expressive code while maintaining full compatibility with existing Java-based technology stacks.

Get started

Kotlin supports gradual migration of large codebases from Java to Kotlin. You can start writing tests or new production code in Kotlin while keeping other parts of your project in Java.

Configure your Java project to work with Kotlin and make use of the automated Java-to-Kotlin converter included in IntelliJ IDEA:

Introduce Kotlin to your Java project

Explore frameworks

Kotlin is fully compatible with all Java-based frameworks, so you can use your familiar technology stack while benefiting from Kotlin syntax. In addition to great IDE support, Kotlin offers framework-specific tooling, such as support for Spring and Ktor in IntelliJ IDEA with an Ultimate subscription.

Spring

Spring makes use of Kotlin's language features to offer more concise APIs. The online project generator allows you to quickly generate a new project in Kotlin.

Get started with Spring Boot and Kotlin

Ktor

Ktor is a JetBrains framework for creating web applications in Kotlin. It uses coroutines for high scalability and offers an easy-to-use idiomatic API.

Create a new Ktor project

Other frameworks

Here are some other examples of the backend frameworks for Kotlin:

Framework	Description
Quarkus	An open-sourced framework that provides first-class support for Kotlin. Quarkus was built from the ground up for Kubernetes and provides a cohesive full-stack framework by leveraging a growing list of hundreds of best-of-breed libraries.
Vert.x	A framework for building reactive web applications on the JVM. Vert.x offers dedicated support for Kotlin, including integration of Kotlin coroutines.
kotlinx.html	A DSL that can be used to build HTML in web applications. It serves as an alternative to traditional templating systems such as JSP and FreeMarker.
Micronaut	A modern JVM-based full-stack framework for building modular, easily testable microservices and serverless applications. Watch a webinar Micronaut for microservices with Kotlin and explore a detailed guide showing how you can use Kotlin extension functions in the Micronaut framework.

Framework	Description
http4k	A functional toolkit with a small footprint for Kotlin HTTP applications, written in pure Kotlin. http4k provides the CLI-enabled toolbox to generate fully formed project templates, and a web-based project wizard to bootstrap a working http4k application with selected backend, modules, and build tooling.
Javalin	A very lightweight web framework for Kotlin and Java which supports WebSockets, HTTP2, and async requests.

Deploy your applications

Kotlin applications can be deployed to any host that supports Java web applications, including Amazon Web Services (AWS), Google Cloud Platform (GCP), and many others.

- **AWS** provides a dedicated SDK for Kotlin to interact with its services. For serverless deployments, you can refer to AWS Lambda code examples for Kotlin.
- **Ktor** allows you to publish Kotlin applications to various cloud providers. For instance, you can follow Ktor tutorials to learn more about deploying to Google App Engine and other services.
- **Spring** applications are also compatible with most popular cloud providers. See how to deploy Spring Boot applications to the cloud in the official Spring documentation.

Join the community

- **Kotlin slack**: get an invitation and join the #spring and #server channels
- **Stack Overflow**: subscribe to the "kotlin", "spring-kotlin", or "ktor" tags
- **Kotlin YouTube channel**: subscribe and watch videos about Kotlin with Spring

Next steps

- Learn how to test your Java Maven project with Kotlin and JUnit
- Explore how to build asynchronous server applications with Ktor

Adding Kotlin to a Java project – tutorial

Kotlin is fully interoperable with Java, so you can gradually introduce it into your existing Java projects without having to rewrite everything.

In this tutorial, you'll learn how to:

- Set up Maven or Gradle build tools to compile both Java and Kotlin code.
- Organize Java and Kotlin source files in your project directories.
- Convert Java files to Kotlin using IntelliJ IDEA.

You can use any existing Java project for this tutorial or clone our public sample project with both Maven and Gradle build files already set up.

You can also hand over the conversion to your AI agent of choice using our prepared skill. Keep in mind that AI processing results are not entirely predictable.

Project configuration

To add Kotlin to a Java project, you need to configure the project to use both Kotlin and Java, depending on the build tool you use.

The project configuration ensures that both Kotlin and Java code are compiled properly and can reference each other seamlessly.

Maven

Starting with **IntelliJ IDEA 2025.3**, when you add your first Kotlin file to a Maven-based Java project, the IDE automatically updates your `pom.xml` file to include the Kotlin Maven plugin and standard dependencies.
You can still configure it manually if you want to customize versions or build phases.

To use Kotlin and Java together in a Maven project, apply the Kotlin Maven plugin and add Kotlin dependencies in your `pom.xml` file:

1. In the `<properties>` section, add the Kotlin version property:

```
<properties>
        <kotlin.version>2.4.20</kotlin.version>
    </properties>
```

2. In the `<dependencies>` section, add the required dependencies to the `<plugins>` section:

```
<dependencies>
        <!-- Add JUnit Jupiter engine for test runtime -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-params</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

3. In the `<build><plugins>` section, add the Kotlin plugin:

```
<build>
        <pluginManagement><!-- Lock down plugin versions to avoid using Maven defaults (can be
moved to a parent pom file) -->
                <!-- No maven-compiler-plugin needed with Kotlin extensions -->
            </plugins>
            <!-- Activate Kotlin Maven plugin for main and test sources -->
            <plugin>
                <groupId>org.jetbrains.kotlin</groupId>
                <artifactId>kotlin-maven-plugin</artifactId>
                <version>${kotlin.version}</version>
                <extensions>true</extensions>
            </plugin>
        </plugins>
    </build>
```

Enabling `<extensions>true</extensions>` in the Kotlin Maven plugin helps to:

- Automatically add the `kotlin-stdlib` dependency to the project.
- Configure execution phases to compile Kotlin first, then Java.
- Reference Kotlin code in Java code and vice versa.
- Automatically align the JVM target version with the Java compiler version.

You don't need a separate `maven-compiler-plugin` in the `<build><pluginManagement>` section when using the Kotlin Maven plugin with extensions.

4. Reload the Maven project in your IDE.
5. Run tests to verify the configuration:

```
./mvnw clean test
```

Gradle

To use Kotlin and Java together in a Gradle project, apply the Kotlin JVM plugin and add Kotlin dependencies in your `build.gradle.kts` file:

1. In the `plugins {}` block, add the Kotlin JVM plugin:

```
plugins {
    // Other plugins
    kotlin("jvm") version "2.4.20"
}
```

2. Set the JVM toolchain version to match your Java version:

```
kotlin {
    jvmToolchain(17)
}
```

This ensures Kotlin uses the same JDK version as your Java code.

3. In the `dependencies {}` block, add the `kotlin("test")` library that provides Kotlin test utilities and integrates with JUnit:

```
dependencies {
    // Other dependencies

    testImplementation(kotlin("test"))
    // Other test dependencies
}
```

4. Reload the Gradle project in your IDE.
5. Run your tests to verify the configuration:

```
./gradlew clean test
```

Project structure

With this configuration, you can mix Java and Kotlin files in the same source directories:

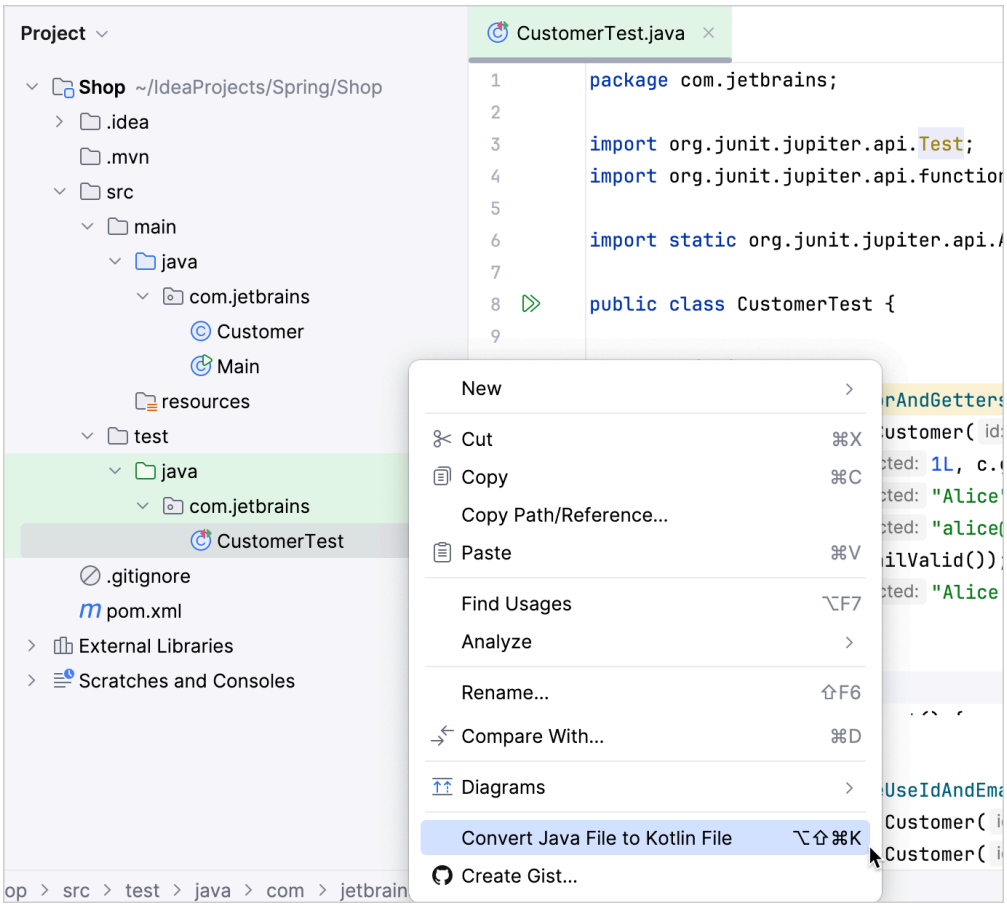
```
src/
  ├── main/
  │    ├── java/          # Java and Kotlin production code
  │    └── kotlin/        # Additional Kotlin production code (optional)
  └── test/
       ├── java/          # Java and Kotlin test code
       └── kotlin/        # Additional Kotlin test code (optional)
```

You can create these directories manually or let IntelliJ IDEA create them when you add your first Kotlin file.

The Kotlin plugin automatically recognizes both `src/main/java` and `src/test/java` directories, so you can keep `.kt` and `.java` files in the same directories.

Convert Java files to Kotlin

The Kotlin plugin also bundles a Java to Kotlin converter (*J2K*) that automatically converts Java files to Kotlin. To use J2K on a file, click **Convert Java File to Kotlin File** in its context menu or in the **Code** menu of IntelliJ IDEA.



Convert Java to Kotlin

While the converter is not fool-proof, it does a pretty decent job of converting most boilerplate code from Java to Kotlin. However, some manual tweaking is sometimes required.

Explore compiler plugins

If you have a more complex project that uses Spring or Java Persistence API (JPA), you can use Kotlin compiler plugins that automatically adapt Kotlin's language features to framework expectations, reducing boilerplate:

- The `all-open` plugin automatically makes classes and their members `open` when used with specific annotations. This is particularly useful for frameworks like Spring that require classes to be non-final.

  For Spring, you can use a dedicated `kotlin-spring` plugin, which is a wrapper on top of `all-open`. It specifies Spring annotations automatically.

- The `no-arg` plugin generates an additional zero-argument constructor for classes with specific annotations. This allows JPA to instantiate classes that otherwise wouldn't have a default constructor.

  You can also use the `kotlin-jpa` plugin, which is a wrapper on top of `no-arg`. It specifies no-arg annotations automatically.

- The `power-assert` plugin improves the debugging experience by providing detailed failure messages with contextual information for assertions. It shows intermediate values and helps you understand why a test failed.

Next step

The easiest way to start using Kotlin in a Java project is by adding Kotlin tests first:

Add your first Kotlin test to your Java project

See also

- Kotlin and Java interoperability details
- Maven build configuration reference

Test Java code using Kotlin and JUnit – tutorial

Kotlin is fully interoperable with Java, which means you can write tests for Java code using Kotlin and run them together with your existing Java tests in the same project.

In this tutorial, you'll learn how to:

- Configure a mixed Java–Kotlin project to run tests using JUnit.
- Add Kotlin tests that verify Java code.
- Run tests using Maven or Gradle.

Before you start, make sure you have:

- IntelliJ IDEA or VS Code with the installed Kotlin extension.
- Java 17 or later.

Configure the project

1. In your IDE, clone the sample project from version control:

   ```
   https://github.com/kotlin-hands-on/kotlin-junit-sample.git
   ```

2. Navigate to the `initial` module and review the project structure:

   ```
   kotlin-junit-sample/
   ├── initial/
   │   ├── src/
   │   │   ├── main/java/    # Java source code
   │   │   └── test/java/    # JUnit test in Java
   │   ├── pom.xml           # Maven configuration
   │   └── build.gradle.kts  # Gradle configuration
   ```

   The `initial` module contains a simple Todo application in Java with a single test.

3. In the same directory, open your build file and update its contents to support Kotlin:

Maven

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-junit-complete</artifactId>
    <version>1.0-SNAPSHOT</version>

    <name>kotlin-junit-complete</name>
    <url>https://kotlinlang.org/docs/jvm-test-using-junit.htm</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.release>17</maven.compiler.release>
        <jexer.version>1.6.0</jexer.version>
        <kotlin.version>2.4.20</kotlin.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.junit</groupId>
                <artifactId>junit-bom</artifactId>
                <version>6.0.3</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <scope>test</scope>
        </dependency>
        <!-- Add JUnit Jupiter engine for test runtime -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <scope>test</scope>
        </dependency>
        <!-- Optionally: parameterized tests support -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-params</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>com.gitlab.klamonte</groupId>
            <artifactId>jexer</artifactId>
            <version>${jexer.version}</version>
        </dependency>
    </dependencies>

    <build>
        <pluginManagement><!-- Lock down plugin versions to avoid using Maven defaults (can be
```

```
moved to a parent pom file) -->
            <plugins>
                <!-- Clean lifecycle, see https://maven.apache.org/ref/current/maven-
core/lifecycles.html#clean_Lifecycle -->
                <plugin>
                    <artifactId>maven-clean-plugin</artifactId>
                    <version>3.4.0</version>
                </plugin>
                <!-- Default lifecycle, jar packaging: see
https://maven.apache.org/ref/current/maven-core/default-
bindings.html#Plugin_bindings_for_jar_packaging -->
                <plugin>
                    <artifactId>maven-resources-plugin</artifactId>
                    <version>3.3.1</version>
                </plugin>
                <plugin>
                    <artifactId>maven-surefire-plugin</artifactId>
                    <version>3.3.0</version>
                </plugin>
                <plugin>
                    <artifactId>maven-jar-plugin</artifactId>
                    <version>3.4.2</version>
                </plugin>
                <plugin>
                    <artifactId>maven-install-plugin</artifactId>
                    <version>3.1.2</version>
                </plugin>
                <plugin>
                    <artifactId>maven-deploy-plugin</artifactId>
                    <version>3.1.2</version>
                </plugin>
                <!-- Site lifecycle, see https://maven.apache.org/ref/current/maven-
core/lifecycles.html#site_Lifecycle -->
                <plugin>
                    <artifactId>maven-site-plugin</artifactId>
                    <version>3.12.1</version>
                </plugin>
                <plugin>
                    <artifactId>maven-project-info-reports-plugin</artifactId>
                    <version>3.6.1</version>
                </plugin>
                <!-- No maven-compiler-plugin needed with Kotlin extensions -->
            </plugins>
        </pluginManagement>
        <plugins>
            <!-- Activate Kotlin Maven plugin for main and test sources -->
            <plugin>
                <groupId>org.jetbrains.kotlin</groupId>
                <artifactId>kotlin-maven-plugin</artifactId>
                <version>${kotlin.version}</version>
                <extensions>true</extensions>
            </plugin>
        </plugins>
    </build>
</project>
```

- In the `<properties>` section, set the Kotlin version.
- In the `<dependencies>` section, add JUnit Jupiter dependencies to run tests.
- In the `<build><plugins>` section, apply `kotlin-maven-plugin` with `<extensions>` set to `true`. It automatically adds corresponding executions and the `kotlin-stdlib` dependency to the build.

- You don't need to add `maven-compiler-plugin` to the `<build><pluginManagement>` section when using the Kotlin Maven plugin with extensions.

Gradle

```
// build.gradle.kts
group = "org.jetbrains.kotlin"
version = "1.0-SNAPSHOT"
description = "kotlin-junit-complete"
java.sourceCompatibility = JavaVersion.VERSION_17

plugins {
    application
    kotlin("jvm") version "2.4.20"
}

kotlin {
    jvmToolchain(17)
}

application {
    mainClass.set("org.jetbrains.kotlin.junit.App")
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("com.gitlab.klamonte:jexer:1.6.0")

    testImplementation(kotlin("test"))
    testImplementation(libs.org.junit.jupiter.junit.jupiter.api)
    testImplementation(libs.org.junit.jupiter.junit.jupiter.params)
    testRuntimeOnly(libs.org.junit.jupiter.junit.jupiter.engine)
    testRuntimeOnly(libs.org.junit.platform.junit.platform.launcher)
}

tasks.test {
    useJUnitPlatform()
}
```

- In the `plugins {}` block, add the `kotlin("jvm")` plugin.
- Set the JVM toolchain version to match your Java version.
- In the `dependencies {}` block, add the `kotlin.test` library that provides Kotlin's test utilities and integrates with JUnit.

Kotlin/JVM supports the latest stable JUnit version, JUnit 6. You can find it in the `gradle/libs.versions.toml` version catalog.

If you generally prefer using the version catalog, you can even add the `kotlin("jvm")` plugin there:

```
# gradle/libs.versions.toml
[versions]
kotlin = "2.4.20"
junit = "6.0.3"

[libraries]
org-junit-jupiter-junit-jupiter-api = { module = "org.junit.jupiter:junit-jupiter-api", version.ref
= "junit" }
org-junit-jupiter-junit-jupiter-params = { module = "org.junit.jupiter:junit-jupiter-params",
version.ref = "junit" }
org-junit-jupiter-junit-jupiter-engine = { module = "org.junit.jupiter:junit-jupiter-engine",
version.ref = "junit" }
org-junit-platform-junit-platform-launcher = { module = "org.junit.platform:junit-platform-
launcher" }

[plugins]
kotlinJvm = { id = "org.jetbrains.kotlin.jvm", version.ref = "kotlin" }
```

4. Reload the build file in your IDE.

For more detailed instructions on build file setup, see Project configuration.

Add your first Kotlin test

The `TodoItemTest.java` test in `initial/src/test/java` already verifies the app basics: item creation, defaults, unique IDs, and state changes.

You can expand the test coverage by adding a Kotlin test that verifies repository-level behavior:

1. Navigate to the same test source directory, `initial/src/test/java`.
2. Create a `TodoRepositoryTest.kt` file in the same package as the Java test.
3. Create the test class with field declarations and setup function:

```
package org.jetbrains.kotlin.junit

import org.junit.jupiter.api.BeforeEach
import org.junit.jupiter.api.Assertions
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.DisplayName

internal class TodoRepositoryTest {
    lateinit var repository: TodoRepository
    lateinit var testItem1: TodoItem
    lateinit var testItem2: TodoItem

    @BeforeEach
    fun setUp() {
        repository = TodoRepository()
        testItem1 = TodoItem("Task 1", "Description 1")
        testItem2 = TodoItem("Task 2", "Description 2")
    }
}
```

- JUnit annotations work the same in Kotlin as in Java.
- In Kotlin, the `lateinit` keyword allows declaring non-null properties that are initialized later. This helps to avoid having to use nullable types (`TodoRepository?`) in your tests.

4. Add a test inside the `TodoRepositoryTest` class to check the initial repository state and its size:

```
@Test
@DisplayName("Should start with empty repository")
fun shouldStartEmpty() {
    Assertions.assertEquals(0, repository.size())
    Assertions.assertTrue(repository.all.isEmpty())
}
```

- Unlike Java static import, Jupiter's `Assertions` is imported as a class and used as a qualifier for assertion functions.
- Instead of `.getAll()` calls, you can access Java getters as properties in Kotlin with `repository.all`.

5. Write another test to verify copy behavior for all items:

```
@Test
@DisplayName("Should return defensive copy of items")
fun shouldReturnDefensiveCopy() {
    repository.add(testItem1)

    val items1 = repository.all
    val items2 = repository.all

    Assertions.assertNotSame(items1, items2)
    Assertions.assertThrows(
        UnsupportedOperationException::class.java
    ) { items1.clear() }
    Assertions.assertEquals(1, repository.size())
}
```

- To get a Java class object from a Kotlin class, use `::class.java`.
- You can split complex assertions across multiple lines without using any special continuation characters.

6. Add a test to verify finding items by ID:

```
@Test
@DisplayName("Should find item by ID")
fun shouldFindItemById() {
    repository.add(testItem1)
    repository.add(testItem2)

     val found = repository.getById(testItem1.id())

     Assertions.assertTrue(found.isPresent)
     Assertions.assertEquals(testItem1, found.get())
}
```

Kotlin works smoothly with the Java `Optional` API. It automatically converts getter methods to properties, that's why the `isPresent()` method is accessed here as a property.

7. Write a test to verify the item removal mechanism:

```kotlin
@Test
 @DisplayName("Should remove item by ID")
 fun shouldRemoveItemById() {
     repository.add(testItem1)
     repository.add(testItem2)

     val removed = repository.remove(testItem1.id())

     Assertions.assertTrue(removed)
     Assertions.assertEquals(1, repository.size())
     Assertions.assertTrue(repository.getById(testItem1.id()).isEmpty)
     Assertions.assertTrue(repository.getById(testItem2.id()).isPresent)
 }

 @Test
 @DisplayName("Should return false when removing non-existent item")
 fun shouldReturnFalseForNonExistentRemoval() {
     repository.add(testItem1)

     val removed = repository.remove("non-existent-id")

     Assertions.assertFalse(removed)
     Assertions.assertEquals(1, repository.size())
 }
```

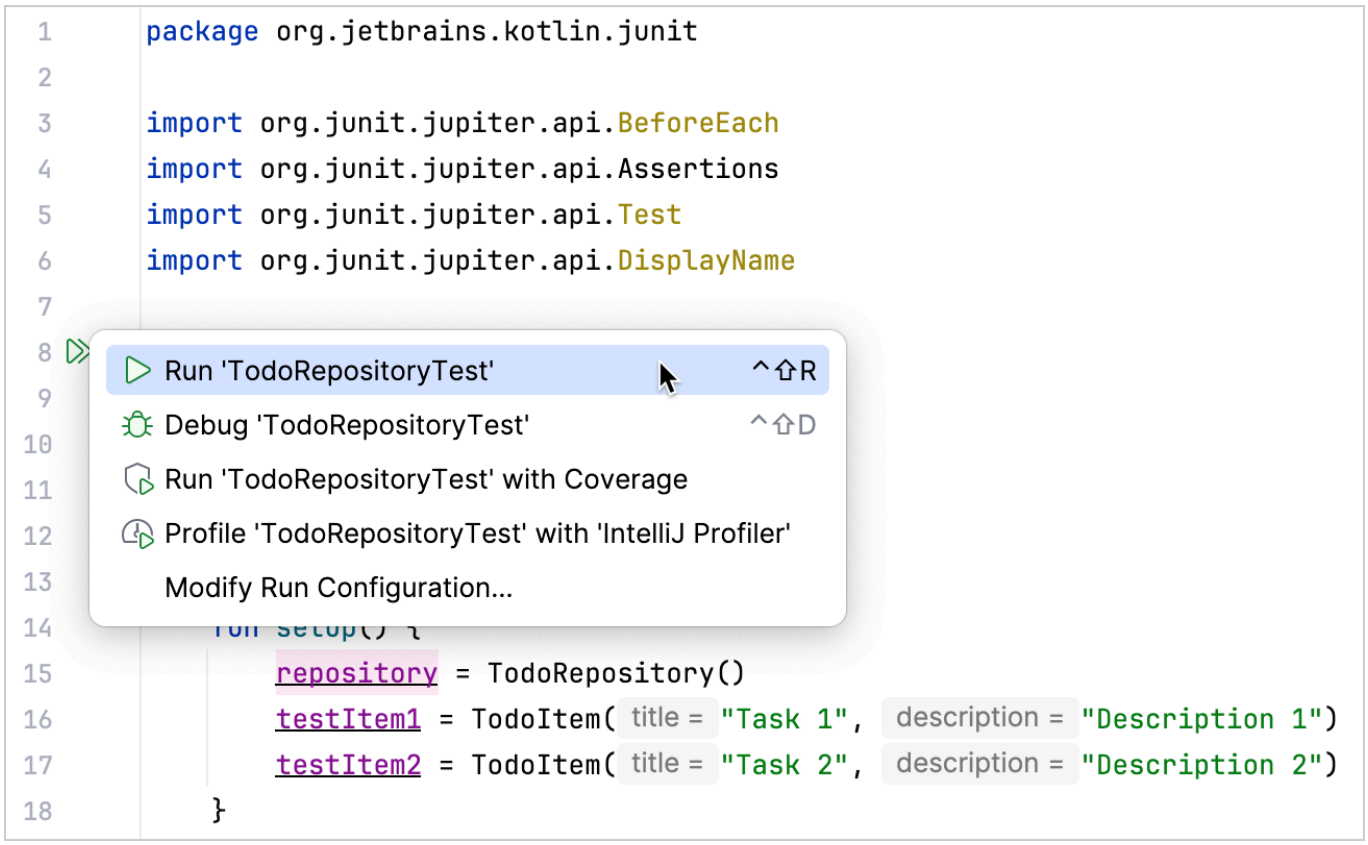
In Kotlin, you can chain method calls and property access, for example `repository.getById(id).isEmpty`.

> You can add even more tests to the `TodoRepositoryTest` test class to cover additional functionality. See the full source code in the sample project's `complete` module.

Run tests

Run both Java and Kotlin tests to verify your project works as expected:

1. Run the test using the gutter icon:



Run the test

You can also run all project tests from the `initial` directory using the command line:

Maven

```
mvn test
```

Gradle

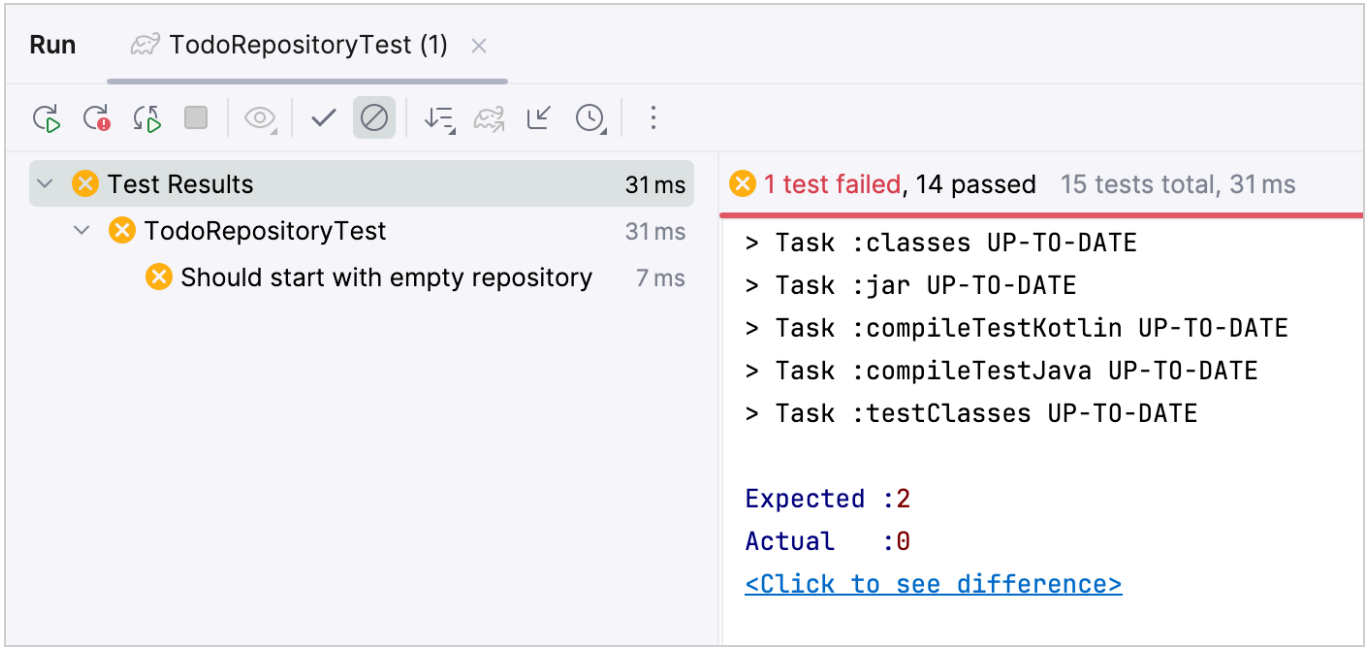
```
./gradlew test
```

2. Check that the test works correctly by changing one of the variable values. For example, modify the `shouldAddItem` test to expect an incorrect repository size:

```
@Test
@DisplayName("Should add item to repository")
fun shouldAddItem() {
    repository.add(testItem1)

    Assertions.assertEquals(2, repository.size())  // Changed from 1 to 2
    Assertions.assertTrue(repository.all.contains(testItem1))
}
```

3. Run the test again and verify that it fails:



Check the test result. The test has failed

You can find the fully configured project with tests in the sample project's `complete` module.

What's next

Learn more about testing Kotlin projects with Maven.

Test Kotlin projects with Maven

Kotlin integrates seamlessly with the Maven ecosystem, allowing you to use industry-standard tools to verify your backend applications. In this guide, you'll learn how to create tests with JUnit and use Maven plugins to run unit and integration tests.

For a detailed guide to setting up your Maven project to use both Kotlin and Java, see Project configuration.

Create tests with JUnit

JUnit is the standard testing framework for Kotlin backend development. While Kotlin supports multiple JUnit versions, most modern projects should use JUnit 6.

To create a test in Kotlin using JUnit, use the `@Test` annotation from the `kotlin.test` or JUnit package.

Add dependency

The `kotlin-test` library is the easiest way to start. It provides a common set of assertions and automatically pulls in the necessary JUnit artifacts.

JUnit 5 and later

For all new projects, use the `kotlin-test-junit5` artifact. It provides full support for JUnit, including features like nested tests and parallel execution. Kotlin/JVM supports the latest stable JUnit version, JUnit 6.

Update your `pom.xml` file as follows:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-test-junit5</artifactId>
        <version>2.4.20</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Despite its name, `kotlin-test-junit5` supports all latest JUnit versions, including JUnit 6.

JUnit 4

If you'd like to use an earlier version of JUnit, for example, for a legacy project, use the `kotlin-test-junit` artifact that utilizes JUnit 4:

```
<!-- pom.xml -->
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-test-junit</artifactId>
        <version>2.4.20</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

For a detailed guide on using JUnit for testing and a sample project, see the Test Java code with Kotlin tutorial.

Write unit tests

Unit tests verify isolated parts of your code, such as individual functions or classes. By convention, unit tests are named with the `*Test` suffix. For example:

```
import kotlin.test.Test
import kotlin.test.assertEquals

class OrderServiceTest {
    @Test
    fun `calculate total should sum item prices`() {
        val service = OrderService()
        val result = service.calculateTotal(listOf(10.0, 25.0))
        assertEquals(35.0, result)
    }
}
```

Write integration tests

Integration tests verify interaction between components, like a service and a database. By convention, integration tests are named with the `*IT` suffix. For example:

```
import kotlin.test.Test
import kotlin.test.assertNotNull

class UserRepositoryIT {
    @Test
    fun saveFindUser() {
        // Example integration with a database or service
        val repository = UserRepository()
        repository.save(User("KotlinUser"))

        val user = repository.findByName("KotlinUser")
        assertNotNull(user)
    }
}
```

Run tests

In Maven projects, test execution is typically split between two plugins: Surefire and Failsafe to ensure a clean build lifecycle.

With Surefire plugin

The Surefire plugin handles *unit tests*. It runs all Kotlin and Java tests that follow the `*Test` naming pattern.

By default, it executes during the `test` phase of the build lifecycle and fails the build immediately if a test fails.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.5.5</version>
</plugin>
```

To run only your unit tests, use the following command:

```
mvn test
```

With Failsafe plugin

The Failsafe plugin handles *integration tests*. It runs all Kotlin and Java tests that follow the `*IT` naming pattern.

Unlike Surefire, Failsafe allows the build to continue if a test fails during the `integration-test` phase, allowing the `post-integration-test` phase tasks (like stopping a Docker container) to run. The build finally fails during the `verify` phase if there were any test failures.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>3.5.5</version>
    <executions>
        <execution>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

To run both unit and integration tests, use the following command:

```
mvn verify
```

Get detailed failure messages

The Kotlin Power-assert compiler plugin generates detailed failure messages that display intermediate values in your assertions, giving you the full diagrams in the console output.

1. To enable Power-assert, add the following configuration to your project's `pom.xml` :

```
<properties>
    <kotlin.version>2.4.20</kotlin.version>
    <kotlin.compiler.jdkRelease>17</kotlin.compiler.jdkRelease>
</properties>

<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-test-junit5</artifactId>
        <version>${kotlin.version}</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <extensions>true</extensions>
            <executions>
                <execution>
                    <id>compile</id>
                    <phase>process-sources</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>test-compile</id>
                    <phase>process-test-sources</phase>
                    <goals>
                        <goal>test-compile</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <!-- Specify the Power-assert plugin -->
                <compilerPlugins>
                    <plugin>power-assert</plugin>
                </compilerPlugins>
                <!-- Specify the functions to transform -->
                <pluginOptions>
                    <option>power-assert:function=kotlin.test.assertEquals</option>
                    <option>power-assert:function=kotlin.test.assertTrue</option>
                </pluginOptions>
            </configuration>
            <!-- Add the Power-assert plugin dependency -->
            <dependencies>
                <dependency>
                    <groupId>org.jetbrains.kotlin</groupId>
                    <artifactId>kotlin-maven-power-assert</artifactId>
                    <version>${kotlin.version}</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
```

- In the `<properties>` section, set the Kotlin version and the target JDK release version, for example, JDK 17.
- In the `<dependencies>` section, add `kotlin-test-junit5` for running tests.

- In the `<build><plugins>` section, configure `kotlin-maven-plugin` with the `power-assert` compiler plugin. List the assertion functions you want to transform under `<pluginOptions>` and add the Power-assert plugin dependency.

2. Create a test file, `UserProfileBackendTest.kt`, in the `src/test/kotlin` directory:

```
import kotlin.test.Test
import kotlin.test.assertEquals

class UserProfileBackendTest {

    data class UserProfile(val id: Long, val email: String)

    @Test
    fun `verify backend entity mapping matches`() {
        // Define expected object state
        val expectedRecord = UserProfile(id = 451L, email = "admin-dev@company.internal")

        // Simulate a mismatched backend response
        val actualRecord = UserProfile(id = 451L, email = "admin-prod@company.com")

        // Use a standard kotlin.test assertion
        assertEquals(expectedRecord, actualRecord, "Profile configurations out of sync")
    }
}
```

3. Run the test and verify the output:

```
mvn test
```

The console displays the full Power-assert diagram, showing the difference between the expected and actual values at each position in the expression:

```
Profile configurations out of sync
assertEquals(expectedRecord, actualRecord, "Profile configurations out of sync")
             |               |
             |               UserProfile(id=451, email=admin-prod@company.com)
             UserProfile(id=451, email=admin-dev@company.internal)
```

Explore other testing frameworks

Besides JUnit, you can use other popular frameworks to make Kotlin tests more idiomatic and readable:

Library	Description
AssertJ	Fluent assertion library with chainable assertions.
Mockito-Kotlin	Kotlin wrapper for Mockito that provides helper functions and better integration with the Kotlin type system.
MockK	Native Kotlin mocking library that supports Kotlin-specific features, including coroutines and extension functions.

Library	Description
Kotest	Assertion library for Kotlin offering multiple assertion styles and extensive matcher support.
Strikt	Assertion library for Kotlin with type-safe assertions and support for data classes.

What's next

- Explore the features of the `kotlin.test` library.
- Learn more about the Power-assert compiler plugin.

Use annotation processors in Kotlin projects

Annotation processors analyze your source code at compile time to generate boilerplate code, validate usage, or produce other artifacts. Kotlin supports two ways to work with annotation processors:

- The kapt compiler plugin works by generating stub files from Kotlin source code and then running the Java annotation processors on those stubs. This extra stub-generation step makes the build time slower and means kapt can't understand Kotlin-specific constructs, such as extension functions or null safety.

  kapt supports both Maven and Gradle. It's recommended for all Maven projects and for Gradle projects with processor libraries that haven't yet adopted KSP, such as MapStruct.

- The KSP framework reads Kotlin source code directly through a Kotlin-first API, without generating stubs. It understands Kotlin-specific features natively and runs builds faster than kapt.

  Currently, KSP has official support only for Gradle. It's recommended for writing your own processors and working with KSP-compatible libraries like Dagger.

Use kapt with Java annotation processors

kapt lets you use existing Java annotation processors in Kotlin projects without any changes to the processors themselves.

The example below shows how to use the MapStruct annotation processor, which generates type-safe mapper implementations between Java beans at compile time.

1. In your build file, apply the `kapt` plugin and add MapStruct to the `dependencies` section:

Maven

```xml
<properties>
    <kotlin.compiler.jvmTarget>11</kotlin.compiler.jvmTarget>
    <mapstruct.version>1.6.3</mapstruct.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct</artifactId>
        <version>${mapstruct.version}</version>
    </dependency>
</dependencies>

<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <extensions>true</extensions>
    <executions>
        <execution>
            <id>kapt</id>
            <goals>
                <goal>kapt</goal>
            </goals>
            <configuration>
                <sourceDirs>
                    <sourceDir>src/main/kotlin</sourceDir>
                    <sourceDir>src/main/java</sourceDir>
                </sourceDirs>
                <aptMode>stubs</aptMode>
                <annotationProcessorPaths>
                    <annotationProcessorPath>
                        <groupId>org.mapstruct</groupId>
                        <artifactId>mapstruct-processor</artifactId>
                        <version>${mapstruct.version}</version>
                    </annotationProcessorPath>
                </annotationProcessorPaths>
            </configuration>
        </execution>
    </executions>
</plugin>
```

- Add the execution of the `kapt` goal from `kotlin-maven-plugin` **before** the `compile` execution.
- Configure the mode of annotation processing using the `aptMode` option.

Gradle Kotlin

```kotlin
plugins {
    kotlin("kapt") version "2.4.20"
}

dependencies {
    implementation("org.mapstruct:mapstruct:1.6.3")
    kapt("org.mapstruct:mapstruct-processor:1.6.3")
}
```

Gradle Groovy

```
plugins {
    id "org.jetbrains.kotlin.kapt" version "2.4.20"
}

dependencies {
    implementation "org.mapstruct:mapstruct:1.6.3"
    kapt "org.mapstruct:mapstruct-processor:1.6.3"
}
```

2. Define your data classes and a mapper interface:

```
import org.mapstruct.Mapper
import org.mapstruct.factory.Mappers

data class UserDto(val id: Long, val firstName: String, val lastName: String)

data class UserEntity(val id: Long, val firstName: String, val lastName: String)

@Mapper
interface UserMapper {
    fun toDto(entity: UserEntity): UserDto
    fun toEntity(dto: UserDto): UserEntity

    companion object : UserMapper by Mappers.getMapper(UserMapper::class.java)
}
```

3. Build the project. MapStruct generates the `UserMapperImpl` class in the generated sources' directory. Use the `UserMapper` companion object to call the generated implementation:

```
fun main() {
    val entity = UserEntity(id = 1L, firstName = "John", lastName = "Doe")
    val dto = UserMapper.toDto(entity)
    println(dto)
    // UserDto(id=1, firstName=John, lastName=Doe)
}
```

Use KSP in Gradle projects

With KSP, you can use existing annotation processors in Gradle projects and create your own processors that generate code based on annotations in your source code.

Use KSP with Java annotation processors

For Gradle projects, use KSP with the compatible annotation processors. KSP is faster than kapt and can understand Kotlin-specific features natively. See the list of libraries that already support KSP.

The example below shows how to use Dagger, a compile-time dependency injection framework that generates the wiring code for your dependency graph.

1. In your `build.gradle(.kts)` file, apply the KSP plugin and add Dagger to the `dependencies` block:

Kotlin

```
// build.gradle.kts

plugins {
    kotlin("jvm") version "2.4.20"
    id("com.google.devtools.ksp") version "2.3.10"
}

dependencies {
    implementation("com.google.dagger:dagger:2.59.2")
    ksp("com.google.dagger:dagger-compiler:2.59.2")
}
```

Groovy

```
// build.gradle

plugins {
    id 'org.jetbrains.kotlin.jvm' version '2.4.20'
    id 'com.google.devtools.ksp' version '2.3.10'
}

dependencies {
    implementation 'com.google.dagger:dagger:2.59.2'
    ksp 'com.google.dagger:dagger-compiler:2.59.2'
}
```

ℹ To find the latest version of KSP, check the GitHub Releases page.

2. Annotate your Kotlin classes with Dagger annotations:

```
import javax.inject.Inject
import javax.inject.Singleton
import dagger.Component
import dagger.Module
import dagger.Provides

@Singleton
class UserRepository @Inject constructor() {
    fun getUser(): String = "John Doe"
}

@Module
class AppModule {
    @Provides
    @Singleton
    fun provideUserRepository(): UserRepository = UserRepository()
}

@Singleton
@Component(modules = [AppModule::class])
interface AppComponent {
    fun userRepository(): UserRepository
}
```

3. Build the project. Dagger generates implementation classes, such as `DaggerAppComponent` in the `build/generated/ksp` directory. Use the generated class in your code:

```
fun main() {
    val appComponent = DaggerAppComponent.create()
    val userRepository = appComponent.userRepository()
    println("User: ${userRepository.getUser()}")
    // User: John Doe
}
```

For more information on Dagger support for KSP, see its documentation.

Create your own annotation processor

You can use the KSP API to write your own annotation processors that generate code at compile time. A new processor requires three modules:

- An `annotation` module that declares the custom annotation.
- A `processor` module that implements the `SymbolProcessor` and `SymbolProcessorProvider` factories. `SymbolProcessor` contains the main logic, while `SymbolProcessorProvider` creates the processor and registers the provider in the `META-INF/services/` path.
- An `app` module that applies the KSP plugin depends on the processor, and uses the annotation.

For complete step-by-step instructions, see the KSP quickstart.

What's next

- Learn more about kapt configuration
- Get started with KSP
- See how to migrate from kapt to KSP

Code quality tools in Kotlin projects

Code quality tools help you enforce coding standards, detect bugs early, measure test coverage, and maintain code quality across your Kotlin projects. This guide shows how to integrate popular tools like ktlint, detekt, SonarQube, SonarCloud, and Kover into your backend project for Maven or Gradle.

Code formatting with ktlint

ktlint is a Kotlin linter and formatter that enforces the official Kotlin coding conventions with no extra configuration.

ktlint checks rules such as indentation, spacing around operators, import ordering, and trailing commas. If a violation is found, the build fails with a message indicating the file and line number. Besides reporting violations, ktlint can also automatically fix straightforward issues.

To integrate ktlint into your project:

1. Add the plugin to your build file:

Maven

```
<!-- pom.xml -->
<plugin>
    <groupId>com.github.gantsign.maven</groupId>
    <artifactId>ktlint-maven-plugin</artifactId>
    <version>3.7.1</version>
    <executions>
        <execution>
            <id>check</id>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Gradle

```
// build.gradle.kts
plugins {
    id("org.jlleitschuh.gradle.ktlint") version "12.3.0"
}
```

2. Run the linter to check your code style:

Maven

```
mvn ktlint:check
```

Gradle

```
./gradlew ktlintCheck
```

3. (Optional) You can also add an `.editorconfig` file to the root of your project to customize rules. For example, to allow wildcard imports and disable trailing comma enforcement:

```
[*.{kt,kts}]
ij_kotlin_imports_layout = *
ktlint_standard_trailing-comma-on-call-site = disabled
ktlint_standard_trailing-comma-on-declaration-site = disabled
```

By default, ktlint follows the official Kotlin coding conventions. If you prefer the Android Kotlin style guide, which differs noticeably from the Kotlin conventions, set the code style to `android_studio` in your `.editorconfig` file:

```
[*.{kt,kts}]
ktlint_code_style = android_studio
```

4. To automatically fix formatting issues, run:

Maven

```
mvn ktlint:format
```

Gradle

```
./gradlew ktlintFormat
```

For more information on available features and rules, see the ktlint documentation.

Code analysis with detekt

detekt is a static code analysis tool for Kotlin that detects code smells, complexity issues, and potential bugs.

To integrate detekt into your project:

1. Add the plugin to your build file:

Maven

```
<!-- pom.xml -->
<plugin>
    <groupId>com.github.ozsie</groupId>
    <artifactId>detekt-maven-plugin</artifactId>
    <version>1.23.8</version>
    <executions>
        <execution>
            <phase>verify</phase>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Gradle

```
// build.gradle.kts
plugins {
    id("io.gitlab.arturbosch.detekt") version "1.23.8"
}
```

2. Generate a default `detekt.yml` configuration file:

Maven

```
mvn detekt:generate-config
```

Gradle

```
./gradlew detektGenerateConfig
```

3. Open the `detekt.yml` file and customize the rules in the generated file, for example:

```
complexity:
  LongMethod:
    threshold: 50
style:
  MagicNumber:
    active: false
```

4. Reference the configuration file in your build file so that detekt can apply the new rules:

Maven

```
<!-- pom.xml -->
<plugin>
    <groupId>com.github.ozsie</groupId>
    <artifactId>detekt-maven-plugin</artifactId>
    <version>1.23.8</version>
    <configuration>
        <config>detekt.yml</config>
        <report>
            <report>txt:reports/detekt.txt</report>
            <report>xml:reports/detekt.xml</report>
        </report>
    </configuration>
    <executions>
        <execution>
            <phase>verify</phase>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Gradle

```
// build.gradle.kts
detekt {
    toolVersion = "1.23.8"
    config.setFrom(file("config/detekt/detekt.yml"))
    buildUponDefaultConfig = true
}
```

5. Run the analysis:

Maven

```
mvn detekt:check
```

Gradle

```
./gradlew detekt
```

detekt produces a report listing all rule violations with their severity, file location, and a description of the issue. By default, Gradle outputs reports to `build/reports/detekt`, while Maven outputs to the `reports/detekt` directory in the project root.

For more information, see the detekt documentation for Gradle and Maven.

Code quality with SonarSource

SonarQube and SonarCloud from SonarSource provide deep static analysis for Kotlin projects, including bug detection, vulnerability scanning, and code coverage tracking through a web dashboard.

To analyze your project with SonarQube:

1. Add the plugin to your build file:

Maven

```
<!-- pom.xml -->
<plugin>
    <groupId>org.sonarsource.scanner.maven</groupId>
    <artifactId>sonar-maven-plugin</artifactId>
    <version>5.7.0.6970</version>
</plugin>
```

Gradle

```
// build.gradle.kts
plugins {
    id("org.sonarqube") version "6.2.0.5505"
}

sonar {
    properties {
        property("sonar.projectKey", "my-project")
        property("sonar.host.url", "http://localhost:9000")
    }
}
```

2. (Optional) Configure analysis properties. For example, to make the build wait for the quality gate result and fail if the gate is not passed, add the `sonar.qualitygate.wait` property:

Maven

```
mvn verify sonar:sonar \
  -Dsonar.qualitygate.wait=true \
  -Dsonar.projectKey=my-project \
  -Dsonar.host.url=http://localhost:9000 \
  -Dsonar.token=YOUR_TOKEN
```

Gradle

```
// build.gradle.kts
sonar {
    properties {
        property("sonar.qualitygate.wait", "true")
    }
}
```

Quality gate rules (such as minimum coverage thresholds and allowed issue counts) are defined in the SonarQube or SonarCloud web interface under **Quality Gates**, not in the build file.

3. Run the analysis against your SonarQube server:

Maven

```
mvn verify sonar:sonar \
  -Dsonar.projectKey=my-project \
  -Dsonar.host.url=http://localhost:9000 \
  -Dsonar.token=YOUR_TOKEN
```

For SonarCloud, replace the host URL with `https://sonarcloud.io` and provide your organization key:

```
mvn verify sonar:sonar \
  -Dsonar.projectKey=my-project \
  -Dsonar.organization=my-org \
  -Dsonar.host.url=https://sonarcloud.io \
  -Dsonar.token=YOUR_TOKEN
```

To run the analysis, use the `sonar` task and provide your authentication token:

Gradle

```
./gradlew sonar \
  -Dsonar.token=YOUR_TOKEN
```

By default, the analysis is run against a local SonarQube server. To use SonarCloud, update the `sonar {}` block in your `build.gradle.kts` to use `https://sonarcloud.io` and add your organization key:

```
sonar {
    properties {
        property("sonar.projectKey", "example-project")
        property("sonar.organization", "example-org")
        property("sonar.host.url", "https://sonarcloud.io")
    }
}
```

4. Open the SonarQube or SonarCloud dashboard to review the results. The dashboard shows issues grouped by type (bug, vulnerability, code smell) and severity.

For more information, see the SonarSource documentation.

Code coverage with Kover

Kover is the official JetBrains code coverage tool for Kotlin. It measures which lines and branches of your code are covered by tests and generates human-readable reports.

Unlike JaCoCo, Kover correctly interprets Kotlin-specific constructs such as inline functions and data classes, so it reports accurate coverage numbers without false negatives caused by compiler-generated bytecode.

To integrate Kover into your project:

1. Add the plugin to your build file:

Maven

```
<!-- pom.xml -->
<plugin>
    <groupId>org.jetbrains.kotlinx</groupId>
    <artifactId>kover-maven-plugin</artifactId>
    <version>0.9.8</version>
    <executions>
        <execution>
            <id>instr</id>
            <goals>
                <goal>instrumentation</goal>
            </goals>
        </execution>
        <execution>
            <id>kover-verify</id>
            <goals>
               <goal>verify</goal>
            </goals>
        </execution>
        <execution>
            <id>kover-report-xml</id>
            <goals>
                <goal>report-xml</goal>
            </goals>
        </execution>
        <execution>
            <id>kover-report-html</id>
            <goals>
                <goal>report-html</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Gradle

```
// build.gradle.kts
plugins {
    id("org.jetbrains.kotlinx.kover") version "0.9.8"
}
```

2. Run your tests to collect coverage data and generate the report:

Maven

```
mvn verify
```

Gradle

```
./gradlew koverVerify koverHtmlReport
```

3. Open the HTML report generated in the `target/site/kover/html/` directory (`build/reports/kover/html/` for Gradle) to review line-by-line coverage.
4. (Optional) To enforce a minimum coverage threshold that fails the build if the conditions are not met, you can add a coverage verification configuration to your build file. For example:

Maven

```
<!-- pom.xml -->
<configuration>
  <!-- Create new coverage verification rule -->
  <rules>
      <rule>
          <bounds>
              <bound>
                  <minValue>50</minValue>
                  <coverageUnits>LINE</coverageUnits>
                  <aggregationForGroup>COVERED_PERCENTAGE</aggregationForGroup>
              </bound>
          </bounds>
      </rule>
  </rules>
</configuration>
```

Gradle

```
// build.gradle.kts
// Add a `kover` configuration block
import kotlinx.kover.gradle.plugin.dsl.*

kover {
    reports {
        total {
            log {
                aggregationForGroup = AggregationType.COVERED_PERCENTAGE
                coverageUnits = CoverageUnit.LINE
            }
            verify {
                rule {
                    minBound(50)
                }
            }
        }
    }
}
```

For more information on configuring verification rules, see the Kover documentation for Maven and Gradle.

Other tools

Besides ktlint, detekt, SonarQube, SonarCloud, and Kover, try out other tools to improve Kotlin code quality:

Tool	Description
CodeQL	Semantic code analysis engine by GitHub. Supports Kotlin and integrates with GitHub Actions to find security vulnerabilities automatically.
Semgrep	Fast, lightweight static analysis tool that supports custom rules. Can be used to enforce patterns or detect anti-patterns in Kotlin code.
PMD	Source code analyzer that supports Kotlin (via its CPD copy-paste detector) and detects common programming flaws and duplicate code.

What's next

- Run tests in Maven projects
- Configure a Kotlin project with Maven

Get started with Spring Boot and Kotlin

Get started with Spring Boot and Kotlin by completing this tutorial: it walks you through the process of creating a simple application with Spring Boot and adding a database to store the information.

Going through these four steps, you'll learn a lot of essential features of the Kotlin language:

1. Create a Spring Boot project
2. Add a data class to Spring Boot project
3. Add database support for the Spring Boot project
4. Use Spring Data CrudRepository for database access

Next step

Start

See also

If you like to work with AI agents, try our tutorial for creating a task manager application with Spring Boot and Claude.

Look through our Java to Kotlin (J2K) interop and migration guides:

- Calling Java from Kotlin and Calling Kotlin from Java
- Collections in Java and Kotlin
- Strings in Java and Kotlin

Create a Spring Boot project with Kotlin

The first part of the tutorial shows how to create a Spring Boot project with Gradle in IntelliJ IDEA using the Project Wizard.

> This tutorial doesn't require using Gradle as the build system. You can follow the same steps if you use Maven.

Before you start

Download and install the latest version of IntelliJ IDEA and use an Ultimate subscription.

> If you use IntelliJ IDEA without an Ultimate subscription or you use another IDE, you can generate a Spring Boot project using a web-based project generator.

Create a Spring Boot project

Create a new Spring Boot project with Kotlin by using the Project Wizard in IntelliJ IDEA:

1. In IntelliJ IDEA, select **File** | **New** | **Project**.
2. In the panel on the left, select **Spring Boot** in the **Generators** section.
3. Specify the following fields and options in the **New Project** window:
   - **Name**: demo
   - **Language**: Kotlin
   - **Type**: Gradle - Kotlin

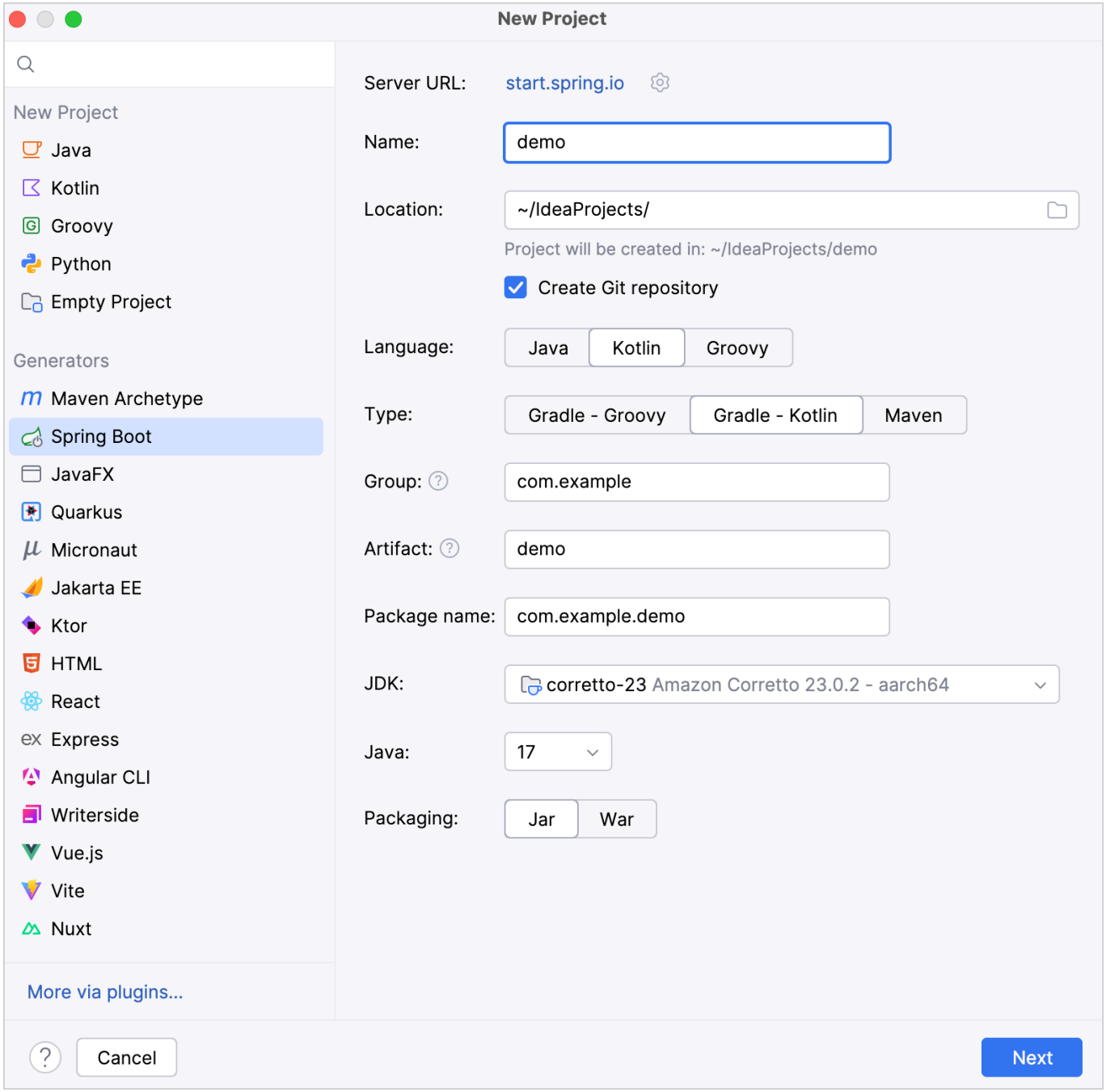
     > This option specifies the build system and the DSL.

   - **Package name**: com.example.demo
   - **JDK**: Java JDK

     > This tutorial uses **Amazon Corretto version 23**. If you don't have a JDK installed, you can download it from the dropdown list.

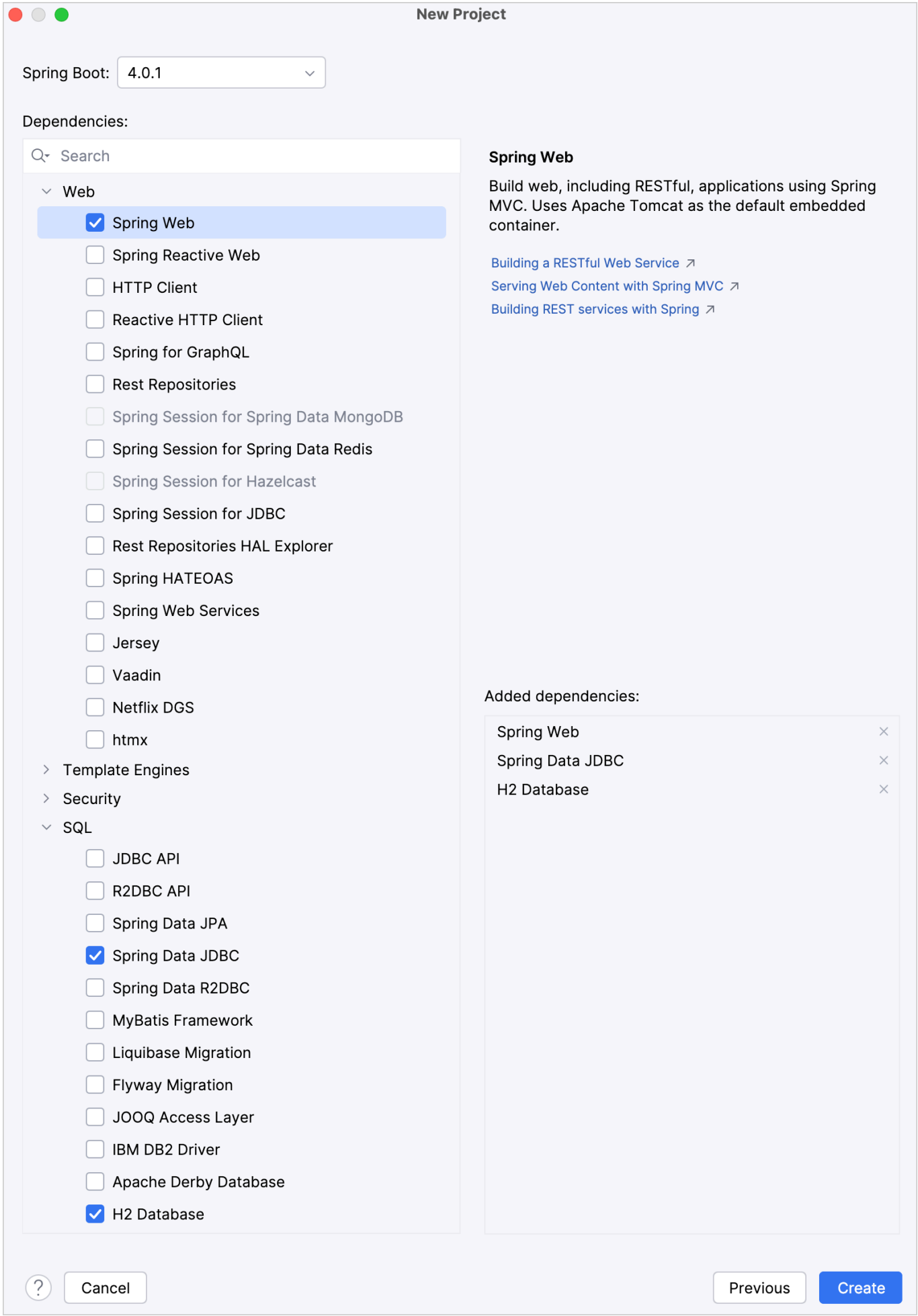
   - **Java**: 17

     > If you don't have Java 17 installed, you can download it from the JDK dropdown list.



Create Spring Boot project

4. Ensure that you have specified all the fields and click **Next**.
5. Select the following dependencies that will be required for the tutorial:
   - **Web | Spring Web**
   - **SQL | Spring Data JDBC**
   - **SQL | H2 Database**

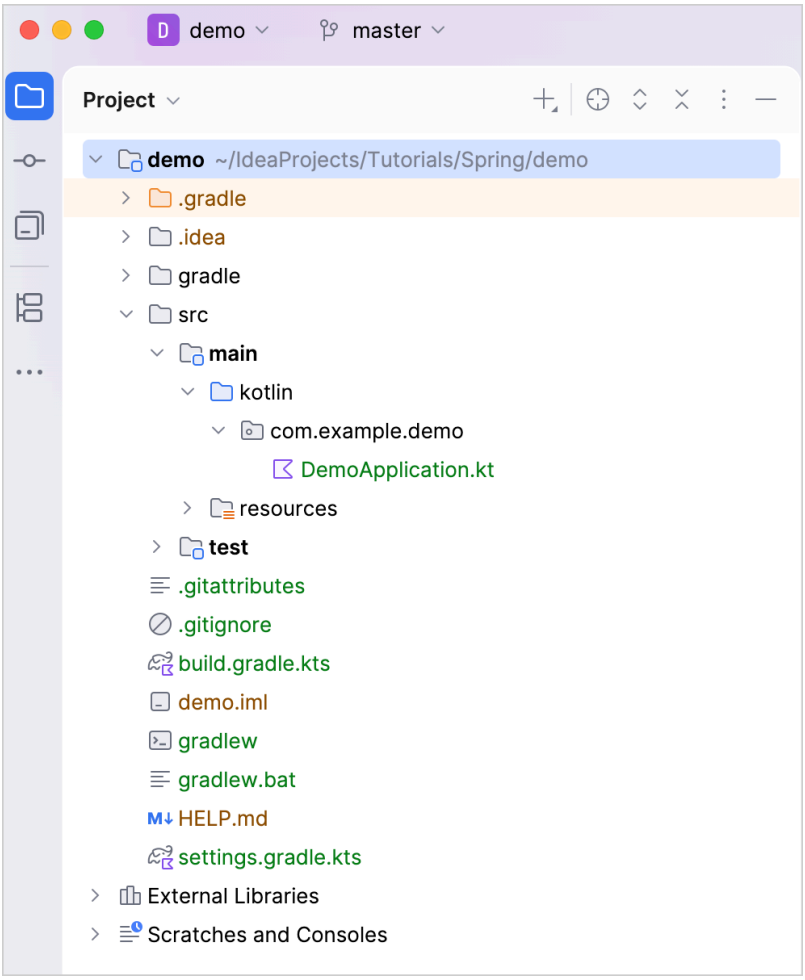


Set up Spring Boot project

6. Click **Create** to generate and set up the project.

> The IDE will generate and open a new project. It may take some time to download and import the project dependencies.

7. After this, you can observe the following structure in the **Project view**:



Set up Spring Boot project

The generated Gradle project corresponds to the Maven's standard directory layout:

- There are packages and classes under the `main/kotlin` folder that belong to the application.
- The entry point to the application is the `main()` method of the `DemoApplication.kt` file.

Explore the project Gradle build file

Open the `build.gradle.kts` file: it is the Gradle Kotlin build script, which contains a list of the dependencies required for the application.

The Gradle file is standard for Spring Boot, but it also contains necessary Kotlin dependencies, including the kotlin-spring Gradle plugin – `kotlin("plugin.spring")`.

Here is the full script with the explanation of all parts and dependencies:

```
// build.gradle.kts
plugins {
    kotlin("jvm") version "2.3.21" // The version of Kotlin to use
    kotlin("plugin.spring") version "2.3.21" // The Kotlin Spring plugin
    id("org.springframework.boot") version "4.1.1"
    id("io.spring.dependency-management") version "1.1.7"
}

group = "com.example"
version = "0.0.1-SNAPSHOT"

java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(17)
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.springframework.boot:spring-boot-h2console")
    implementation("org.springframework.boot:spring-boot-starter-data-jdbc")
    implementation("org.springframework.boot:spring-boot-starter-webmvc")
    implementation("org.jetbrains.kotlin:kotlin-reflect") // Kotlin reflection library, required for
working with Spring
    implementation("tools.jackson.module:jackson-module-kotlin") // Jackson extensions for Kotlin for
working with JSON
    runtimeOnly("com.h2database:h2")
    testImplementation("org.springframework.boot:spring-boot-starter-data-jdbc-test")
    testImplementation("org.springframework.boot:spring-boot-starter-webmvc-test")
    testImplementation("org.jetbrains.kotlin:kotlin-test-junit5")
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")
}

kotlin {
    compilerOptions {
        freeCompilerArgs.addAll("-Xjsr305=strict", "-Xannotation-default-target=param-property") // `-
Xjsr305=strict` enables the strict mode for JSR-305 annotations
    }
}

tasks.withType<Test> {
    useJUnitPlatform()
}
```

As you can see, there are a few Kotlin-related artifacts added to the Gradle build file:

1. In the `plugins` block, there are two Kotlin artifacts:
    - The `kotlin("jvm")` plugin defines the version of Kotlin to be used in the project.
    - The Kotlin Spring compiler plugin, `kotlin("plugin.spring")`, adds the `open` modifier to Kotlin classes to make them compatible with Spring Framework features
2. In the `dependencies` block, a few Kotlin-related modules are listed:
    - The `tools.jackson.module:jackson-module-kotlin` module adds support for serialization and deserialization of Kotlin classes and data classes.

- `org.jetbrains.kotlin:kotlin-reflect` is a Kotlin reflection library that enables full support for the reflection features.

3. After the dependencies section, you can see the `kotlin` plugin configuration block. This is where you can add extra arguments to the compiler to enable or disable various language features.

Learn more about the Kotlin compiler options in Compiler options in the Kotlin Gradle plugin.

Explore the generated Spring Boot application

Open the `DemoApplication.kt` file:

```
// DemoApplication.kt
package com.example.demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}
```

Declaring classes – class DemoApplication

Right after package declaration and import statements you can see the first class declaration, `class DemoApplication`.

In Kotlin, if a class doesn't include any members (properties or functions), you can omit the class body (`{}`) for good.

@SpringBootApplication annotation

`@SpringBootApplication annotation` is a convenience annotation in a Spring Boot application. It enables Spring Boot's auto-configuration, component scan, and be able to define an extra configuration on their "application class".

Program entry point – main()

The `main()` function is the entry point to the application.

It is declared as a top-level function outside the `DemoApplication` class. The `main()` function invokes the Spring's `runApplication(*args)` function to start the application with the Spring Framework.

Variable arguments – args: Array<String>

If you check the declaration of the `runApplication()` function, you will see that the parameter of the function is marked with `vararg` modifier: `vararg args: String`. This means that you can pass a variable number of String arguments to the function.

The spread operator – (*args)

The `args` is a parameter to the `main()` function declared as an array of Strings. Since there is an array of strings, and you want to pass its content to the function, use the spread operator (prefix the array with a star sign `*`).

Create a controller

The application is ready to run, but let's update its logic first.

In the Spring application, a controller is used to handle the web requests. In the same package, next to the `DemoApplication.kt` file, create the `MessageController.kt` file with the `MessageController` class as follows:

```
// MessageController.kt
package com.example.demo

import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.RestController

@RestController
class MessageController {
    @GetMapping("/")
    fun index(@RequestParam("name") name: String) = "Hello, $name!"
}
```

@RestController annotation

You need to tell Spring that `MessageController` is a REST Controller, so you should mark it with the `@RestController` annotation.

This annotation means this class will be picked up by the component scan because it's in the same package as our `DemoApplication` class.

@GetMapping annotation

`@GetMapping` marks the functions of the REST controller that implement the endpoints corresponding to HTTP GET calls:

```
@GetMapping("/")
      fun index(@RequestParam("name") name: String) = "Hello, $name!"
```

@RequestParam annotation

The function parameter `name` is marked with `@RequestParam` annotation. This annotation indicates that a method parameter should be bound to a web request parameter.

Hence, if you access the application at the root and supply a request parameter called "name", like `/?name=<your-value>`, the parameter value will be used as an argument for invoking the `index()` function.

Single-expression functions – index()

Since the `index()` function contains only one statement you can declare it as a single-expression function.

This means the curly braces can be omitted and the body is specified after the equals sign `=`.

Type inference for function return types

The `index()` function does not declare the return type explicitly. Instead, the compiler infers the return type by looking at the result of the statement on the right-hand side from the equals sign `=`.

The type of `Hello, $name!` expression is `String`, hence the return type of the function is also `String`.

String templates – $name

`Hello, $name!` expression is called a String template in Kotlin.

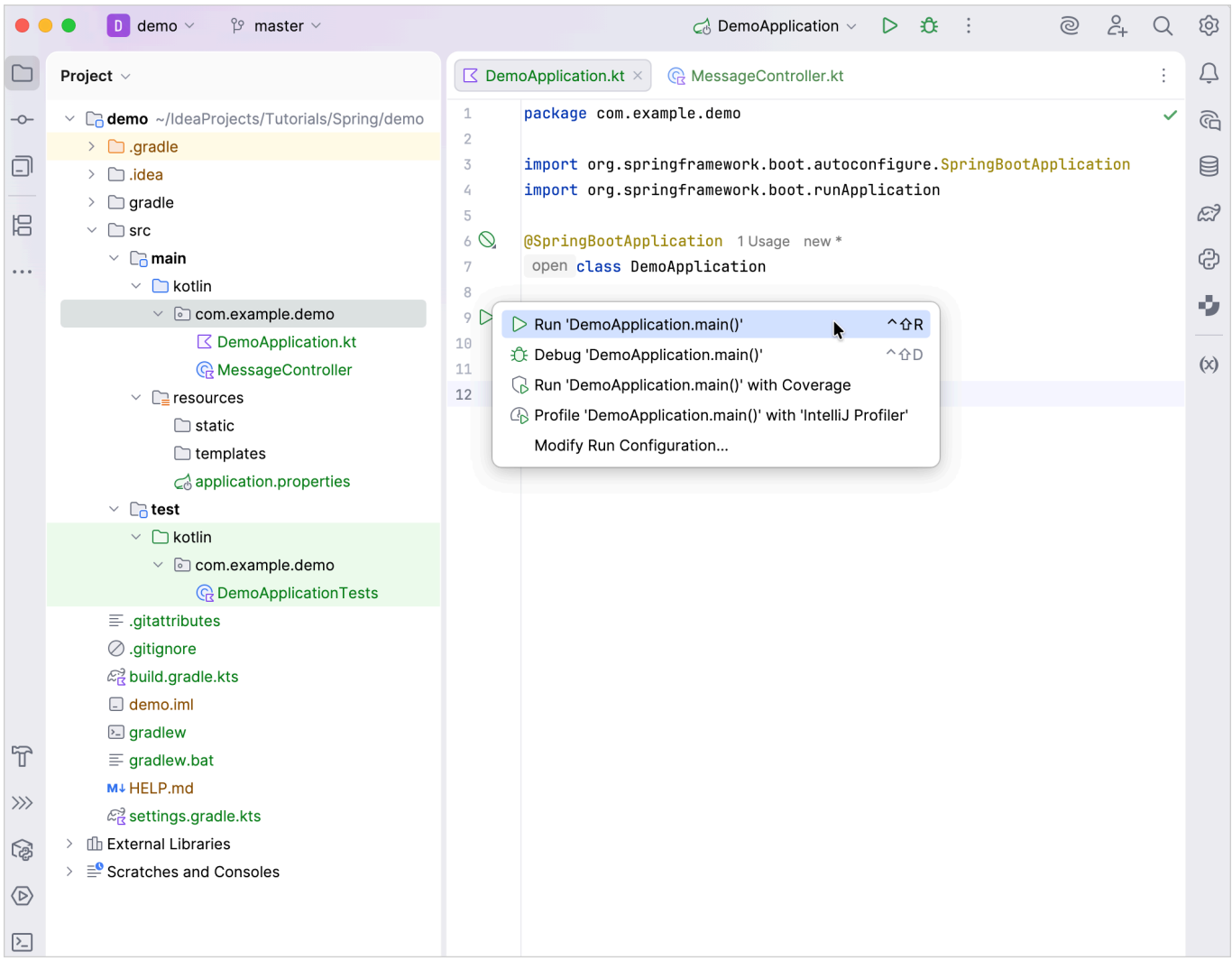
String templates are String literals that contain embedded expressions.

This is a convenient replacement for String concatenation operations.

Run the application

The Spring application is now ready to run:

1. In the `DemoApplication.kt` file, click the green **Run** icon in the gutter beside the `main()` method:



Run Spring Boot application

> You can also run the `./gradlew bootRun` command in the terminal.

This starts the local server on your computer.

2. Once the application starts, open the following URL:

```
http://localhost:8080?name=John
```

You should see "Hello, John!" printed as a response:



Spring Application response

Next step

In the next part of the tutorial, you'll learn about Kotlin data classes and how you can use them in your application.

- Next step

Add a data class to Spring Boot project

In this part of the tutorial, you'll add some more functionality to the application and discover more Kotlin language features, such as data classes. It requires changing the `MessageController` class to respond with a JSON document containing a collection of serialized objects.

Update your application

1. In the same package, next to the `DemoApplication.kt` file, create a `Message.kt` file.
2. In the `Message.kt` file, create a data class with two properties: `id` and `text`:

```
// Message.kt
package com.example.demo

data class Message(val id: String?, val text: String)
```

`Message` class will be used for data transfer: a list of serialized `Message` objects will make up the JSON document that the controller is going to respond to the browser request.

Data classes – data class Message
The main purpose of data classes in Kotlin is to hold data. Such classes are marked with the `data` keyword, and some standard functionality and some utility functions are often mechanically derivable from the class structure.

In this example, you declared `Message` as a data class as its main purpose is to store the data.

val and var properties

Properties in Kotlin classes can be declared either as:

- mutable, using the `var` keyword
- read-only, using the `val` keyword

The `Message` class declares two properties using `val` keyword, the `id` and `text`. The compiler will automatically generate the getters for both of these properties. It will not be possible to reassign the values of these properties after an instance of the `Message` class is created.

Nullable types – String?
Kotlin provides built-in support for nullable types. In Kotlin, the type system distinguishes between references that can hold `null` (nullable references) and those that cannot (non-nullable references).
For example, a regular variable of type `String` cannot hold `null`. To allow nulls, you can declare a variable as a nullable string by writing `String?`.

The `id` property of the `Message` class is declared as a nullable type this time. Hence, it is possible to create an instance of `Message` class by passing `null` as a value for `id`:

```
Message(null, "Hello!")
```

3. In the `MessageController.kt` file, instead of the `index()` function, create the `listMessages()` function returning a list of `Message` objects:

```
// MessageController.kt
package com.example.demo

import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController

@RestController
@RequestMapping("/")
class MessageController {
    @GetMapping
    fun listMessages() = listOf(
        Message("1", "Hello!"),
        Message("2", "Bonjour!"),
        Message("3", "Privet!"),
    )
}
```

Collections – listOf()
The Kotlin Standard Library provides implementations for basic collection types: sets, lists, and maps.
Each collection type can be read-only or mutable:

- A read-only collection comes with operations for accessing collection elements.
- A mutable collection comes also with write operations for adding, removing, and updating its elements.

The corresponding factory functions are also provided by the Kotlin Standard Library to create instances of such collections.

In this tutorial, you use the `listOf()` function to create a list of `Message` objects. This is the factory function to create a read-only list of objects: you can't add or remove elements from the list.
If it is required to perform write operations on the list, call the `mutableListOf()` function to create a mutable list instance.

Trailing comma

A trailing comma is a comma symbol after the last item of a series of elements:

```
Message("3", "Privet!"),
```

This is a convenient feature of Kotlin syntax and is entirely optional – your code will still work without them.

In the example above, creating a list of `Message` objects includes the trailing comma after the last `listOf()` function argument.

The response from `MessageController` will now be a JSON document containing a collection of `Message` objects.

> Any controller in the Spring application renders a JSON response by default if the Jackson library is on the classpath. As you specified the `spring-boot-starter-webmvc` dependency in the `build.gradle.kts` file, you received Jackson as a *transitive* dependency. Hence, the application responds with a JSON document if the endpoint returns a data structure that can be serialized to JSON.

Here is a complete code of the `DemoApplication.kt`, `MessageController.kt`, and `Message.kt` files:

```
// DemoApplication.kt
package com.example.demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}
```

```
// MessageController.kt
package com.example.demo

import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController

@RestController
@RequestMapping("/")
class MessageController {
    @GetMapping
    fun listMessages() = listOf(
        Message("1", "Hello!"),
        Message("2", "Bonjour!"),
        Message("3", "Privet!"),
    )
}
```

```
// Message.kt
package com.example.demo

data class Message(val id: String?, val text: String)
```

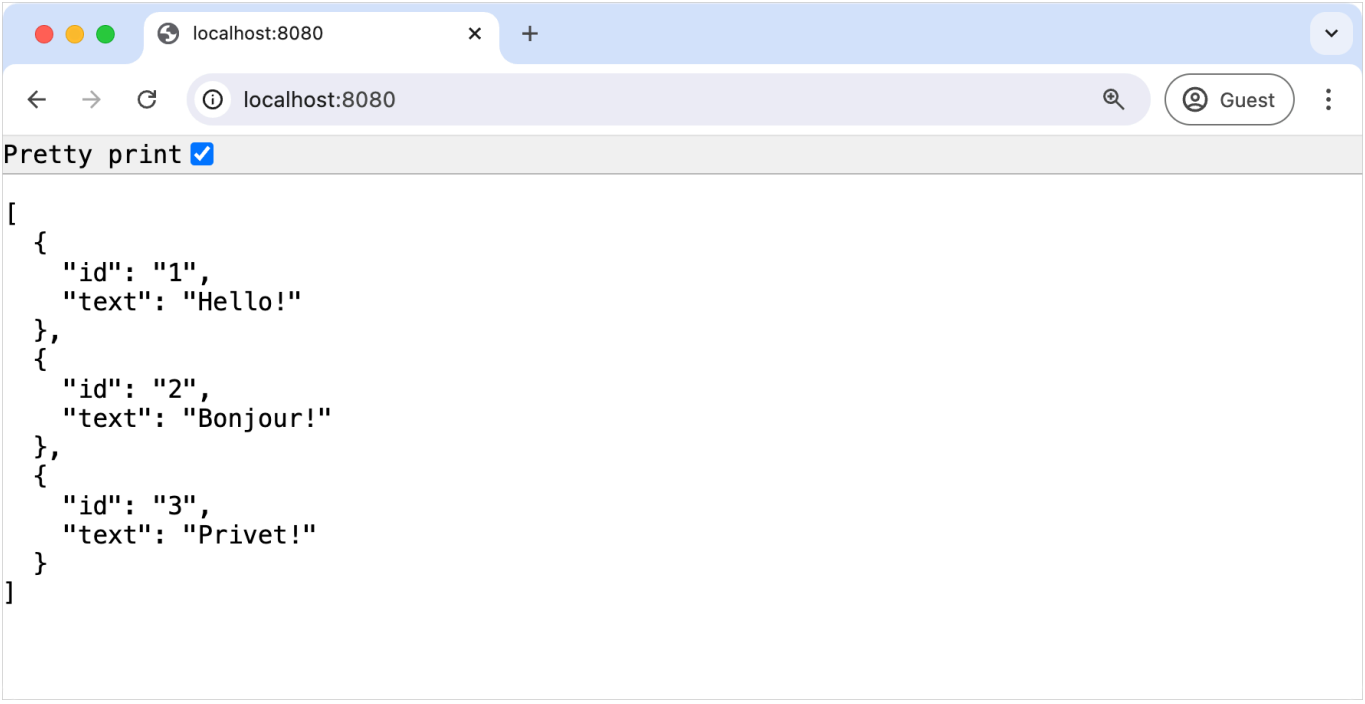
Run the application

The Spring application is ready to run:

1. Run the application again.
2. Once the application starts, open the following URL:

   ```
   http://localhost:8080
   ```

   You will see a page with a collection of messages in JSON format:



Run the application

Next step

In the next part of the tutorial, you'll add and configure a database to your project and make HTTP requests.

Previous step

Next step

Add database support for Spring Boot project

In this part of the tutorial, you'll add and configure a database to your project using *Java Database Connectivity* (JDBC). In JVM applications, you use JDBC to interact with databases. For convenience, the Spring Framework provides the `JdbcTemplate` class that simplifies the use of JDBC and helps to avoid common errors.

Add database support

The common practice in Spring Framework-based applications is to implement the database access logic within the so-called *service* layer – this is where business logic lives. In Spring, you should mark classes with the

`@Service` annotation to imply that the class belongs to the service layer of the application. In this application, you will create the `MessageService` class for this purpose.

In the same package, create the `MessageService.kt` file and the `MessageService` class as follows:

```
// MessageService.kt
package com.example.demo

import org.springframework.stereotype.Service
import org.springframework.jdbc.core.JdbcTemplate

@Service
class MessageService(private val db: JdbcTemplate) {
    fun findMessages(): List<Message> = db.query("select * from messages") { response, _ ->
        Message(response.getString("id"), response.getString("text"))
    }

    fun save(message: Message): Message {
        db.update(
            "insert into messages values ( ?, ? )",
            message.id, message.text
        )
        return message
    }
}
```

Constructor argument and dependency injection – (private val db: JdbcTemplate)

A class in Kotlin has a primary constructor. It can also have one or more secondary constructors. The primary constructor is a part of the class header, and it goes after the class name and optional type parameters. In our case, the constructor is `(val db: JdbcTemplate)`.

`val db: JdbcTemplate` is the constructor's argument:

```
@Service
      class MessageService(private val db: JdbcTemplate)
```

Trailing lambda and SAM conversion

The `findMessages()` function calls the `query()` function of the `JdbcTemplate` class. The `query()` function takes two arguments: an SQL query as a String instance, and a callback that will map one object per row:

```
db.query("...", RowMapper { ... } )
```

The `RowMapper` interface declares only one method, so it is possible to implement it via lambda expression by omitting the name of the interface. The Kotlin compiler knows the interface that the lambda expression needs to be converted to because you use it as a parameter for the function call. This is known as SAM conversion in Kotlin:

```
db.query("...", { ... } )
```

After the SAM conversion, the query function ends up with two arguments: a String at the first position, and a lambda expression at the last position. According to the Kotlin convention, if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses. Such syntax is also known as trailing lambda:

```
db.query("...") { ... }
```

Underscore for unused lambda argument

For a lambda with multiple parameters, you can use the underscore `_` character to replace the names of the parameters you don't use.

Hence, the final syntax for the query function call looks like this:

```
db.query("select * from messages") { response, _ ->
          Message(response.getString("id"), response.getString("text"))
      }
```

Update the MessageController class

Update `MessageController.kt` to use the new `MessageService` class:

```
// MessageController.kt
package com.example.demo

import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PostMapping
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController
import java.net.URI

@RestController
@RequestMapping("/")
class MessageController(private val service: MessageService) {
    @GetMapping
    fun listMessages() = service.findMessages()

    @PostMapping
    fun post(@RequestBody message: Message): ResponseEntity<Message> {
        val savedMessage = service.save(message)
        return ResponseEntity.created(URI("/${savedMessage.id}")).body(savedMessage)
    }
}
```

@PostMapping annotation

The method responsible for handling HTTP POST requests needs to be annotated with `@PostMapping` annotation. To be able to convert the JSON sent as HTTP Body content into an object, you need to use the `@RequestBody` annotation for the method argument. Thanks to having the Jackson library in the classpath of the application, the conversion happens automatically.

ResponseEntity

`ResponseEntity` represents the whole HTTP response: status code, headers, and body.

Using the `created()` method you configure the response status code (201) and set the location header indicating the context path for the created resource.

Update the MessageService class

The `id` for `Message` class was declared as a nullable String:

```
data class Message(val id: String?, val text: String)
```

It would not be correct to store the `null` as an `id` value in the database though: you need to handle this situation gracefully.

Update your code of the `MessageService.kt` file to generate a new value when the `id` is `null` while storing the messages in the database:

```
// MessageService.kt
package com.example.demo

import org.springframework.stereotype.Service
import org.springframework.jdbc.core.JdbcTemplate
import java.util.UUID

@Service
class MessageService(private val db: JdbcTemplate) {
    fun findMessages(): List<Message> = db.query("select * from messages") { response, _ ->
        Message(response.getString("id"), response.getString("text"))
    }

    fun save(message: Message): Message {
        val id = message.id ?: UUID.randomUUID().toString() // Generate new id if it is null
        db.update(
            "insert into messages values ( ?, ? )",
            id, message.text
        )
        return message.copy(id = id) // Return a copy of the message with the new id
    }
}
```

Elvis operator – ?:

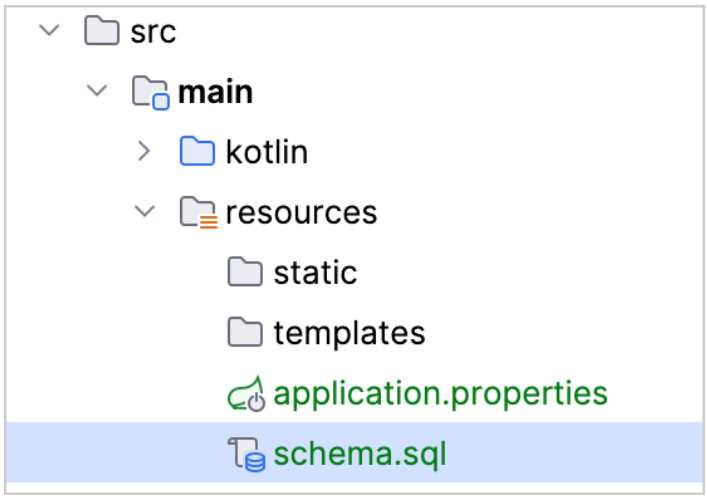
The code `message.id ?: UUID.randomUUID().toString()` uses the Elvis operator (if-not-null-else shorthand) `?:`. If the expression to the left of `?:` is not `null`, the Elvis operator returns it; otherwise, it returns the expression to the right. Note that the expression on the right-hand side is evaluated only if the left-hand side is `null`.

The application code is ready to work with the database. It is now required to configure the data source.

Configure the database

Configure the database in the application:

1. Create the `schema.sql` file in the `src/main/resources` directory. It will store the database object definitions:



Create database schema

2. Update the `src/main/resources/schema.sql` file with the following code:

```
-- schema.sql
CREATE TABLE IF NOT EXISTS messages (
id       VARCHAR(60)  PRIMARY KEY,
text     VARCHAR      NOT NULL
);
```

It creates the `messages` table with two columns: `id` and `text`. The table structure matches the structure of the `Message` class.

3. Open the `application.properties` file located in the `src/main/resources` folder and add the following application properties:

```
spring.application.name=demo
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:file:./data/testdb
spring.datasource.username=name
spring.datasource.password=password
spring.sql.init.schema-locations=classpath:schema.sql
spring.sql.init.mode=always
```

These settings enable the database for the Spring Boot application.
See the full list of common application properties in the Spring documentation.

Add messages to database via HTTP request

You should use an HTTP client to work with previously created endpoints. In IntelliJ IDEA, use the embedded HTTP client:

1. Run the application. Once the application is up and running, you can execute POST requests to store messages in the database.
2. Create the `requests.http` file in the project root folder and add the following HTTP requests:

```
### Post "Hello!"
POST http://localhost:8080/
Content-Type: application/json

{
  "text": "Hello!"
}

### Post "Bonjour!"

POST http://localhost:8080/
Content-Type: application/json

{
  "text": "Bonjour!"
}

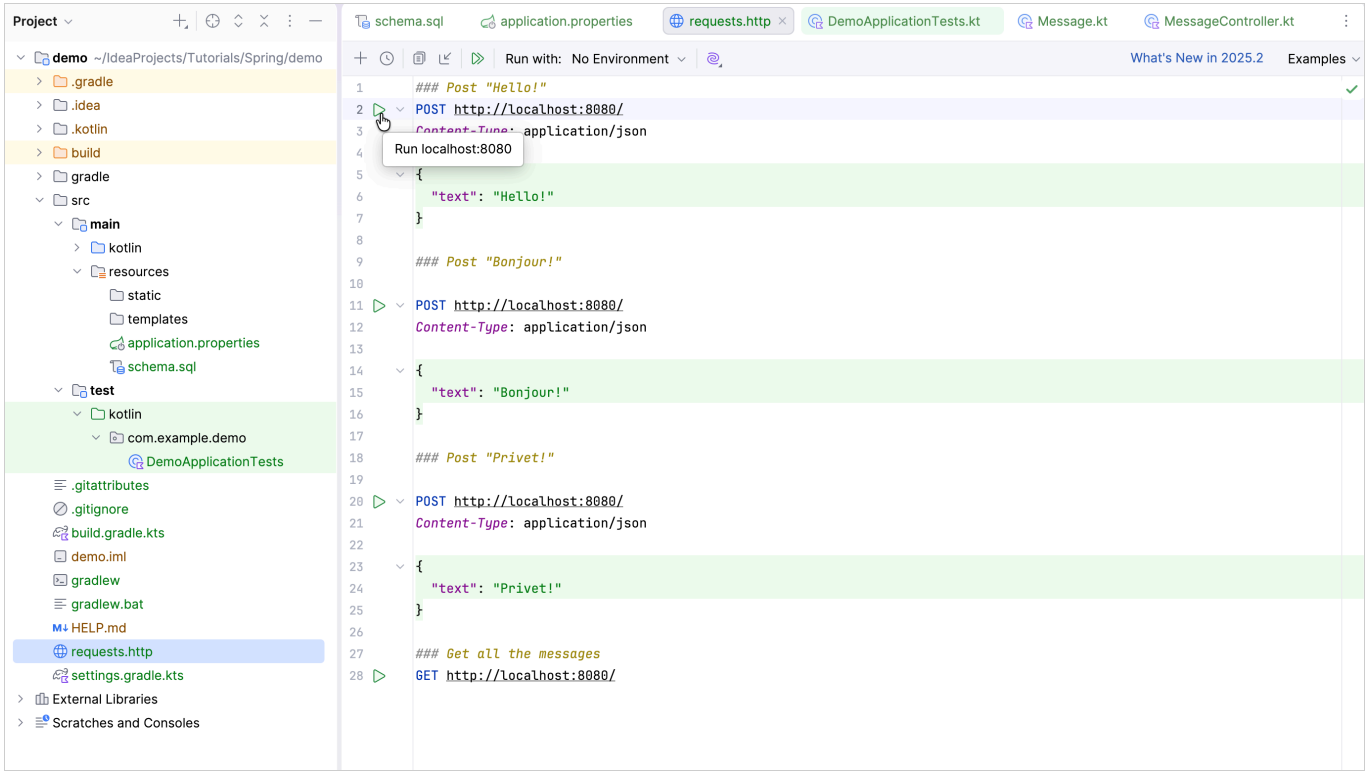
### Post "Privet!"

POST http://localhost:8080/
Content-Type: application/json

{
  "text": "Privet!"
}

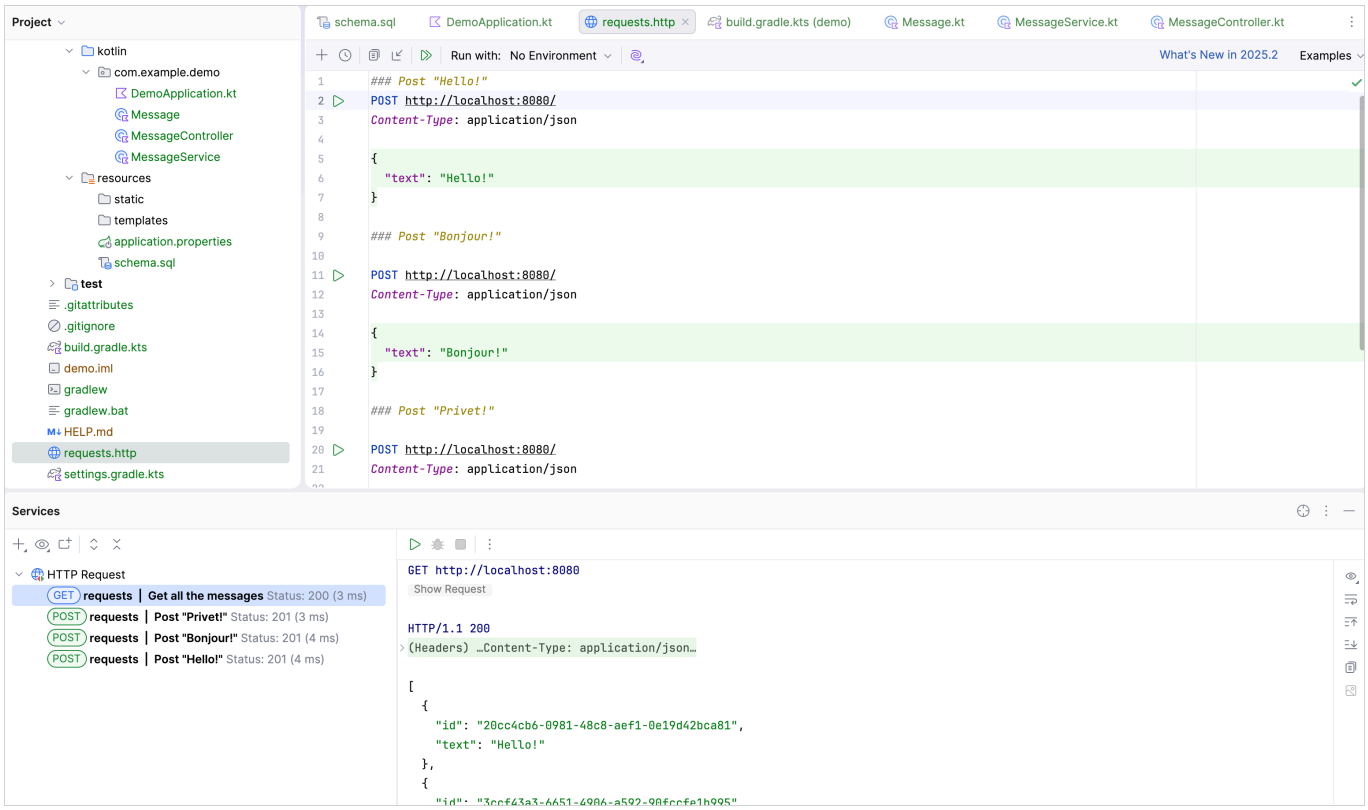
### Get all the messages
GET http://localhost:8080/
```

3. Execute all POST requests. Use the green **Run** icon in the gutter next to the request declaration. These requests write the text messages to the database:



Execute POST request

4. Execute the GET request and see the result in the **Run** tool window:



Execute GET requests

Alternative way to execute requests

You can also use any other HTTP client or the cURL command-line tool. For example, run the following commands in the terminal to get the same result:

```
curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json" -d "{ \"text\": \"Hello!\" }"

curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json" -d "{ \"text\": \"Bonjour!\" }"

curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json" -d "{ \"text\": \"Privet!\" }"

curl -X GET --location "http://localhost:8080"
```

Retrieve messages by id

Extend the functionality of the application to retrieve the individual messages by id.

1. In the `MessageService` class, add the new function `findMessageById(id: String)` to retrieve the individual messages by id:

```kotlin
// MessageService.kt
package com.example.demo

import org.springframework.stereotype.Service
import org.springframework.jdbc.core.JdbcTemplate
import org.springframework.jdbc.core.query
import java.util.*

@Service
class MessageService(private val db: JdbcTemplate) {
    fun findMessages(): List<Message> = db.query("select * from messages") { response, _ ->
        Message(response.getString("id"), response.getString("text"))
    }

    fun findMessageById(id: String): Message? = db.query("select * from messages where id = ?", id)
{ response, _ ->
        Message(response.getString("id"), response.getString("text"))
    }.singleOrNull()

    fun save(message: Message): Message {
        val id = message.id ?: UUID.randomUUID().toString() // Generate new id if it is null
        db.update(
            "insert into messages values ( ?, ? )",
            id, message.text
        )
        return message.copy(id = id) // Return a copy of the message with the new id
    }
}
```

vararg argument position in the parameter list
The `query()` function takes three arguments:

- SQL query string that requires a parameter to run
- `id`, which is a parameter of type String
- `RowMapper` instance, which implemented by a lambda expression

The second parameter for the `query()` function is declared as a variable argument (`vararg`). In Kotlin, the position of the variable arguments parameter is not required to be the last in the parameters list.

singleOrNull() function
The `singleOrNull()` function returns a single element, or `null` if the array is empty or has more than one element with the same value.

> The `.query()` function used to fetch the message by its id is a Kotlin extension function provided by the Spring Framework. It requires an additional import `import org.springframework.jdbc.core.query` as demonstrated in the code above.

2. Add the new `index(...)` function with the `id` parameter to the `MessageController` class:

```kotlin
// MessageController.kt
package com.example.demo

import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.PostMapping
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController
import java.net.URI

@RestController
@RequestMapping("/")
class MessageController(private val service: MessageService) {
    @GetMapping
    fun listMessages() = ResponseEntity.ok(service.findMessages())

    @PostMapping
    fun post(@RequestBody message: Message): ResponseEntity<Message> {
        val savedMessage = service.save(message)
        return ResponseEntity.created(URI("/${savedMessage.id}")).body(savedMessage)
    }

    @GetMapping("/{id}")
    fun getMessage(@PathVariable id: String): ResponseEntity<Message> =
        service.findMessageById(id).toResponseEntity()

    private fun Message?.toResponseEntity(): ResponseEntity<Message> =
        // If the message is null (not found), set response code to 404
        this?.let { ResponseEntity.ok(it) } ?: ResponseEntity.notFound().build()
}
```

Retrieving a value from the context path

The message `id` is retrieved from the context path by the Spring Framework as you annotated the new function by `@GetMapping("/{id}")`. By annotating the function argument with `@PathVariable`, you tell the framework to use the retrieved value as a function argument. The new function makes a call to `MessageService` to retrieve the individual message by its id.

Extension function with nullable receiver

Extensions can be defined with a nullable receiver type. If the receiver is `null`, then `this` is also `null`. So when defining an extension with a nullable receiver type, it is recommended performing a `this == null` check inside the function body.

You can also use the null-safe invocation operator (`?.`) to perform the null check as in the `toResponseEntity()` function above:

```kotlin
this?.let { ResponseEntity.ok(it) }
```

ResponseEntity

`ResponseEntity` represents the HTTP response, including the status code, headers, and body. It is a generic wrapper that allows you to send customized HTTP responses back to the client with more control over the content.

Here is a complete code of the application:

```kotlin
// DemoApplication.kt
package com.example.demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}
```

```kotlin
// Message.kt
package com.example.demo

data class Message(val id: String?, val text: String)
```

```kotlin
// MessageService.kt
package com.example.demo

import org.springframework.stereotype.Service
import org.springframework.jdbc.core.JdbcTemplate
import org.springframework.jdbc.core.query
import java.util.*

@Service
class MessageService(private val db: JdbcTemplate) {
    fun findMessages(): List<Message> = db.query("select * from messages") { response, _ ->
        Message(response.getString("id"), response.getString("text"))
    }

    fun findMessageById(id: String): Message? = db.query("select * from messages where id = ?", id) {
response, _ ->
        Message(response.getString("id"), response.getString("text"))
    }.singleOrNull()

    fun save(message: Message): Message {
        val id = message.id ?: UUID.randomUUID().toString()
        db.update(
            "insert into messages values ( ?, ? )",
            id, message.text
        )
        return message.copy(id = id)
    }
}
```

```kotlin
// MessageController.kt
package com.example.demo

import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.PostMapping
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController
import java.net.URI

@RestController
@RequestMapping("/")
class MessageController(private val service: MessageService) {
    @GetMapping
    fun listMessages() = ResponseEntity.ok(service.findMessages())

    @PostMapping
    fun post(@RequestBody message: Message): ResponseEntity<Message> {
        val savedMessage = service.save(message)
        return ResponseEntity.created(URI("/${savedMessage.id}")).body(savedMessage)
    }

    @GetMapping("/{id}")
    fun getMessage(@PathVariable id: String): ResponseEntity<Message> =
        service.findMessageById(id).toResponseEntity()

    private fun Message?.toResponseEntity(): ResponseEntity<Message> =
        this?.let { ResponseEntity.ok(it) } ?: ResponseEntity.notFound().build()
}
```

Run the application

The Spring application is ready to run:

1. Run the application again.
2. Open the `requests.http` file and add the new GET request:

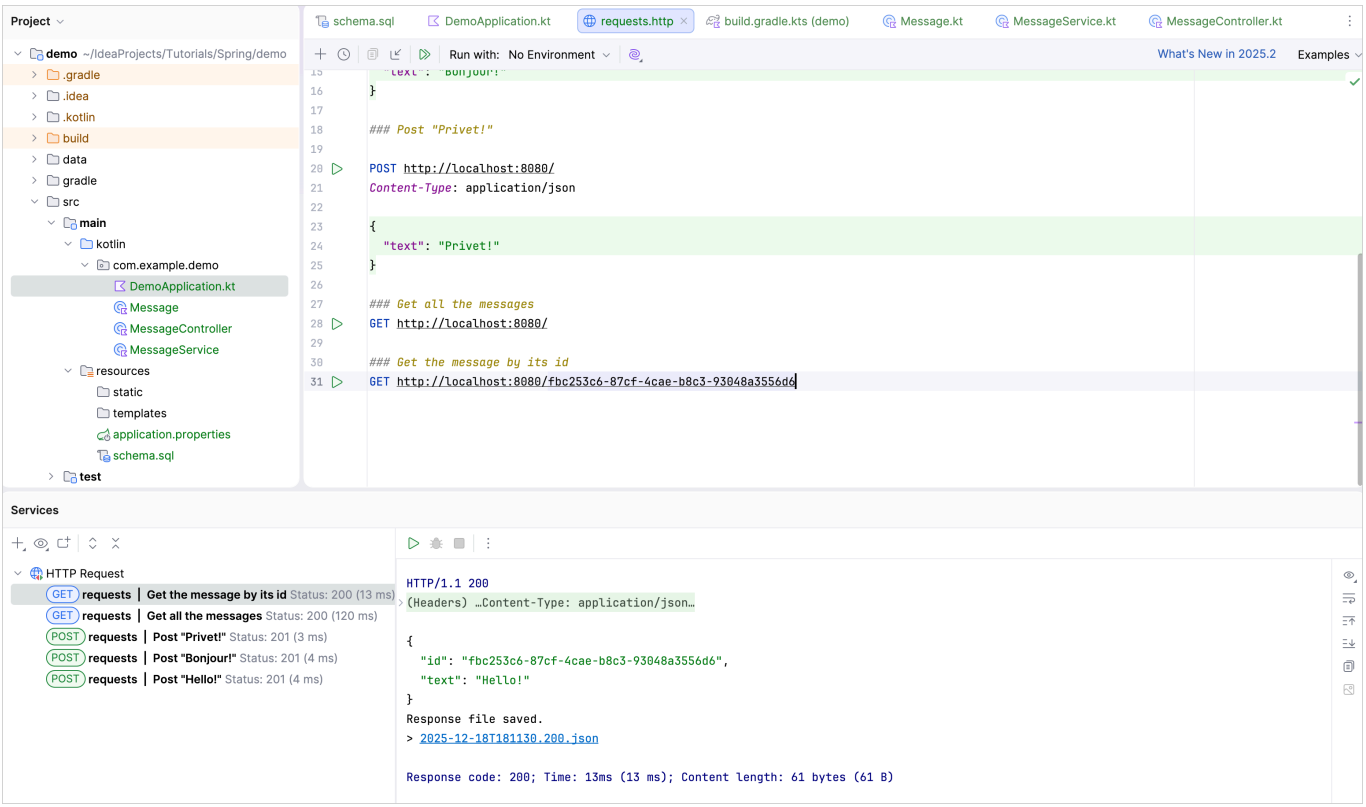
```
### Get the message by its id
GET http://localhost:8080/id
```

3. Execute the GET request to retrieve all the messages from the database.
4. In the **Run** tool window, copy one of the ids and add it to the request, like this:

```
### Get the message by its id
GET http://localhost:8080/f910aa7e-11ee-4215-93ed-1aeeac822707
```

> Put your message id instead of the mentioned above.

5. Execute the GET request and see the result in the **Run** tool window:



Retrieve message by its id

Next step

The final step shows you how to use a more popular connection to the database using Spring Data.

Previous step

Next step

Use Spring Data CrudRepository for database access

In this part, you will migrate the service layer to use the Spring Data `CrudRepository` instead of `JdbcTemplate` for database access. *CrudRepository* is a Spring Data interface for generic CRUD operations on a repository of a specific type. It provides several methods out of the box for interacting with a database.

Update your application

First, you need to adjust the `Message` class for work with the `CrudRepository` API:

1. Add the `@Table` annotation to the `Message` class to declare mapping to a database table.
   Add the `@Id` annotation before the `id` field.

> These annotations also require additional imports.

```kotlin
// Message.kt
package com.example.demo

import org.springframework.data.annotation.Id
import org.springframework.data.relational.core.mapping.Table

@Table("MESSAGES")
data class Message(@Id val id: String?, val text: String)
```

In addition, to make the use of the `Message` class more idiomatic, you can set the default value for `id` property to null and flip the order of the data class properties:

```kotlin
@Table("MESSAGES")
data class Message(val text: String, @Id val id: String? = null)
```

Now if you need to create a new instance of the `Message` class, you can only specify the `text` property as a parameter:

```kotlin
val message = Message("Hello") // id is null
```

2. Declare an interface for the `CrudRepository` that will work with the `Message` data class. Create the `MessageRepository.kt` file and add the following code to it:

```kotlin
// MessageRepository.kt
package com.example.demo

import org.springframework.data.repository.CrudRepository

interface MessageRepository : CrudRepository<Message, String>
```

3. Update the `MessageService` class. It will now use the `MessageRepository` instead of executing SQL queries:

```kotlin
// MessageService.kt
package com.example.demo

import org.springframework.data.repository.findByIdOrNull
import org.springframework.stereotype.Service

@Service
class MessageService(private val db: MessageRepository) {
    fun findMessages(): List<Message> = db.findAll().toList()

    fun findMessageById(id: String): Message? = db.findByIdOrNull(id)

    fun save(message: Message): Message = db.save(message)
}
```

Extension functions
The `findByIdOrNull()` function is an extension function for `CrudRepository` interface in Spring Data JDBC.

CrudRepository save() function
This function works with an assumption that the new object doesn't have an id in the database. Hence, the id should be null for insertion.

If the id isn't null, `CrudRepository` assumes that the object already exists in the database and this is an update operation as opposed to an insert operation. After the insert operation, the `id` will be generated by the data store and assigned back to the `Message` instance.

4. Update the messages table definition to generate the ids for the inserted objects. Since `id` is a string, you can use the `RANDOM_UUID()` function to generate the id value by default:

```
-- schema.sql
CREATE TABLE IF NOT EXISTS messages (
    id      VARCHAR(60)  DEFAULT RANDOM_UUID() PRIMARY KEY,
    text    VARCHAR      NOT NULL
);
```

5. Update the name of the database in the `application.properties` file located in the `src/main/resources` folder:

```
spring.application.name=demo
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:file:./data/testdb2
spring.datasource.username=name
spring.datasource.password=password
spring.sql.init.schema-locations=classpath:schema.sql
spring.sql.init.mode=always
```

Here is the complete code of the application:

```
// DemoApplication.kt
package com.example.demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}
```

```
// Message.kt
package com.example.demo

import org.springframework.data.annotation.Id
import org.springframework.data.relational.core.mapping.Table

@Table("MESSAGES")
data class Message(val text: String, @Id val id: String? = null)
```

```
// MessageRepository.kt
package com.example.demo

import org.springframework.data.repository.CrudRepository

interface MessageRepository : CrudRepository<Message, String>
```

```kotlin
// MessageService.kt
package com.example.demo

import org.springframework.data.repository.findByIdOrNull
import org.springframework.stereotype.Service

@Service
class MessageService(private val db: MessageRepository) {
    fun findMessages(): List<Message> = db.findAll().toList()

    fun findMessageById(id: String): Message? = db.findByIdOrNull(id)

    fun save(message: Message): Message = db.save(message)
}
```

```kotlin
// MessageController.kt
package com.example.demo

import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.PostMapping
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController
import java.net.URI

@RestController
@RequestMapping("/")
class MessageController(private val service: MessageService) {
    @GetMapping
    fun listMessages() = ResponseEntity.ok(service.findMessages())

    @PostMapping
    fun post(@RequestBody message: Message): ResponseEntity<Message> {
        val savedMessage = service.save(message)
        return ResponseEntity.created(URI("/${savedMessage.id}")).body(savedMessage)
    }

    @GetMapping("/{id}")
    fun getMessage(@PathVariable id: String): ResponseEntity<Message> =
        service.findMessageById(id).toResponseEntity()

    private fun Message?.toResponseEntity(): ResponseEntity<Message> =
        // If the message is null (not found), set response code to 404
        this?.let { ResponseEntity.ok(it) } ?: ResponseEntity.notFound().build()
}
```

Run the application

Congratulations! The application is ready to run again. After replacing `JdbcTemplate` with `CrudRepository`, the functionality remains the same, so the application works just as before.

You can now run the POST and GET HTTP requests from the `requests.http` file and get the same results.

- Previous step

What's next

Get your personal language map to help you navigate Kotlin features and track your progress in studying the language:

Get the Kotlin language map

- Check out the Spring Framework documentation.
- Create a simple web application with protected resources in the Securing a web application tutorial.
- Complete the Building web applications with Spring Boot and Kotlin tutorial.

Create a task manager app with Spring Boot and Claude

In this tutorial, you'll learn how to use Claude to create a Kotlin app to manage tasks. The tutorial uses Spring Boot to manage the backend infrastructure, while Claude plans and develops the application.

If you prefer to create an app without the help of AI, you can follow our Create a web app with Kotlin and Spring Boot tutorial.

As with any AI-powered tool, Claude can make mistakes. Review Claude's changes carefully and only use it with code you trust. For more information on Claude's security policy, see Claude Code Docs.

Set up the environment

This tutorial uses Claude through the JetBrains AI Assistant, but you can also work through the tutorial steps using Claude Code in the terminal.

1. Download and install the latest version of IntelliJ IDEA.
2. Install the JetBrains AI Assistant.
3. Activate Claude Agent in one of the following ways:
   - Using a JetBrains AI subscription
   - Using an API key
   - Using Anthropic Console

Create a project

You can also create a Spring Boot project using Spring's web-based project generator.

Create a new Spring Boot project in IntelliJ IDEA:

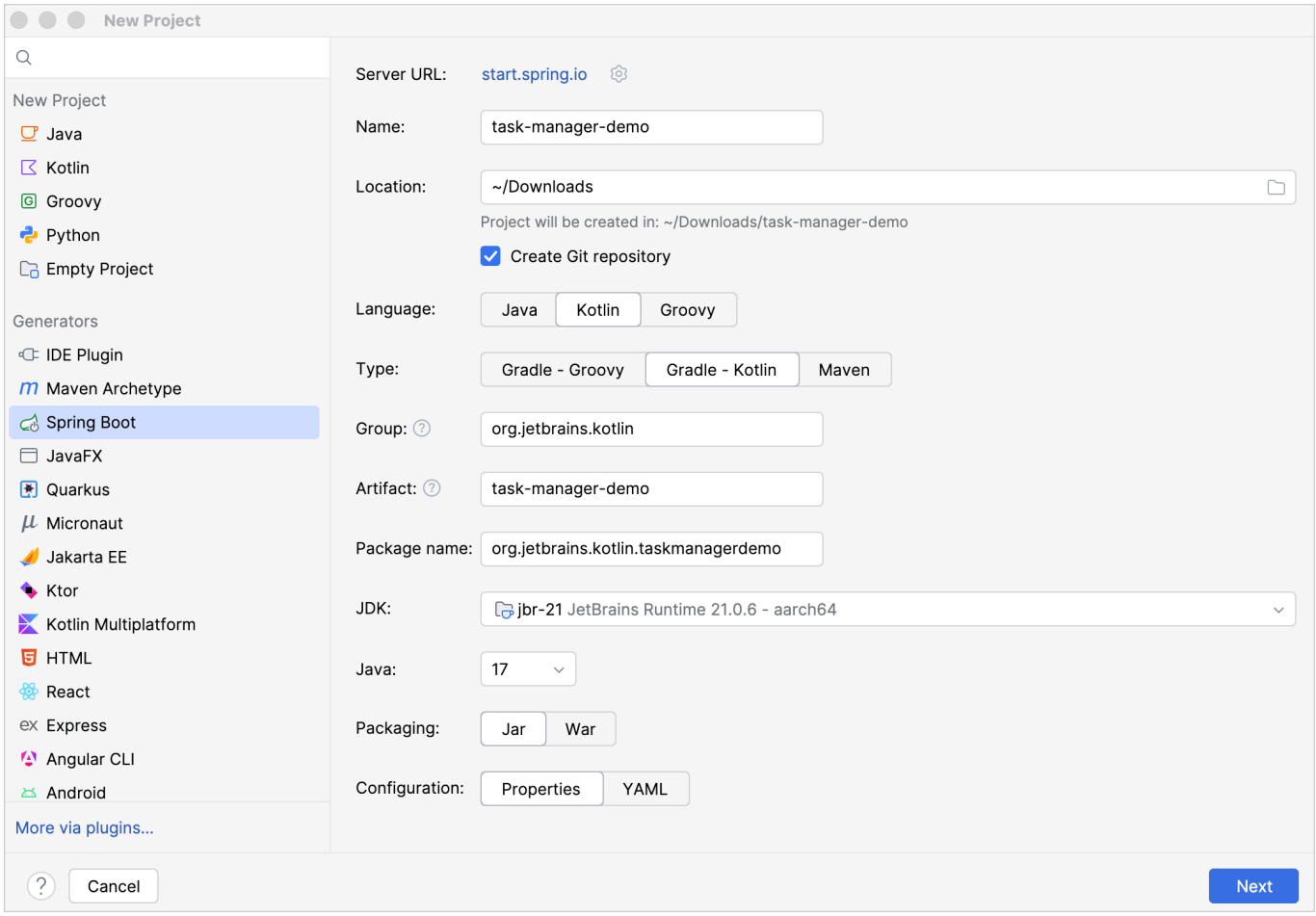
1. In IntelliJ IDEA, select **File** | **New** | **Project**.
2. In the panel on the left, select **New Project** | **Spring Boot**.
3. Specify the following fields and options in the **New Project** window:

- **Name**: task-manager-demo
- **Language**: Kotlin
- **Type**: Gradle - Kotlin

> This option specifies the build system and the DSL.

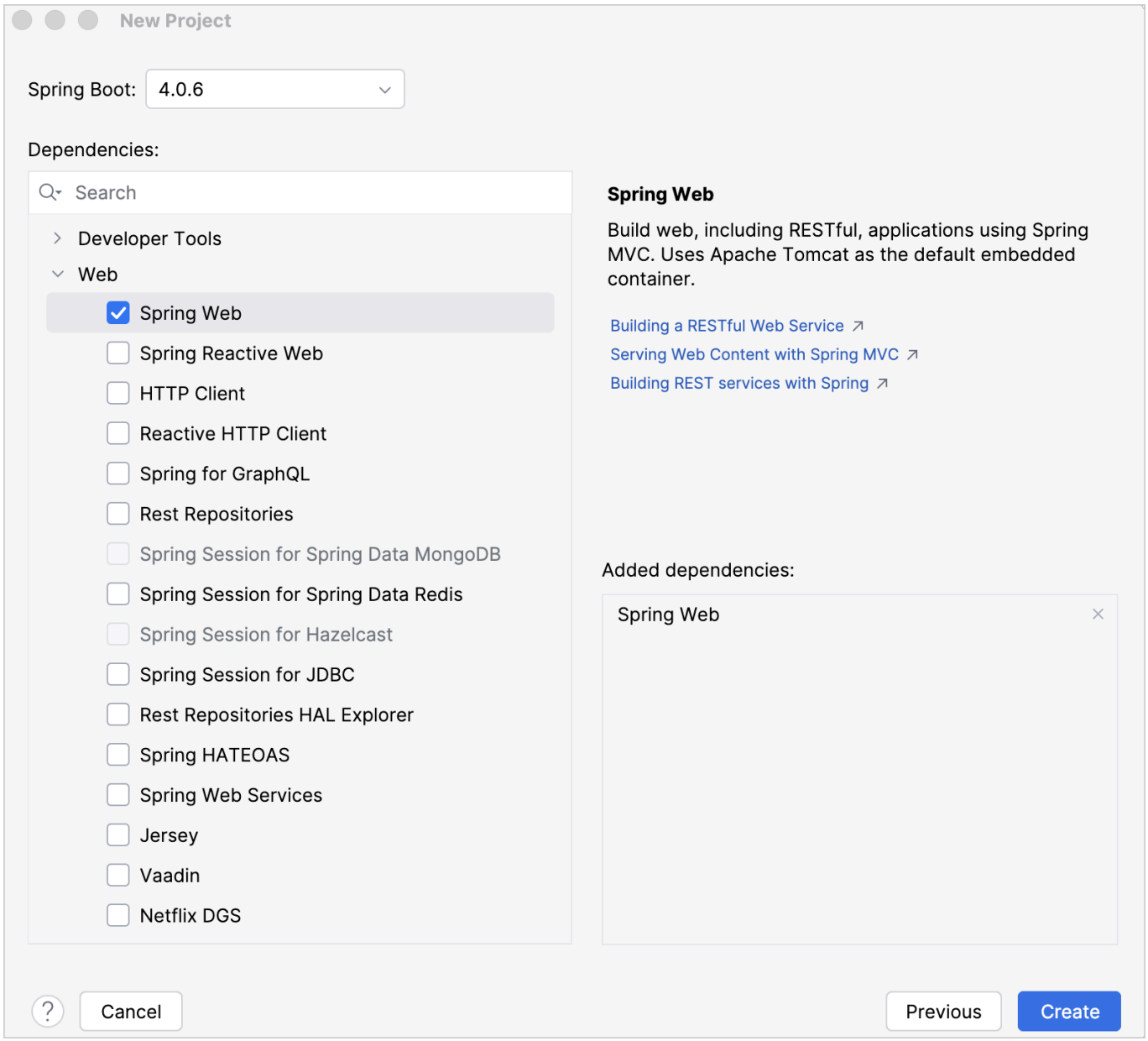
- **Package name**: org.jetbrains.kotlin.taskmanagerdemo
- **JDK**: jbr-21
- **Java**: 17

> If you don't have these Java and JDK versions installed, you can download them from the dropdown lists.



Create Spring Boot project

4. Make sure that you specified all the fields, then click **Next**.
5. Select the latest stable Spring Boot version in the **Spring Boot** field.
6. Select the **Web | Spring Web** dependency.



Set up Spring Boot project

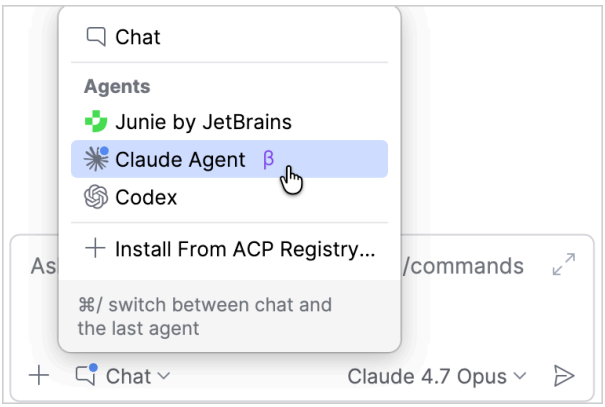
7. Click **Create** to generate and set up the project.

   The IDE generates and opens the new project. It may take some time to download and import the project dependencies.

Create a development plan

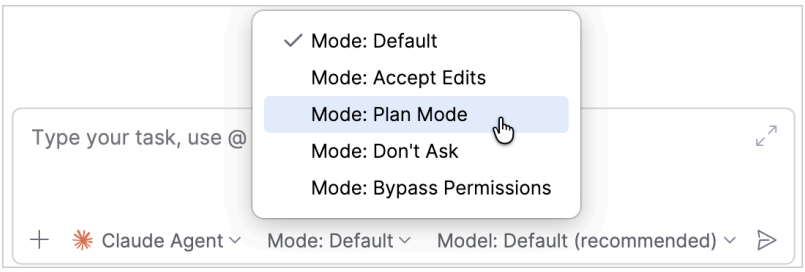
In your project:

1. Open the **AI Chat** tool window. By default, the **Chat** mode is selected. Select **Claude Agent**.



Select Claude Agent

2. Click **Mode: Default** and select **Mode: Plan Mode**. Claude Agent is now ready to plan without executing actions.



Select Plan Mode

> For more information about the different operation modes, see Select operation mode.

3. Write a prompt asking Claude to create a task manager app. Share some details on what you think it should include. For example:

```
I'd like to create a task manager application for managing tasks, such as a grocery list.
It should have a basic UI and include categories, due dates, priorities, and status tracking.

Use VCS while working. Work step by step and create commits at each stage so I can review the
changes afterward.
```

> For guidance on how to design your prompt, see Claude Code best practices.

Claude explores the existing project structure and suggests a plan.

4. Review the plan carefully before proceeding. If you want to make some modifications, select **No, keep planning** and share your follow-up comments.
5. When you're ready to proceed, select the **Yes ...** option that fits how much control you want over Claude's changes.



Ready to code

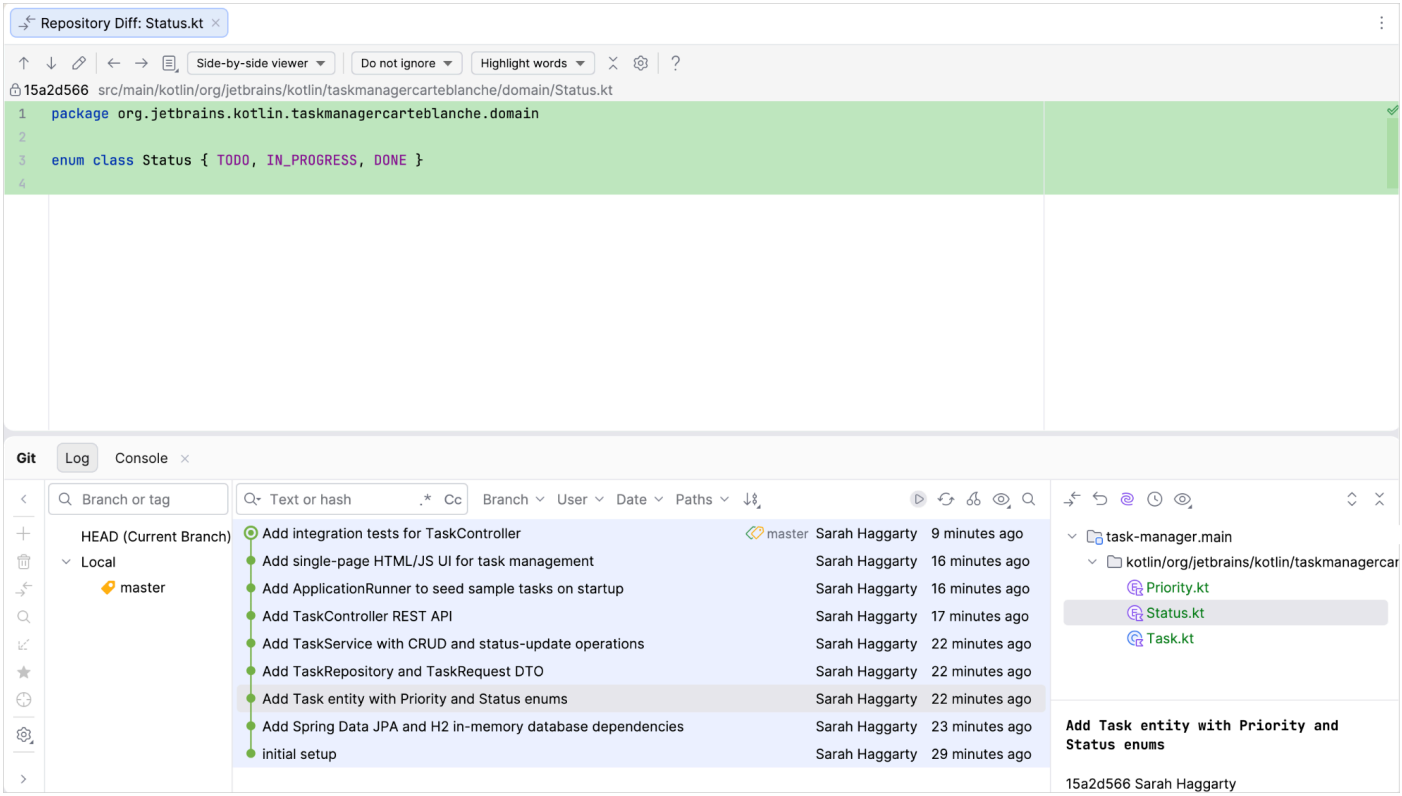
> For more information on the different options, see Claude Code permission modes.

6. Claude exits **Plan Mode** and begins working. Wait until the work is complete.

Review the commits

Before running the app, review the generated changes carefully:

1. Open the **Git** tool window to see the list of commits.
2. Select a commit and double-click each modified file to review the diff in IntelliJ IDEA's side-by-side viewer.



Side-by-side viewer

Run the app

Once you are happy with the changes, run the app:

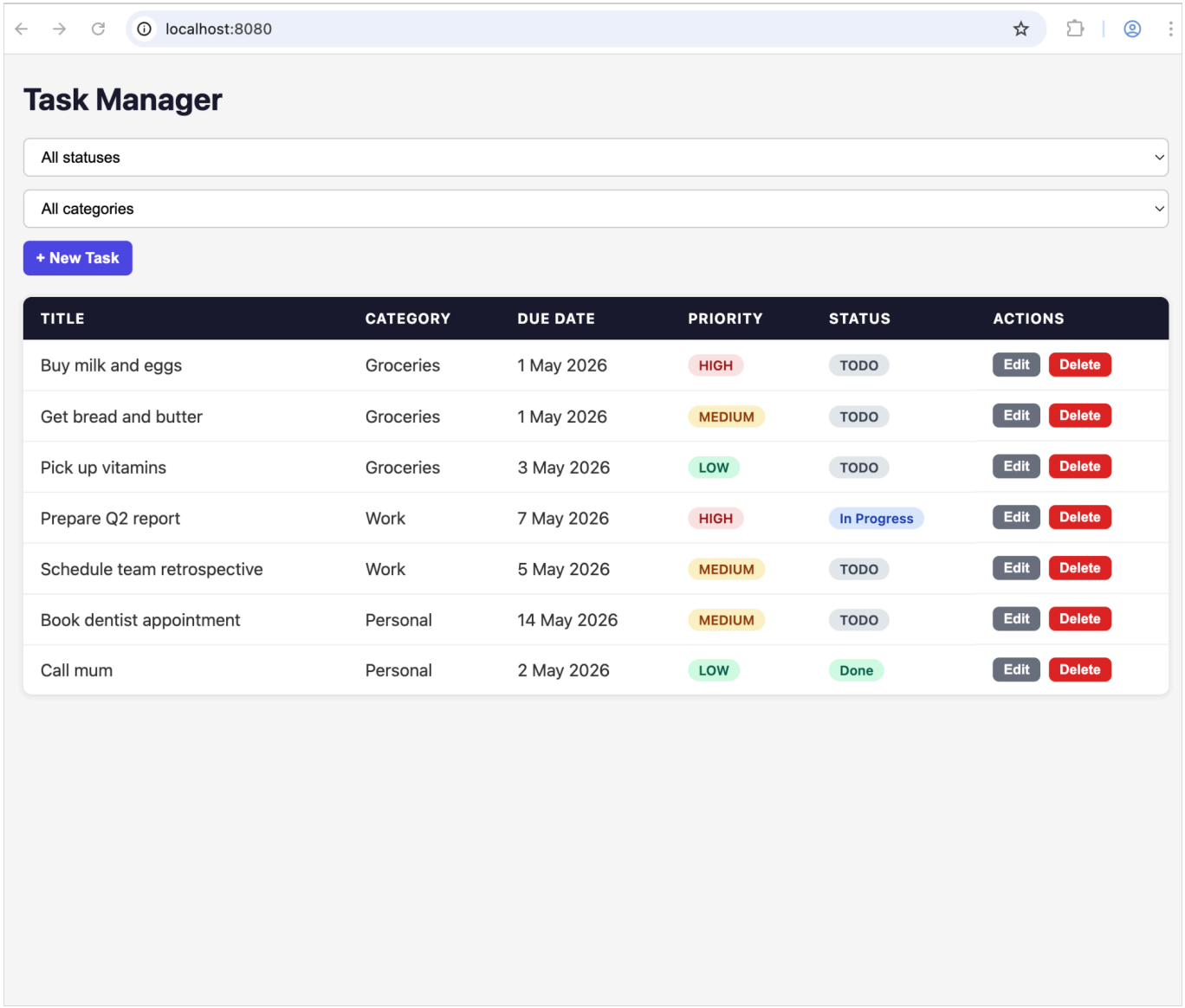
1. Run the `bootRun` Gradle task or enter the following command in the terminal:

```
./gradlew bootRun
```

2. In a browser, open the localhost URL. The default is usually:

```
http://localhost:8080
```

You should now see the basic UI that Claude created.



Run the app

Since Claude designs the UI, your UI may look different from the version in this tutorial.

Test the app

Now it's time for you to test the app.

Test the UI manually

Start by testing the UI functionality. Try some simple actions:

1. Create a task and test the form fields.
2. Edit a task to check that changes persist.
3. Change the status of a task.
4. Delete a task.
5. Change the category of a task.

If any of these actions don't work, send a new prompt to Claude asking it to investigate and fix the problem.

Run unit tests

Claude also creates some tests automatically. Check that all the tests pass by running:

```
./gradlew test
```

Alternatively, in the `src/test` directory, open a test and click the run icon in the gutter. A successful test shows .

If any test doesn't work, send a new prompt to Claude asking it to investigate and fix the problem.

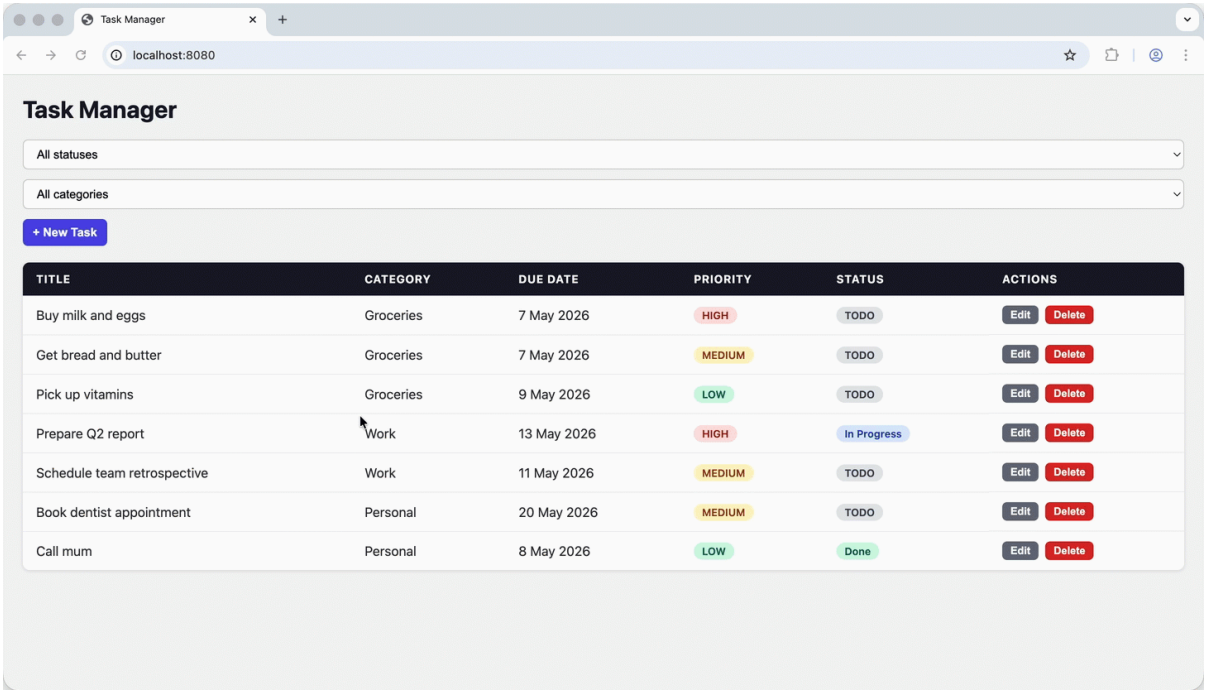
Make refinements

Now that the initial task is complete, you can make refinements. For example, let's improve the UI so users can edit tasks directly in the list.

You can send a prompt like:

```
As a next step, allow users to edit tasks inline. For example, let users click on a task title to edit
it directly in the list,
and update fields like priority, due date, or status without leaving the current view.
This change should make the app feel faster and more intuitive to use.
```

Just as before, Claude explores the existing project structure and suggests a plan. After you accept the plan, wait for Claude to finish, review the changes, and then run the app again.



Congratulations! You used Claude to plan, build, test, and refine a Kotlin Spring Boot application directly in IntelliJ IDEA.

What's next?

- Learn about Kotlin AI skills
- Check out our tutorial on using Junie with Kotlin AI skills

Overview

Kotlin, through Kotlin Multiplatform, offers two approaches for web development:

- JavaScript-based (using the Kotlin/JS compiler)
- WebAssembly-based (using the Kotlin/Wasm compiler)

Both approaches let you share code in your web app, but they support different use cases. They also differ in technical aspects such as target browser support.

Kotlin/JS

Kotlin/JS enables running Kotlin apps in JavaScript (JS) environments by transpiling your code, standard library, and all the supported dependencies into JS.

When developing with Kotlin/JS, you can run your apps either in the browser or the Node.js environment.

> For information about configuring Kotlin/JS targets, see the Configure a Gradle project guide.

Kotlin/JS use cases

Kotlin/JS is suitable when your goal is to:

- Share business logic with a JavaScript/TypeScript codebase.
- Build non-shareable web apps with Kotlin.

Share business logic with a JavaScript/TypeScript codebase

If you need to share Kotlin code (such as domain or data logic) with a native JavaScript/TypeScript app, the Kotlin/JS target offers:

- Direct interoperability with JavaScript/TypeScript.
- Minimal overhead in interoperability (for example, avoiding unnecessary data copying). This allows the shared code to integrate smoothly into a JS-based workflow.

Build web apps with Kotlin without sharing the code

For projects where the web app is implemented entirely in Kotlin, without sharing it with other platforms (iOS, Android, or Desktop), an HTML-based solution provides better control.

HTML-based solutions improve SEO and accessibility. They also provide better browser integration, including features such as find on page and page translation.

For HTML-based solutions, Kotlin/JS supports multiple approaches:

- Using Compose HTML-based frameworks, such as Kobweb or Kilua, to build UIs with a Compose-style architecture.
- Using React-based solutions with Kotlin wrappers to implement React components in Kotlin.

Beta

Kotlin/Wasm

Kotlin/Wasm compiles Kotlin code into WebAssembly (Wasm), enabling apps to run across environments and devices that support Wasm while meeting Kotlin's requirements.

In browsers, Kotlin/Wasm lets you build web apps with Compose Multiplatform. Outside the browser, it runs in standalone Wasm virtual machines, using the WebAssembly System Interface (WASI) to access platform APIs.

When developing with Kotlin/Wasm, you can target:

- `wasmJs`: for running in browsers or Node.js.
- `wasmWasi`: for running in Wasm environments supporting WASI, such as Wasmtime, WasmEdge, and so on.

For information about configuring Kotlin/Wasm targets, see the Configure a Gradle project guide.

Kotlin/Wasm use cases

Use Kotlin/Wasm if you want to share both logic and UI across multiple platforms.

Build cross-platform apps with Compose Multiplatform

If you want to share both logic and UI across multiple platforms, including web, Kotlin/Wasm with Compose Multiplatform provides a shared UI layer:

- Ensuring a consistent UI implementation for all platforms.
- Using Wasm for improved rendering and smoother UI updates, such as responsive animations.
- Supporting the latest version of the WebAssembly Garbage Collection (WasmGC) proposal, which allows Kotlin/Wasm to run on all major modern browsers.

Choose your web approach

The table below summarizes the recommended target based on your use case:

Use case	Recommended target	Description
Sharing business logic, but using web-native UI	Kotlin/JS	Offers straightforward interop with JS and minimal overhead.
Sharing both UI and business logic	Kotlin/Wasm	Provides better performance for rendering with Compose Multiplatform.

Use case	Recommended target	Description
Non-shareable UI	Kotlin/JS	Allows building UIs with HTML-based frameworks like Kobweb, Kilua, or React, using existing JS ecosystems and tooling.

If you need guidance on selecting the appropriate target, join our Slack community. You can ask about platform differences, performance considerations, and recommended practices for specific use cases.

Compatibility mode for web targets

You can enable compatibility mode for your web app to ensure it works on all browsers out of the box. In this mode, you can build the UI with Wasm for modern browsers, while older ones fall back to JS.

Compatibility mode is achieved through cross-compilation for both the `js` and `wasmJs` targets. See more information about compatibility mode for web and how to enable it.

Kotlin/JavaScript

Kotlin/JavaScript (Kotlin/JS) lets you transpile your Kotlin code, the Kotlin standard library, and any compatible dependencies to JavaScript. This way, your Kotlin applications can run in any environment that supports JavaScript.

Use Kotlin/JS through the Kotlin Multiplatform Gradle plugin (`kotlin.multiplatform`) to configure and manage Kotlin projects targeting JavaScript from a single place.

The Kotlin Multiplatform Gradle plugin gives you access to features such as controlling your application's bundling and adding JavaScript dependencies directly from npm. To get an overview of the available configuration options, see Set up a Kotlin/JS project.

The current implementation of Kotlin/JS targets the ES5 and ES2015 standards.

Use cases for Kotlin/JS

Here are some common ways to use Kotlin/JS:

- **Sharing common logic between frontend and JVM backend**

  If your backend is written in Kotlin or another JVM-compatible language, you can share common code between your web application and the backend. This includes data-transfer objects (DTOs), validation and authentication rules, abstractions for REST API endpoints, and more.

- **Sharing common logic between Android, iOS, and web clients**

  You can share business logic between your web interface and mobile applications for Android and iOS, while keeping native user interfaces. This avoids duplicating common functionality such as REST API abstractions, user authentication, form validation, and domain models.

- **Building frontend web applications using Kotlin/JS**

Use Kotlin to develop traditional web frontends while integrating with existing tools and libraries:

- If you are familiar with Android development, you can build web applications with Compose-based frameworks like Kobweb or Kilua.
- Build fully type-safe React applications with Kotlin/JS using the Kotlin wrappers for common JavaScript libraries provided by JetBrains. The Kotlin wrappers (`kotlin-wrappers`) offer abstractions and integrations for React and other JavaScript frameworks.

  These wrappers also support complementary libraries, like React Redux, React Router, and styled-components. You can also use third-party React components and component libraries through interoperability with the JavaScript ecosystem.
- Use Kotlin/JS frameworks, which integrate with the Kotlin ecosystem and support concise and expressive code.

- **Building multiplatform applications that support older browsers**

  With Compose Multiplatform, you can use Kotlin to build applications and reuse mobile and desktop user interfaces in your web projects. While Kotlin/Wasm is the primary target for this purpose, you can extend support to older browsers by also targeting Kotlin/JS.
- **Building server-side and serverless applications using Kotlin/JS**

  The Node.js target in Kotlin/JS lets you create applications for server-side or serverless environments on a JavaScript runtime. This offers a fast startup and low memory usage. The `kotlinx-nodejs` library provides type-safe access to the Node.js API from Kotlin.

Depending on your use case, Kotlin/JS projects can use compatible libraries from the Kotlin ecosystem and third-party libraries from the JavaScript and TypeScript ecosystems.

To use third-party libraries from your Kotlin code, you can create your own type-safe wrappers or use community-maintained wrappers. Additionally, you can use the Kotlin/JS dynamic type, which lets you skip strict typing and library wrappers, at the cost of type safety.

Kotlin/JS is also compatible with the most common module systems: ESM, CommonJS, UMD, and AMD. This allows you to produce and consume modules and integrate with the JavaScript ecosystem in a structured manner.

Share your use cases

The list in Use cases for Kotlin/JS isn't exhaustive. Feel free to experiment with different approaches and find the best fit for your project.

Share your use cases, experiences, and questions with the Kotlin/JS community in the #javascript channel on Kotlin Slack.

Get started with Kotlin/JS

Explore the fundamentals and initial steps to start working with Kotlin/JS:

- If you're new to Kotlin, start by reviewing the basic syntax and exploring the Kotlin tour.
- Check out the list of Kotlin/JS sample projects for inspiration. These samples contain useful code snippets and patterns that can help you get started with your projects.
- If you’re new to Kotlin/JS, start with the setup guide before exploring more advanced topics.

Would you like to try Kotlin/JS yourself?

Get started with Kotlin/JS

Sample projects for Kotlin/JS

The following table lists a set of sample projects demonstrating various Kotlin/JS use cases, architectures, and code-sharing strategies:

Project	Description
Petclinic with common code between Spring and Angular	Demonstrates how to avoid code duplication in enterprise applications by sharing data-transfer objects, validation and authentication rules, and abstractions for REST API endpoints. The code is shared between a Spring Boot backend and a Angular frontend.
Fullstack Conference CMS	Showcases multiple code-sharing approaches, ranging from the simplest to all-in code sharing between Ktor, Jetpack Compose, and Vue.js applications.
Todo App on a Compose-HTML-based Kobweb framework	Shows how to create a to-do list application by reusing an approach familiar to Android developers. It builds a client UI application powered by the Kobweb framework.
Simple logic sharing between Android, iOS, and web	Contains a template for building a project with common logic in Kotlin, which is consumed in platform-native UI applications on Android (Jetpack Compose), iOS (SwiftUI), and web (React).
Full-stack collaborative to-do list	Shows how to create a to-do list application for collaborative work using Kotlin Multiplatform with JS and JVM targets. It uses Ktor for the backend, and Kotlin/JS with React for the frontend.

Kotlin/JS frameworks

Kotlin/JS frameworks simplify web development by providing ready-to-use components, routing, state management, and other tools for building modern web applications.

Check the available frameworks for Kotlin/JS written by different authors.

Join the Kotlin/JS community

You can join the #javascript channel in the official Kotlin Slack to chat with the community and the Kotlin/JS team.

What's next

- Set up a Kotlin/JS project
- Run Kotlin/JS projects
- Debug Kotlin/JS code
- Run tests in Kotlin/JS

Get started with Kotlin/JS

This tutorial shows how to create a web application for the browser using Kotlin/JavaScript (Kotlin/JS). To create your app, choose the tool that best fits your workflow:

- IntelliJ IDEA: clone the project template from version control and work on it in IntelliJ IDEA.
- Gradle build system: create build files for your project manually to better understand how the setup works under the hood.

> Besides targeting the browser, with Kotlin/JS you can compile for other environments. For more information, see Execution environments.

Create your application in IntelliJ IDEA

To create your Kotlin/JS web application, you can use IntelliJ IDEA.

Set up the environment

1. Download and install the latest version of IntelliJ IDEA.
2. Install the Kotlin Multiplatform IDE plugin (not to be confused with the Kotlin Multiplatform Gradle plugin).

Create your project

1. In IntelliJ IDEA, select **File** | **New** | **Project from Version Control**.
2. Enter the URL of the Kotlin/JS template project:

   ```
   https://github.com/Kotlin/kmp-js-wizard
   ```

3. Click **Clone**.

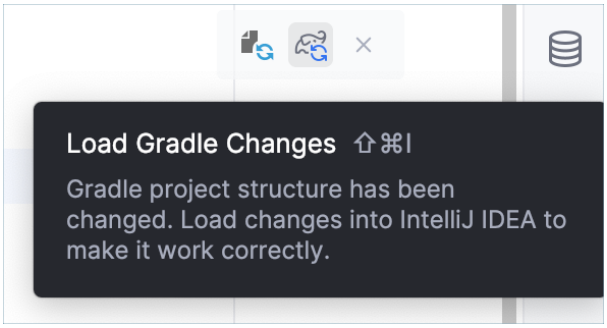
Configure your project

1. Open the `kmp-js-wizard/gradle/libs.versions.toml` file. It contains the version catalog for project dependencies.
2. Make sure the Kotlin version matches the version of the Kotlin Multiplatform Gradle plugin, which is required to create a web application targeting Kotlin/JS:

   ```
   [versions]
   kotlin = "2.4.20"

   [plugins]
   kotlin-multiplatform = { id = "org.jetbrains.kotlin.multiplatform", version.ref = "kotlin" }
   ```

3. Synchronize the Gradle files (if you updated the `libs.versions.toml` file). Click the **Load Gradle Changes** icon that appears in your build file.



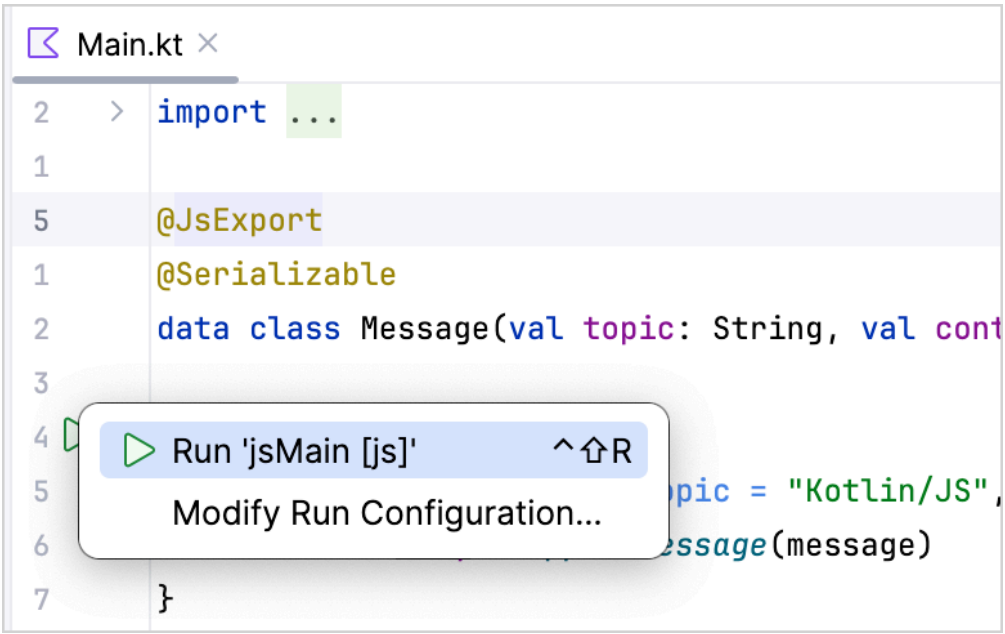
Load the Gradle changes button

Alternatively, click the refresh button in the Gradle tool window.

For more information about the Gradle configuration for multiplatform projects, see the Multiplatform Gradle DSL reference.

Build and run the application

1. Open the `src/jsMain/kotlin/Main.kt` file.
   - The `src/jsMain/kotlin/` directory contains the main Kotlin source files for the JavaScript target of your project.
   - The `Main.kt` file includes code that uses the `kotlinx.browser` API to render "Hello, Kotlin/JS!" on the browser page.
2. Click the **Run** icon in the `main()` function to run the code.

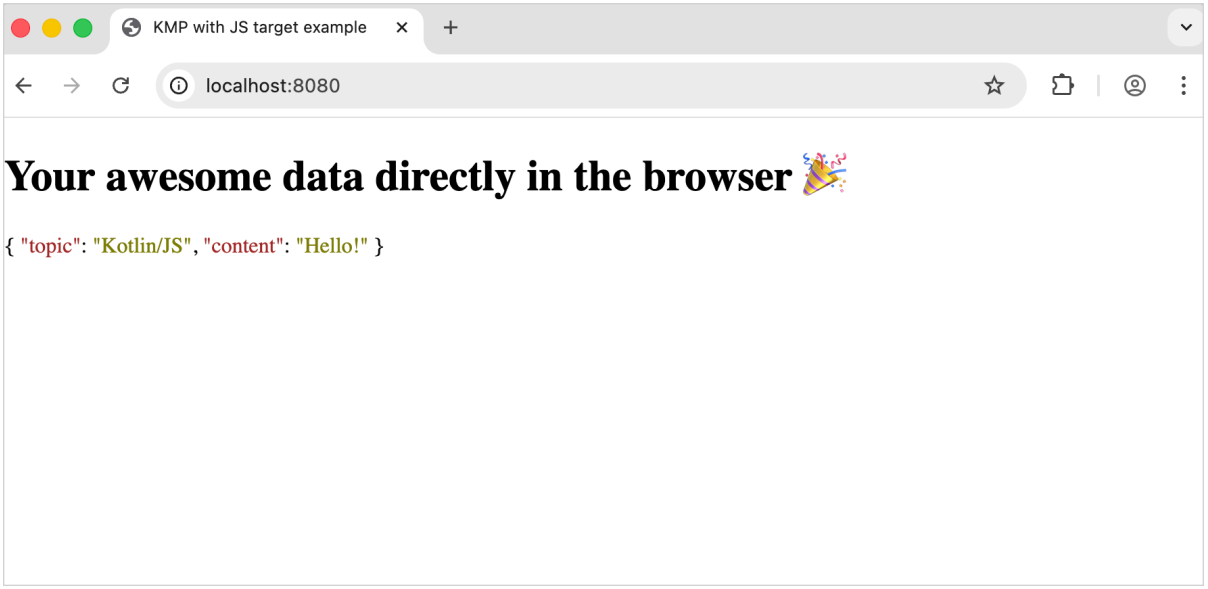


Run the application

The web application opens automatically in your browser. Alternatively, you can open the following URL in your browser when the run is finished:

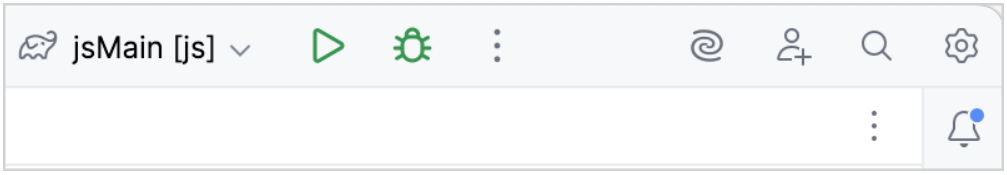
```
http://localhost:8080/
```

You can see the web application:



Application output

After you run the application for the first time, IntelliJ IDEA creates its corresponding run configuration (**jsMain [js]**) at the top bar:



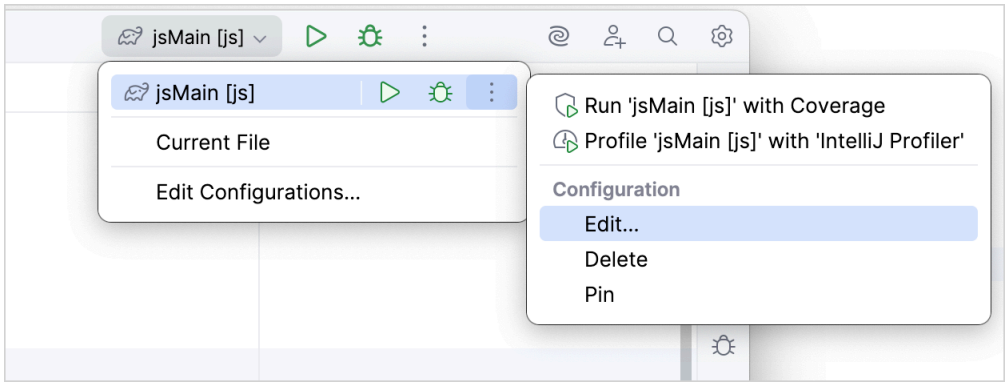
Gradle run configuration

> In IntelliJ IDEA with an Ultimate subscription, you can use the JS Debugger to debug code directly from the IDE.

Enable continuous build

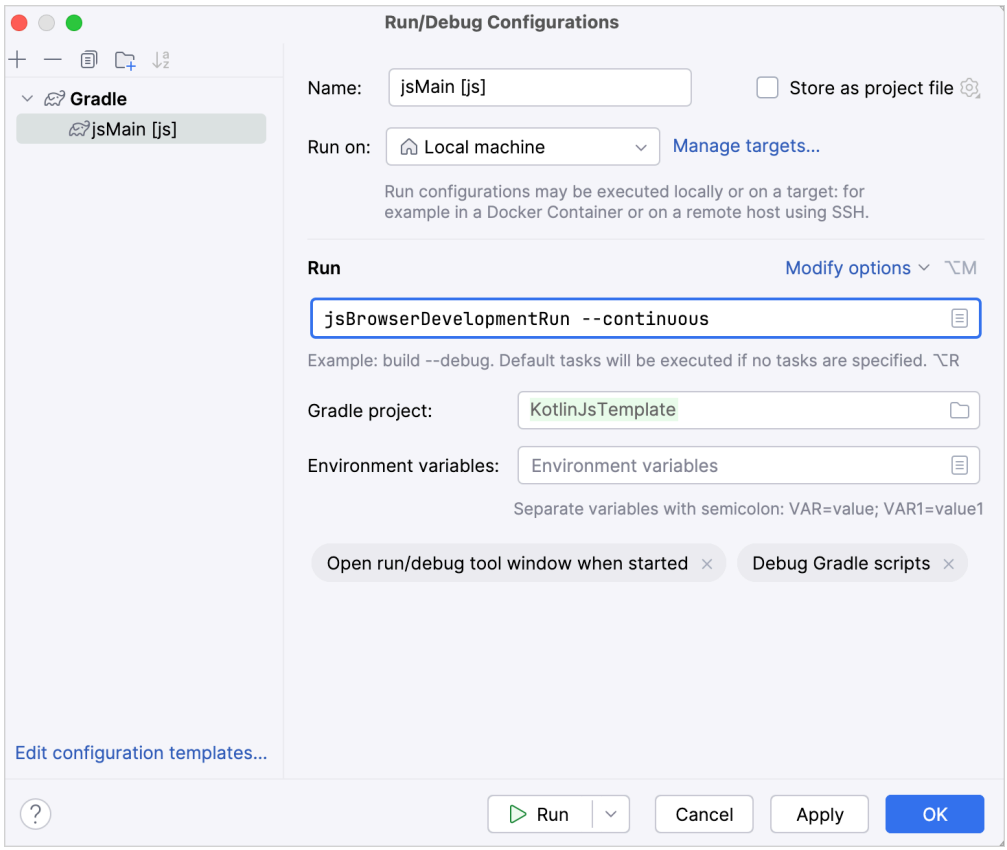
Gradle can automatically rebuild your project whenever you make changes:

1. Select **jsMain [js]** in the list of run configurations and click **More Actions** | **Edit**.



Gradle edit run configuration

2. In the **Run/Debug Configurations** dialog, enter `jsBrowserDevelopmentRun --continuous` within the **Run** field.



Continuous run configuration

3. Click **OK**.

Now, when you run the application and make any changes, Gradle automatically performs incremental builds for the project and hot-reloads the browser whenever you save (Ctrl + S/Cmd + S) or change a class file.

Modify the application

Modify the application to add a feature that counts the number of letters in a word.

Add an input element

1. In the `src/jsMain/kotlin/Main.kt` file, add an HTML input element through an extension function to read user input:

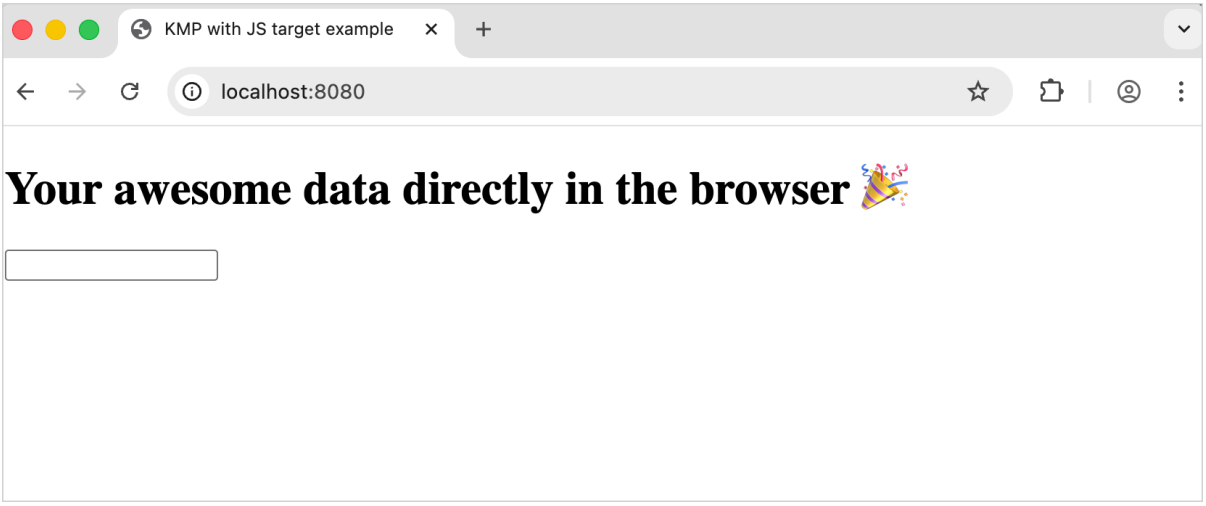
```
// Replace the Element.appendMessage() function
fun Element.appendInput() {
    val input = document.createElement("input")
    appendChild(input)
}
```

2. Call the `appendInput()` function in `main()`. It displays an input element on the page:

```
fun main() {
    // Replace document.body!!.appendMessage(message)
    document.body?.appendInput()
}
```

3. Run the application again.

   Your application looks like this:



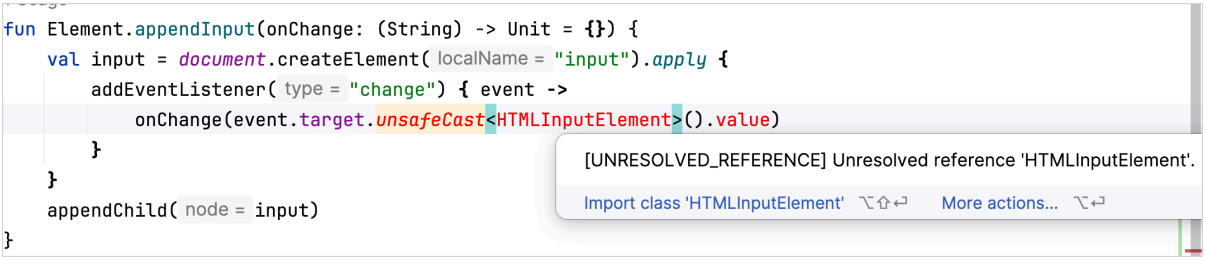
Application with an input element

Add an input event handling

1. Add a listener inside the `appendInput()` function to read the input value and react to changes:

```
// Replace the current appendInput() function
fun Element.appendInput(onChange: (String) -> Unit = {}) {
    val input = document.createElement("input").apply {
        addEventListener("change") { event ->
            onChange(event.target.unsafeCast<HTMLInputElement>().value)
        }
    }
    appendChild(input)
}
```

2. Follow the IDE's suggestions to import the `HTMLInputElement` dependency.



Import dependencies

3. Call the `onChange` callback in `main()`. It reads and handles the input value:

```
fun main() {
    // Replace document.body?.appendInput()
    document.body?.appendInput(onChange = { println(it) })
}
```

Add an output element

1. Add a text element to display the output by defining an extension function that creates a paragraph:

```
fun Element.appendTextContainer(): Element {
    return document.createElement("p").also(::appendChild)
}
```

2. Call the `appendTextContainer()` function in `main()`. It creates the output element:

```
fun main() {
    // Creates a text container for our output
    // Replace val message = Message(topic = "Kotlin/JS", content = "Hello!")
    val output = document.body?.appendTextContainer()

    // Reads the input value
    document.body?.appendInput(onChange = { println(it) })
}
```

Process the input to count the letters

Process the input by removing whitespace and displaying the output with the number of letters.

Add the following code to the `appendInput()` function within the `main()` function:

```
fun main() {
    // Creates a text container for our output
    val output = document.body?.appendTextContainer()

    // Reads the input value
    // Replace the current appendInput() function
    document.body?.appendInput(onChange = { name →
        name.replace(" ", "").let {
            output?.textContent = "Your name contains ${it.length} letters"
        }
    })
}
```

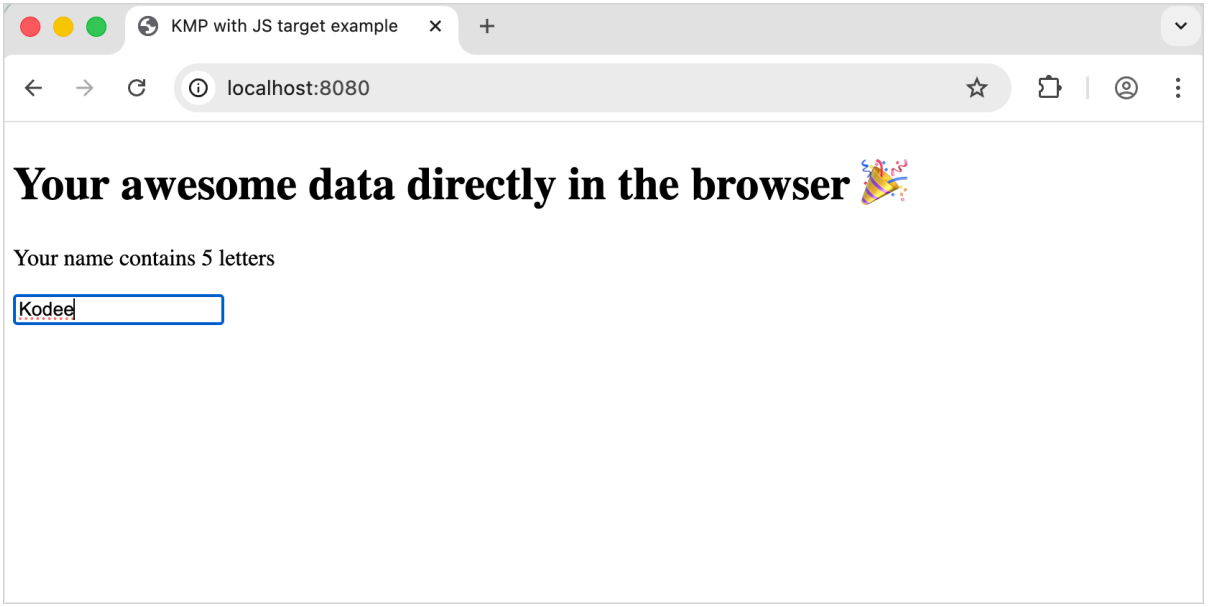
From the code above:

- The `replace()` function removes the empty spaces in the name.
- The `let{}` scope function runs the function within the object context.
- The string template (`${it.length}`) inserts the word's length into the string by prefixing it with a dollar sign (`$`) and enclosing it in curly braces (`{}`). Whereas `it` is the default name of a lambda parameter.

Run the application

1. Run the application.
2. Enter your name.
3. Press Enter.

You can see the result:



Application output

Process the input to count unique letters

As an additional exercise, let's process the input to calculate and display the number of unique letters in the word:

1. In the `src/jsMain/kotlin/Main.kt` file, add the `.countDistinctCharacters()` extension function for `String`:

   ```
   fun String.countDistinctCharacters() = lowercase().toList().distinct().count()
   ```

   From the code above:

   - The `.lowercase()` function converts the name to lowercase.
   - The `toList()` function converts the input string to a list of characters.
   - The `distinct()` function selects only the distinct characters from the word.
   - The `count()` function counts the distinct characters.

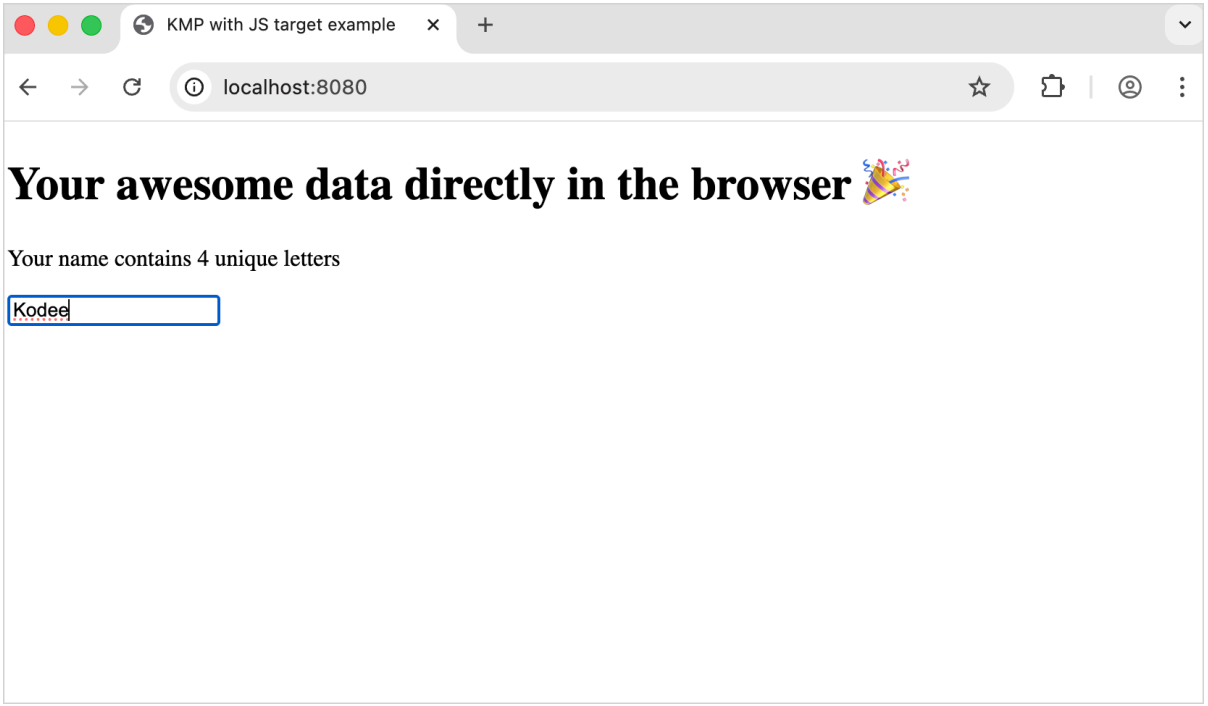
2. Call the `.countDistinctCharacters()` function in `main()`. It counts the unique letters in your name:

   ```
   fun main() {
       // Creates a text container for our output
       val output = document.body?.appendTextContainer()

       // Reads the input value
       document.body?.appendInput(onChange = { name ->
           name.replace(" ", "").let {
               // Prints the number of unique letters
               // Replace output?.textContent = "Your name contains ${it.length} letters"
               output?.textContent = "Your name contains ${it.countDistinctCharacters()} unique
   letters"
           }
       })
   }
   ```

3. Follow the steps to run the application and enter your name.

You can see the result:



Application output

Create your application using Gradle

In this section, you can learn how to manually create a Kotlin/JS application using Gradle.

Gradle is the default build system for Kotlin/JS and Kotlin Multiplatform projects. It is also commonly used in Java, Android, and other ecosystems.

Create project files

1. Make sure you use a Gradle version that is compatible with the Kotlin Gradle plugin (KGP). See the compatibility table for more details.
2. Create an empty directory for your project using your file explorer, the command line, or any tool you prefer.
3. Inside the project directory, create a `build.gradle.kts` file with the following content:

Kotlin

```
// build.gradle.kts
plugins {
    kotlin("multiplatform") version "2.4.20"
}

repositories {
    mavenCentral()
}

kotlin {
    js {
        // Use browser() for running in a browser or nodejs() for running in Node.js
        browser()
        binaries.executable()
    }
}
```

Groovy

```
// build.gradle
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.4.20'
}

repositories {
    mavenCentral()
}

kotlin {
    js {
        // Use browser() for running in a browser or nodejs() for running in Node.js
        browser()
        binaries.executable()
    }
}
```

You can use different execution environments, such as `browser()` or `nodejs()`. Each environment defines where your code runs and determines how Gradle generates task names in the project.

4. Within the project directory, create an empty `settings.gradle.kts` file.
5. Within the project directory, create a `src/jsMain/kotlin` directory.
6. Inside the `src/jsMain/kotlin` directory, add a `hello.kt` file with the following content:

```
fun main() {
    println("Hello, Kotlin/JS!")
}
```

By convention, all sources are located in the `src/<target name>[Main|Test]/kotlin` directory:

- `Main` is the location for the source code.
- `Test` is the location for tests.

- `<target name>` corresponds to the target platform (`js` in this case).

For the `browser` environment

> Follow the next steps if you are working with the `browser` environment. If you are working with the `nodejs` environment, go to the Build and run the project section.

1. Within the project directory, create a `src/jsMain/resources` directory.
2. Inside the `src/jsMain/resources` directory, create an `index.html` file with the following content:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Application title</title>
</head>
<body>
    <script src="$NAME_OF_YOUR_PROJECT_DIRECTORY.js"></script>
</body>
</html>
```

3. Replace the `<$NAME_OF_YOUR_PROJECT_DIRECTORY>` placeholder with the name of your project directory.

Build and run the project

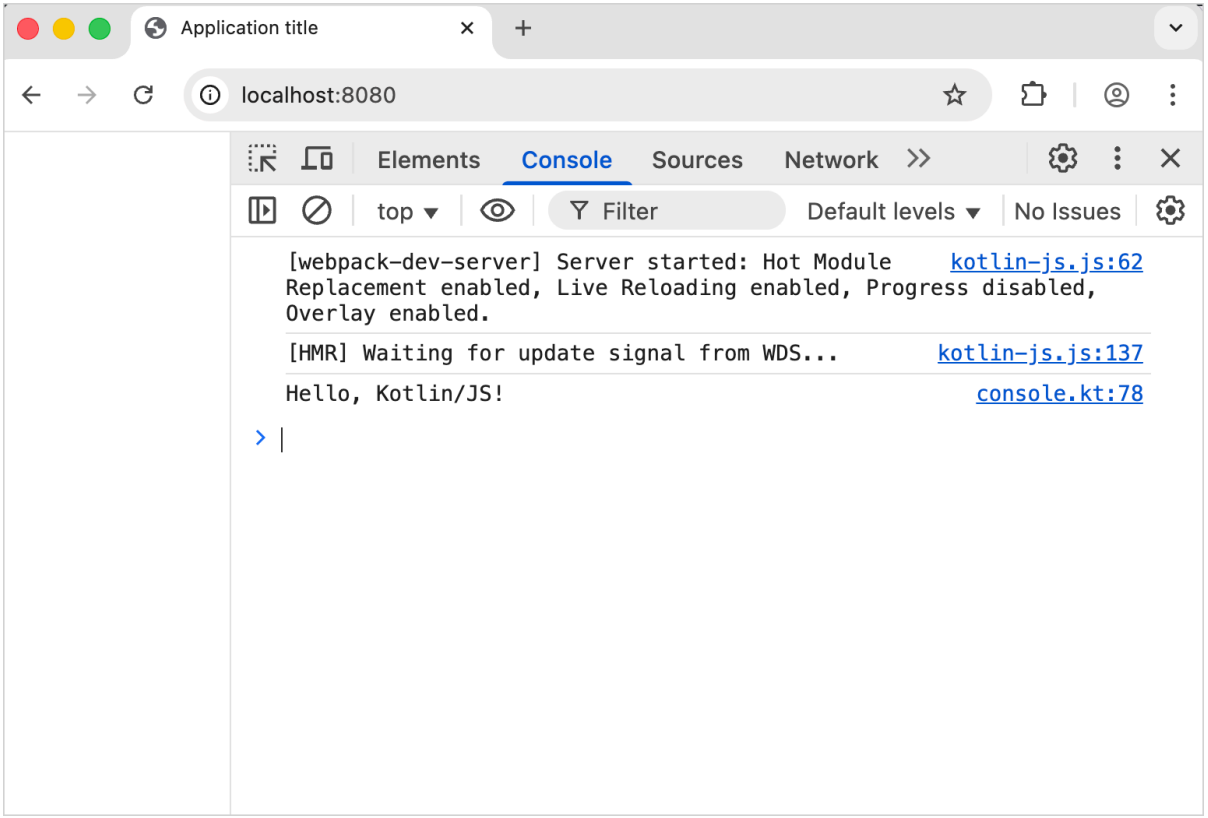
To build the project, run the following command from the root project directory:

```
# For browser
gradle jsBrowserDevelopmentRun

# OR

# For Node.js
gradle jsNodeDevelopmentRun
```

If you are using the `browser` environment, you can see that the browser opens the `index.html` file and prints `"Hello, Kotlin/JS!"` to the browser console. You can open the console using the Ctrl + Shift + J/Cmd + Option + J commands.



Application output

If you are using the `nodejs` environment, you can see that the terminal prints `"Hello, Kotlin/JS!"`.

```
> Task :jsNodeDevelopmentRun
Hello, Kotlin/JS!

BUILD SUCCESSFUL in 12s
17 actionable tasks: 17 executed
```

Application output

Open the project in an IDE

You can open your project in any IDE that supports Gradle.

If you use IntelliJ IDEA:

1. Select **File** | **Open**.
2. Find the project directory.
3. Click **Open**.

IntelliJ IDEA automatically detects if it's a Kotlin/JS project. If you encounter a problem with the project, IntelliJ IDEA displays the error message in the **Build** pane.

What's next?

- Set up your Kotlin/JS project.

- Learn how to debug Kotlin/JS applications.
- Learn how to write and run tests with Kotlin/JS.
- Learn how to write Gradle build scripts for real-life Kotlin/JS projects.
- Read more about the Gradle build system.

Set up a Kotlin/JS project

Kotlin/JS projects use Gradle as a build system. To let developers easily manage their Kotlin/JS projects, we offer the `kotlin.multiplatform` Gradle plugin that provides project configuration tools together with helper tasks for automating routines typical for JavaScript development.

The plugin downloads npm dependencies in the background using the npm or Yarn package managers and builds a JavaScript bundle from a Kotlin project using webpack. Dependency management and configuration adjustments can be done to a large part directly from the Gradle build file, with the option to override automatically generated configurations for full control.

You can apply the `org.jetbrains.kotlin.multiplatform` plugin to a Gradle project manually in the `build.gradle(.kts)` file:

Kotlin

```
plugins {
    kotlin("multiplatform") version "2.4.20"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.4.20'
}
```

The Kotlin Multiplatform Gradle plugin lets you manage aspects of your project in the `kotlin {}` block of the build script:

```
kotlin {
    // ...
}
```

Inside the `kotlin {}` block, you can manage the following aspects:

- Target execution environment: browser or Node.js
- Support for ES2015 features: classes, modules, and generators
- Configure output granularity
- Generation of TypeScript declaration files
- Project dependencies: Maven and npm
- Run configuration

- Test configuration
- Bundling and CSS support for browser projects
- Target directory and module name
- Project's `package.json` file

Execution environments

Kotlin/JS projects can target two different execution environments:

- Browser for client-side scripting in browsers
- Node.js for running JavaScript code outside a browser, for example, for server-side scripting.

To define the target execution environment for a Kotlin/JS project, add the `js {}` block with `browser {}` or `nodejs {}` inside:

```
kotlin {
    js {
        browser {
        }
        binaries.executable()
    }
}
```

The instruction `binaries.executable()` explicitly instructs the Kotlin compiler to emit executable `.js` files. Omitting `binaries.executable()` will cause the compiler to only generate Kotlin-internal library files, which can be used from other projects but not run on their own.

> This is typically faster than creating executable files, and can be a possible optimization when dealing with non-leaf modules of your project.

The Kotlin Multiplatform plugin automatically configures its tasks for working with the selected environment. This includes downloading and installing the required environment and dependencies for running and testing the application. This allows developers to build, run, and test simple projects without additional configuration. There is also an option to use an existing installation. Learn how to use pre-installed Node.js.

Support for ES2015 features

Kotlin provides support for ES2015 features, including:

- Modules that simplify your codebase and improve maintainability.
- Classes that allow incorporating OOP principles, resulting in cleaner and more intuitive code.
- Generators for compiling suspend functions that improve the final bundle size and help with debugging.
- JavaScript code inlining.

You can enable all the supported ES2015 features at once by adding the `es2015` compilation target to your `build.gradle(.kts)` file:

```
tasks.withType<KotlinJsCompile>().configureEach {
    compilerOptions {
        target = "es2015"
    }
}
```

Learn more about ES2015 (ECMAScript 2015, ES6) in the official documentation.

Configure output granularity

You can choose how the compiler outputs `.js` files in your project:

- **One per module**. By default, the JS compiler outputs separate `.js` files for each project module as a compilation result.
- **One per project**. You can compile the whole project into a single `.js` file by adding the following line to the `gradle.properties` file:

  ```
  kotlin.js.ir.output.granularity=whole-program // 'per-module' is the default
  ```

- **One per file**. You can set up a more granular output that generates one (or two, if the file contains exported declarations) JavaScript file per each Kotlin file. To enable the per-file compilation mode:

  1. Set `es2015` as the compilation target to support ES2015 features in your project.
  2. Add the following line to the `gradle.properties` file:

     ```
     kotlin.js.ir.output.granularity=per-file // 'per-module' is the default
     ```

Experimental

Generation of TypeScript declaration files (`d.ts`)

The Kotlin/JS compiler can generate TypeScript definitions from your Kotlin code. These definitions can be used by JavaScript tools and IDEs when working on hybrid applications to:

- Provide autocompletion
- Support static analyzers
- Simplify the addition of Kotlin code in JavaScript and TypeScript projects

Generating TypeScript definitions is especially valuable for the business logic sharing use cases.

The compiler collects any top-level declarations marked with `@JsExport` and automatically generates TypeScript definitions in a `.d.ts` file.

To generate TypeScript definitions, explicitly configure it in your Gradle build file. Add the `generateTypeScriptDefinitions()` function to your `build.gradle.kts` file in the `js {}` block:

```
kotlin {
    js {
        binaries.executable()
        browser {
        }
        generateTypeScriptDefinitions()
    }
}
```

You can find the definitions in the `build/js/packages/<package_name>/kotlin` directory alongside the corresponding un-webpacked JavaScript code.

Dependencies

To declare a dependency, use the `dependencies {}` block in the `jsMain` source set in your `build.gradle(.kts)` file:

Kotlin

```
kotlin {
    sourceSets {
        jsMain {
            dependencies {
                implementation("org.example.myproject:1.1.0")
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        jsMain {
            dependencies {
                implementation 'org.example.myproject:1.1.0'
            }
        }
    }
}
```

Only libraries that include artifacts for Kotlin/JS can be used as dependencies. To resolve dependencies, declare the repositories where Gradle should look for them in the `repositories {}` block of your `build.gradle(.kts)` file. For example:

```
repositories {
    mavenCentral()
}
```

If the library you are adding has dependencies on packages from npm, Gradle automatically resolves these transitive dependencies as well.

Kotlin standard libraries

The dependencies on the standard library are added automatically. The version of the standard library is the same as the version of the Kotlin Multiplatform plugin.

For multiplatform tests, the `kotlin.test` API is available. When you create a multiplatform project, you can add test dependencies to all the source sets by using a single dependency in `commonTest`:

Kotlin

```
kotlin {
    sourceSets {
        commonTest.dependencies {
            implementation(kotlin("test")) // Brings all the platform dependencies automatically
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // Brings all the platform dependencies automatically
            }
        }
    }
}
```

npm dependencies

In the JavaScript world, the most common way to manage dependencies is npm. It offers the biggest public repository of JavaScript modules.

The Kotlin Multiplatform Gradle plugin lets you declare npm dependencies in the Gradle build script, just how you declare any other dependencies.

To declare an npm dependency, pass its name and version to the `npm()` function inside a dependency declaration. You can also specify one or multiple version ranges based on npm's semver syntax.

Kotlin

```
kotlin {
    sourceSets {
        jsMain {
            dependencies {
                implementation(npm("core-js", "^3.38.1"))
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        jsMain {
            dependencies {
                implementation npm('core-js', '^3.38.1')
            }
        }
    }
}
```

By default, the plugin uses a separate instance of the Yarn package manager to download and install npm dependencies. It works out of the box without additional configuration, but you can tune it to specific needs.

You can also work with npm dependencies directly using the npm package manager instead. To use npm as your package manager, in your `gradle.properties` file, set the following property:

```
kotlin.js.yarn=false
```

Besides regular dependencies, there are three more types of dependencies that can be used from the Gradle DSL. To learn more about when each type of dependency can best be used, have a look at the official documentation linked from npm:

- devDependencies, via `devNpm(...)`,
- optionalDependencies via `optionalNpm(...)`, and
- peerDependencies via `peerNpm(...)`.

Once an npm dependency is installed, you can use its API in your code as described in Calling JS from Kotlin.

run task

The Kotlin Multiplatform Gradle plugin provides a `jsBrowserDevelopmentRun` task that lets you run pure Kotlin/JS projects without additional configuration.

For running Kotlin/JS projects in the browser, this task is an alias for the `browserDevelopmentRun` task (which is also available in Kotlin multiplatform projects). It uses the webpack-dev-server to serve your JavaScript artifacts. If you want to customize the configuration used by `webpack-dev-server`, for example, adjust the port the server runs on, use the webpack configuration file.

For running Kotlin/JS projects targeting Node.js, use the `jsNodeDevelopmentRun` task that is an alias for the `nodeRun` task.

To run a project, execute the standard lifecycle `jsBrowserDevelopmentRun` task, or the alias to which it corresponds:

```
./gradlew jsBrowserDevelopmentRun
```

To automatically trigger a re-build of your application after making changes to the source files, use the Gradle continuous build feature:

```
./gradlew jsBrowserDevelopmentRun --continuous
```

or

```
./gradlew jsBrowserDevelopmentRun -t
```

Once the build of your project has succeeded, the `webpack-dev-server` will automatically refresh the browser page.

test task

The Kotlin Multiplatform Gradle plugin automatically sets up a test infrastructure for projects. It downloads and installs the required test runners and other dependencies.

For browser projects, you can choose between the Karma test runner and the new DSL for browser testing. For Node.js projects, the Mocha test framework is available.

The plugin also provides useful testing features, for example:

- Source map generation
- Test report generation
- Test run results in the console

Karma

> The Karma project has been deprecated. No new features or bug fixes are expected. As an alternative for browser testing, try out the new DSL for browser testing.

To configure the Karma test runner, add the `useKarma {}` block inside the browser's `testTask` in your `build.gradle(.kts)` file. For example, to run tests against specific browsers, use:

```
kotlin {
    js {
        browser {
            testTask {
                useKarma {
                    useIe()
                    useSafari()
                    useFirefox()
                    useChrome()
                    useChromeCanary()
                    useChromeHeadless()
                    usePhantomJS()
                    useOpera()
                }
            }
        }
        binaries.executable()
        // ...
    }
}
```

Alternatively, you can add test targets for browsers in the `gradle.properties` file:

```
kotlin.js.browser.karma.browsers=firefox,safari
```

This allows you to define a list of browsers for all modules and then add specific browsers in the build files of particular modules.

The Kotlin Multiplatform Gradle plugin automatically generates a Karma configuration file at `build/js/packages/projectName-test/karma.conf.js` at build time. The file includes settings you set in the `useKarma {}` block in your build file.

You can also place additional configuration files inside the `karma.config.d` directory at the root of your project. All `.js` configuration files in this directory are picked up and automatically merged into the generated `karma.conf.js` at build time.

For more information on the Karma configuration, see Karma's documentation.

Experimental

DSL for browser testing

Kotlin offers an experimental DSL for running Kotlin/JS tests in a browser environment. It's designed to be technology-agnostic. The current implementation includes the following tools under the hood:

- Playwright acts as a browser driver and a distribution manager that supports the Chromium, Firefox, and WebKit (Safari) browser engines.
- Mocha acts as a test runner.
- webpack acts as a bundler (will be replaced with Vite in future releases).

To try out the new DSL for browser testing, add the opt-in `test {}` block inside `browser {}` for your Kotlin/JS target:

```
import org.jetbrains.kotlin.gradle.ExperimentalJsTestDsl
import kotlin.time.Duration.Companion.seconds

kotlin {
    js {
        browser {
            @OptIn(ExperimentalJsTestDsl::class)
            test {
                // Configures the default timeout for all runners with kotlin.Duration
                timeout = 30.seconds
                // Configures headless mode using Gradle providers
                headless = providers
                    .environmentVariable("IS_IN_CI")
                    .map { it.toBoolean() }
                    .orElse(false)
                // Enables and configures a custom Chromium runner
                chromium("chromium-no-webgl2") {
                    // Overrides the default timeout for this runner
                    timeout = 10.seconds

                    // Chromium-specific extra launch argument
                    launchArgs.add("--disable-webgl2")
                }
                // Enables the Firefox runner
                firefox()
                // Enable WebKit (Safari) test runner
                webkit()
                // Enables and configures a custom WebKit runner
                webkit("headful") {
                    headless = false
                }
            }
        }
    }
}
```

For more information on the configuration of the new DSL for browser testing, see Run tests in Kotlin/JS.

Node.js

For Node.js projects, the Kotlin Multiplatform Gradle plugin automatically sets up the Mocha test framework.

To specify environment variables used by your Node.js test runners (for example, to pass external information to your tests, or to fine-tune package resolution), use the `environment()` function with a key-value pair inside the `testTask {}` block in your build file:

```
kotlin {
    js {
        nodejs {
            testTask {
                environment("key", "value")
            }
        }
    }
}
```

Run tests

By default, the Kotlin Multiplatform Gradle plugin uses Headless Chrome to run browser tests. No browsers are bundled with the plugin; test runners handle missing browsers differently:

- With Karma, any other browser should already be installed on your machine so that the plugin can use it to run tests. If you are executing Kotlin/JS tests on a continuous integration server, ensure that the browsers you want to test against are installed there as well.
- With the new DSL for browser testing, the plugin installs the necessary browsers on the first run by using the `playwright install` command. Playwright manages the location of these browsers and doesn't use locally installed browsers.

To run tests, execute the standard lifecycle `check` task:

```
./gradlew check
```

If you want to skip tests, disable them in the `testTask {}` block in your build file:

```
kotlin {
    js {
        browser {
            testTask {
                enabled.set(false)
            }
        }
        binaries.executable()
        // ...
    }
}
```

webpack bundling

For browser targets, the Kotlin Multiplatform Gradle plugin uses the widely known webpack module bundler.

webpack task

The most common webpack adjustments can be made directly via the `kotlin.js.browser.webpackTask {}` configuration block in the Gradle build file:

- `mainOutputFileName` - the name of the webpacked output file. It will be generated in `<projectDir>/build/kotlin-webpack/<targetName>/<binaryName>` after an execution of a webpack task. The default value is the project name.
- `output.libraryTarget` - the module system for the webpacked output. Learn more about available module systems for Kotlin/JS projects. The default value is `umd`.

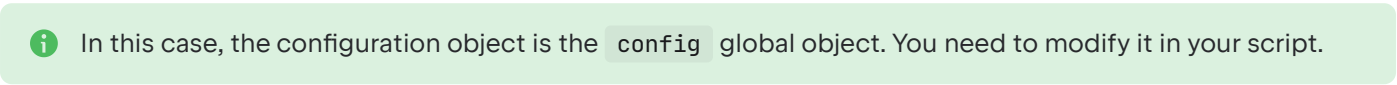
```
webpackTask {
    mainOutputFileName = "mycustomfilename.js"
    output.libraryTarget = "commonjs2"
}
```

You can also configure common webpack settings to use in bundling, running, and testing tasks in the `commonWebpackConfig {}` block.

webpack configuration file

The Kotlin Multiplatform Gradle plugin automatically generates a standard webpack configuration file at the build time. It is located in `build/js/packages/projectName/webpack.config.js`.

If you want to make further adjustments to the webpack configuration, place your additional configuration files inside a directory called `webpack.config.d` in the root of your project. When building your project, all `.js` configuration files will automatically be merged into the `build/js/packages/projectName/webpack.config.js` file. For example, To add a new webpack loader, add the following to a `.js` file inside the `webpack.config.d` directory:



In this case, the configuration object is the `config` global object. You need to modify it in your script.

```
config.module.rules.push({
    test: /\.extension$/,
    loader: 'loader-name'
});
```

All webpack configuration capabilities are well described in its documentation.

Building executables

For building executable JavaScript artifacts through webpack, the Kotlin Multiplatform Gradle plugin contains the `jsBrowserDevelopmentWebpack` and `jsBrowserProductionWebpack` Gradle tasks.

- `jsBrowserDevelopmentWebpack` creates development artifacts, which are larger but take little time to create. As such, use the `jsBrowserDevelopmentWebpack` tasks during active development.
- `jsBrowserProductionWebpack` applies dead code elimination to the generated artifacts and minifies the resulting JavaScript file, which takes more time but generates executables that are smaller. As such, use the `jsBrowserProductionWebpack` task when preparing your project for production use.

Execute either of these tasks to obtain the respective artifacts for development or production. The generated files will be available in `build/kotlin-webpack` unless specified otherwise.

```
./gradlew jsBrowserProductionWebpack
```

Note that these tasks will only be available if your target is configured to generate executable files (via `binaries.executable()`).

To generate the distribution in the `build/dist/<targetName>/<binaryName>` directory, run the `jsBrowserDistribution` task instead:

```
./gradlew jsBrowserDistribution
```

This task generates a ready-to-use distribution that includes the project resources.

CSS

The Kotlin Multiplatform Gradle plugin also provides support for webpack's CSS and style loaders. While all options can be changed by directly modifying the webpack configuration files that are used to build your project, the most commonly used settings are available directly from the `build.gradle(.kts)` file.

To turn on CSS support in your project, set the `cssSupport.enabled` option in the Gradle build file in the `commonWebpackConfig {}` block. This configuration is also enabled by default when creating a new project using the wizard.

Kotlin

```
browser {
    commonWebpackConfig {
        cssSupport {
            enabled.set(true)
        }
    }
}
```

Groovy

```
browser {
    commonWebpackConfig {
        cssSupport {
            it.enabled = true
        }
    }
}
```

Alternatively, you can add CSS support independently for `webpackTask {}`, `runTask {}`, and `testTask {}`:

Kotlin

```
browser {
    webpackTask {
        cssSupport {
            enabled.set(true)
        }
    }
    runTask {
        cssSupport {
            enabled.set(true)
        }
    }
    testTask {
        useKarma {
            // ...
            webpackConfig.cssSupport {
                enabled.set(true)
            }
        }
    }
}
```

Groovy

```
browser {
    webpackTask {
        cssSupport {
            it.enabled = true
        }
    }
    runTask {
        cssSupport {
            it.enabled = true
        }
    }
    testTask {
        useKarma {
            // ...
            webpackConfig.cssSupport {
                it.enabled = true
            }
        }
    }
}
```

Activating CSS support in your project helps prevent common errors that occur when trying to use style sheets from an unconfigured project, such as `Module parse failed: Unexpected character '@' (14:0)`.

You can use `cssSupport.mode` to specify how encountered CSS should be handled. The following values are available:

- `"inline"` (default): styles are added to the global `<style>` tag.
- `"extract"`: styles are extracted into a separate file. They can then be included from an HTML page.
- `"import"`: styles are processed as strings. This can be useful if you need access to the CSS from your code (such as `val styles = require("main.css")`).

To use different modes for the same project, use `cssSupport.rules`. Here, you can specify a list of `KotlinWebpackCssRules`, each of which defines a mode, as well as include and exclude patterns.

Node.js

For Kotlin/JS projects targeting Node.js, the plugin automatically downloads and installs the Node.js environment on the host. You can also use an existing Node.js instance if you have it.

You can configure Node.js settings for each subproject or set them for the project as a whole.

Change Node.js version

The default Node.js version is currently 24.16.0, but you can use a different version for a specific subproject. Add the following lines to your subproject's `build.gradle(.kts)` file. For example:

Kotlin

```
project.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugin> {
    project.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsEnvSpec>().version = "26.2.0"
}
```

Groovy

```
project.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugin) {
    project.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsEnvSpec).version
= "26.2.0"
}
```

To set a version for the entire project, including all subprojects, apply the same code to the `allprojects {}` block. For example:

Kotlin

```
allprojects {
    project.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugin> {
        project.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsEnvSpec>().version = "26.2.0"
    }
}
```

Groovy

```
allprojects {
    project.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugin) {

project.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsEnvSpec).version =
"26.2.0"
    }
}
```

Use pre-installed Node.js

If Node.js is already installed on the host where you build Kotlin/JS projects, you can configure the Kotlin Multiplatform Gradle plugin to use it instead of installing its own Node.js instance.

To use a pre-installed Node.js instance, add the following lines to your `build.gradle(.kts)` file:

Kotlin

```
project.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugin> {
    // Set to `true` for default behavior
    project.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsEnvSpec>().download = false
}
```

Groovy

```
project.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugin) {
    // Set to `true` for default behavior
    project.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsEnvSpec).download
= false
}
```

Yarn

By default, to download and install your declared dependencies at build time, the plugin manages its own instance of the Yarn package manager. It works out of the box without additional configuration, but you can tune it or use Yarn already installed on your host.

Additional Yarn features: .yarnrc

To configure additional Yarn features, place a `.yarnrc` file in the root of your project. At build time, it gets picked up automatically.

For example, to use a custom registry for npm packages, add the following line to a file called `.yarnrc` in the project root:

```
registry "http://my.registry/api/npm/"
```

To learn more about `.yarnrc`, visit the official Yarn documentation.

Use pre-installed Yarn

If Yarn is already installed on the host where you build Kotlin/JS projects, you can configure the Kotlin Multiplatform Gradle plugin to use it instead of installing its own Yarn instance.

To use the pre-installed Yarn instance, add the following lines to `build.gradle(.kts)`:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootEnvSpec>().download = false
    // "true" for default behavior
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootEnvSpec).download
= false
}
```

Version locking via kotlin-js-store

The `kotlin-js-store` directory in the project root is automatically generated by the Kotlin Multiplatform Gradle plugin to hold the `yarn.lock` file, which is necessary for version locking. The lockfile is entirely managed by the Yarn plugin and gets updated during the execution of the `kotlinNpmInstall` Gradle task.

To follow a recommended practice, commit `kotlin-js-store` and its contents to your version control system. It ensures that your application is being built with the exact same dependency tree on all machines.

If needed, you can change both the directory and lockfile names in `build.gradle(.kts)`:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().lockFileDirectory
=
        project.rootDir.resolve("my-kotlin-js-store")
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().lockFileName =
"my-yarn.lock"
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).lockFile
=
        file("my-kotlin-js-store")

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).lockFile
= 'my-yarn.lock'
}
```

Changing the name of the lockfile may cause dependency inspection tools to no longer pick up the file.

To learn more about `yarn.lock`, visit the official Yarn documentation.

Reporting that yarn.lock has been updated

Kotlin/JS provides Gradle settings that could notify you if the `yarn.lock` file has been updated. You can use these settings when you want to be notified if `yarn.lock` has been changed silently during the CI build process:

- `YarnLockMismatchReport`, which specifies how changes to the `yarn.lock` file are reported. You can use one of the following values:
  - `FAIL` fails the corresponding Gradle task. This is the default.
  - `WARNING` writes the information about changes in the warning log.
  - `NONE` disables reporting.
- `reportNewYarnLock`, which reports about the recently created `yarn.lock` file explicitly. By default, this option is disabled: it's a common practice to generate a new `yarn.lock` file at the first start. You can use this option to ensure that the file has been committed to your repository.

- `yarnLockAutoReplace`, which replaces `yarn.lock` automatically every time the Gradle task is run.

To use these options, update `build.gradle(.kts)` as follows:

Kotlin

```
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnLockMismatchReport
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension

rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<YarnRootExtension>().yarnLockMismatchReport =
        YarnLockMismatchReport.WARNING // NONE | FAIL
    rootProject.the<YarnRootExtension>().reportNewYarnLock = false // true
    rootProject.the<YarnRootExtension>().yarnLockAutoReplace = false // true
}
```

Groovy

```
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnLockMismatchReport
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension

rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).yarnLock
=
        YarnLockMismatchReport.WARNING // NONE | FAIL

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).reportNe
= false // true

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).yarnLock
= false // true
}
```

Installing npm dependencies with --ignore-scripts by default

To reduce the likelihood of executing malicious code from compromised npm packages, the Kotlin Multiplatform Gradle plugin prevents the execution of lifecycle scripts during the installation of npm dependencies by default.

You can explicitly enable lifecycle scripts execution by adding the following lines to `build.gradle(.kts)`:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().ignoreScripts =
false
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).ignoreSc
= false
}
```

Distribution target directory

By default, the results of a Kotlin/JS project build reside in the `/build/dist/<targetName>/<binaryName>` directory within the project root.

To set another location for project distribution files, in your build script inside the `browser {}` block, add a `distribution {}` block and assign a value to the `outputDirectory` property by using the `set()` method. Once you run a project build task, Gradle will save the output bundle in this location together with project resources.

Kotlin

```
kotlin {
    js {
        browser {
            distribution {
                outputDirectory.set(projectDir.resolve("output"))
            }
        }
        binaries.executable()
        // ...
    }
}
```

Groovy

```
kotlin {
    js {
        browser {
            distribution {
                outputDirectory = file("$projectDir/output")
            }
        }
        binaries.executable()
        // ...
    }
}
```

Module name

To adjust the name for the JavaScript *module* (which is generated in `build/js/packages/myModuleName`), including the corresponding `.js` and `.d.ts` files, use the `outputModuleName` option:

```
kotlin {
    js {
        outputModuleName = "myModuleName"
    }
}
```

Note that this does not affect the webpacked output in `build/dist`.

package.json customization

The `package.json` file holds the metadata of a JavaScript package. Popular package registries such as npm require all published packages to have such a file. They use it to track and manage package publications.

The Kotlin Multiplatform Gradle plugin automatically generates `package.json` for Kotlin/JS projects during build time. By default, the file contains essential data: name, version, license, dependencies, and some other package attributes.

Aside from basic package attributes, `package.json` can define how a JavaScript project should behave, for example, identifying scripts that are available to run.

You can add custom entries to the project's `package.json` via the Gradle DSL. To add custom fields to your `package.json`, use the `customField()` function in the compilations' `packageJson` block:

```
kotlin {
    js {
        compilations["main"].packageJson {
            customField("hello", mapOf("one" to 1, "two" to 2))
        }
    }
}
```

When you build the project, this code adds the following block to the `package.json` file:

```
"hello": {
    "one": 1,
    "two": 2
}
```

Learn more about writing `package.json` files for the npm registry in the npm docs.

Run Kotlin/JS

Since Kotlin/JS projects are managed with the Kotlin Multiplatform Gradle plugin, you can run your project using the appropriate tasks. If you're starting with a blank project, ensure that you have some sample code to execute. Create the file `src/jsMain/kotlin/App.kt` and fill it with a small "Hello, World"-type code snippet:

```
fun main() {
    console.log("Hello, Kotlin/JS!")
}
```

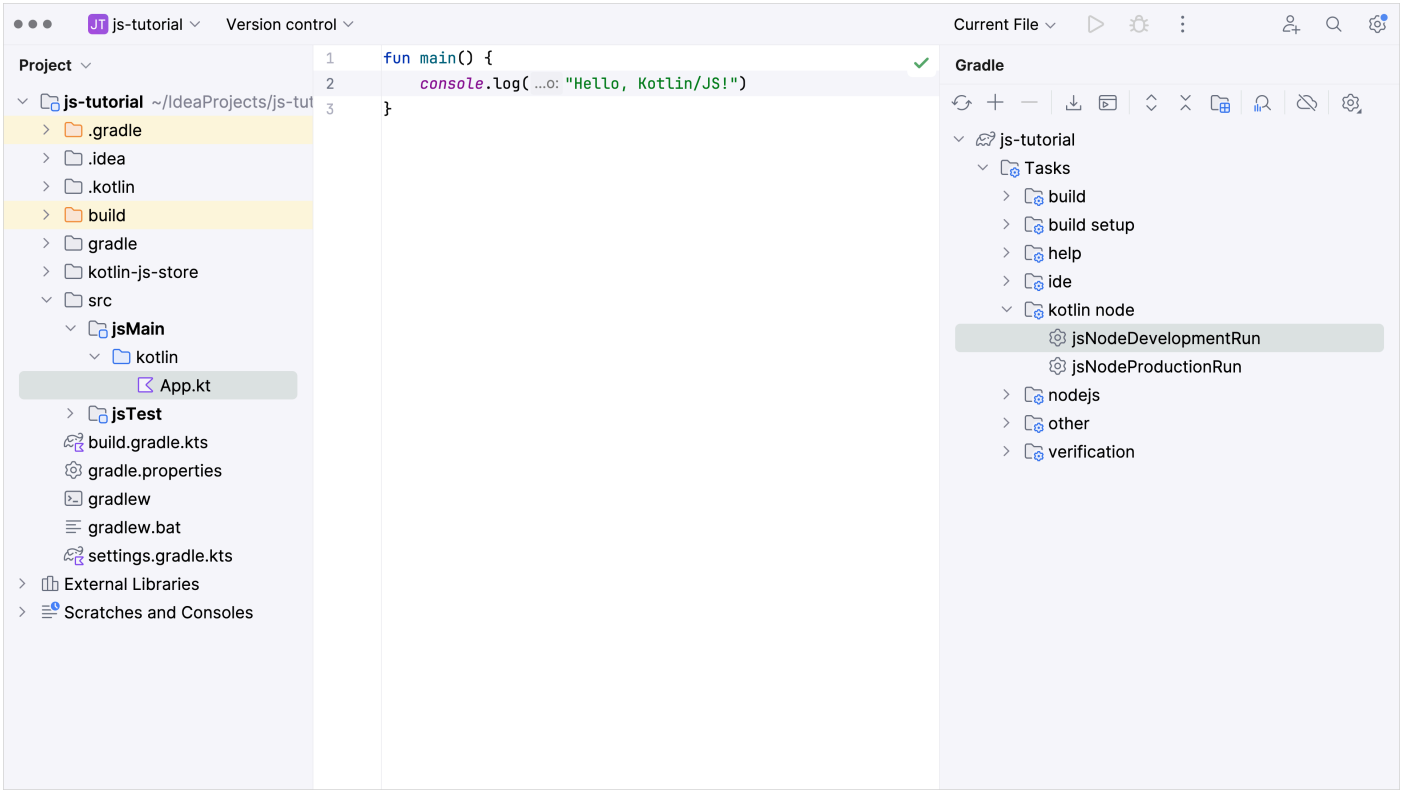
Depending on the target platform, some platform-specific extra setup might be required to run your code for the first time.

Run the Node.js target

When targeting Node.js with Kotlin/JS, you can simply execute the `jsNodeDevelopmentRun` Gradle task. This can be done, for example, via the command line, using the Gradle wrapper:

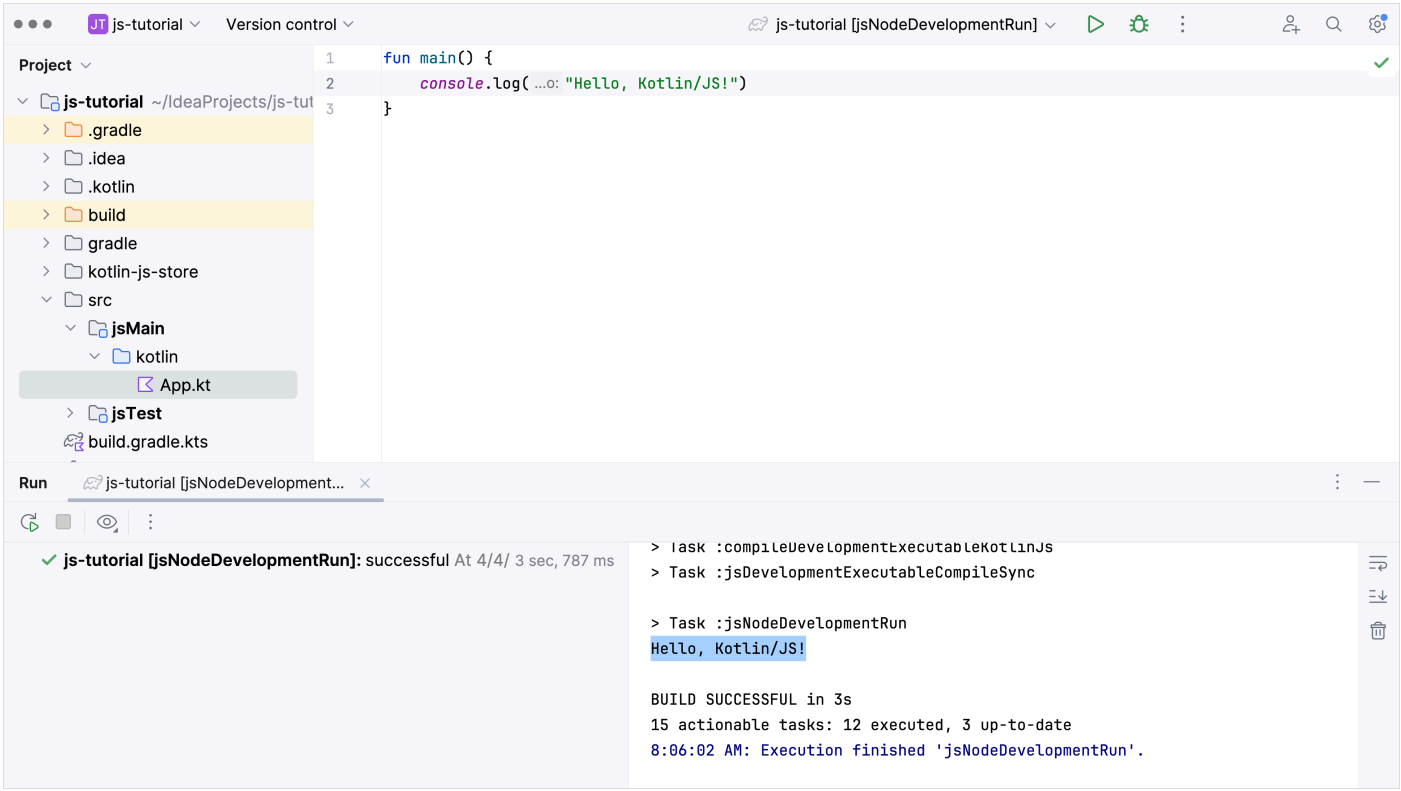
```
./gradlew jsNodeDevelopmentRun
```

If you're using IntelliJ IDEA, you can find the `jsNodeDevelopmentRun` action in the Gradle tool window:



Gradle Run task in IntelliJ IDEA

On first start, the `kotlin.multiplatform` Gradle plugin will download all required dependencies to get you up and running. After the build is completed, the program is executed, and you can see the logging output in the terminal:



Executing the JS target in a Kotlin Multiplatform project in IntelliJ IDEA

Run the browser target

When targeting the browser, your project is required to have an HTML page. This page will be served by the development server while you are working on your application, and should embed your compiled Kotlin/JS file. Create and fill an HTML file `/src/jsMain/resources/index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>JS Client</title>
</head>
<body>
<script src="js-tutorial.js"></script>
</body>
</html>
```

By default, the name of your project's generated artifact (which is created through webpack) that needs to be referenced is your project name (in this case, `js-tutorial`). If you've named your project `followAlong`, make sure to embed `followAlong.js` instead of `js-tutorial.js`

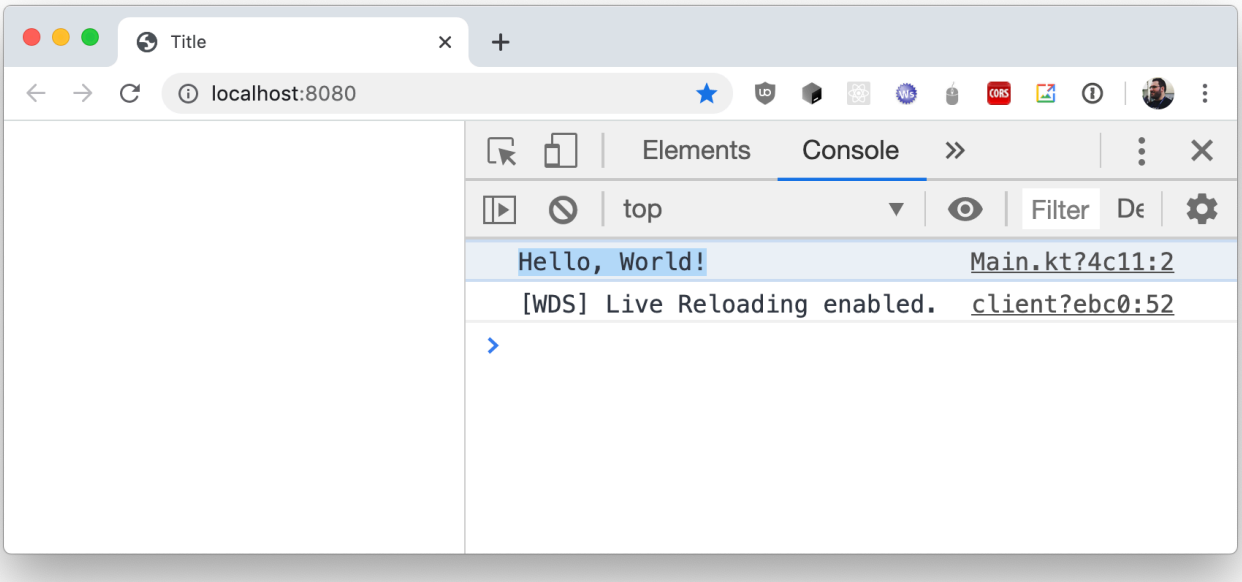
After making these adjustments, start the integrated development server. You can do this from the command line via the Gradle wrapper:

```
./gradlew jsBrowserDevelopmentRun
```

When working from IntelliJ IDEA, you can find the `jsBrowserDevelopmentRun` action in the Gradle tool window.

After the project has been built, the embedded `webpack-dev-server` will start running, and will open a (seemingly empty) browser window pointing to the HTML file you specified previously. To validate that your

program is running correctly, open the developer tools of your browser (for example, by right-clicking and choosing the *Inspect* action). Inside the developer tools, navigate to the console, where you can see the results of the executed JavaScript code:



Console output in browser developer tools

With this setup, you can recompile your project after each code change to see your changes. Kotlin/JS also supports a more convenient way of automatically rebuilding the application while you are developing it. To find out how to set up this *continuous mode*, check out the corresponding tutorial.

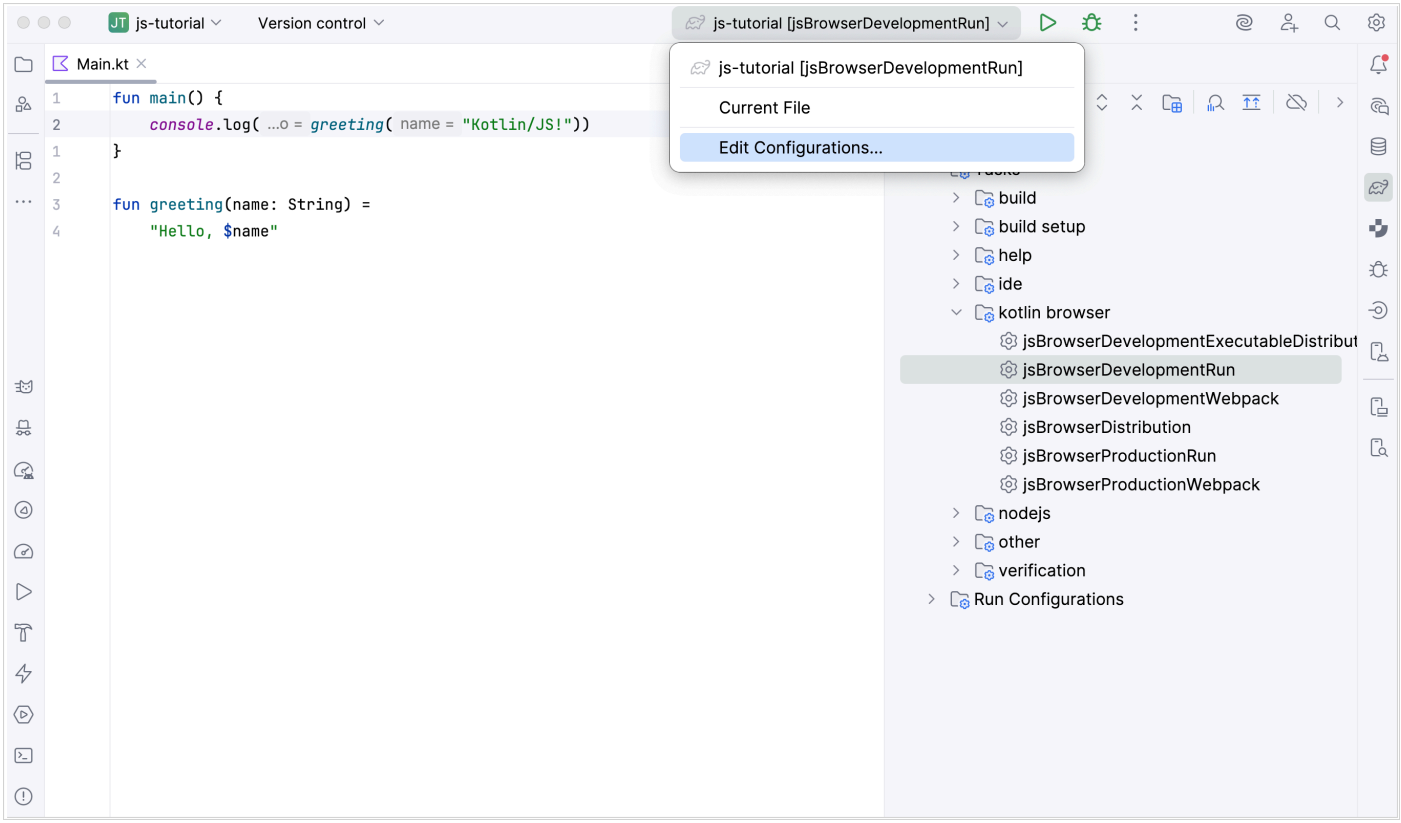
Development server and continuous compilation

Instead of manually compiling and running your Kotlin/JS project each time you want to see the changes you made, you can use the *continuous compilation* mode. Rather than using the regular `jsBrowserDevelopmentRun` (for `browser`) and `jsNodeDevelopmentRun` (for `nodejs`) command, invoke the Gradle wrapper in continuous mode:

```
# For `browser` project
./gradlew jsBrowserDevelopmentRun --continuous

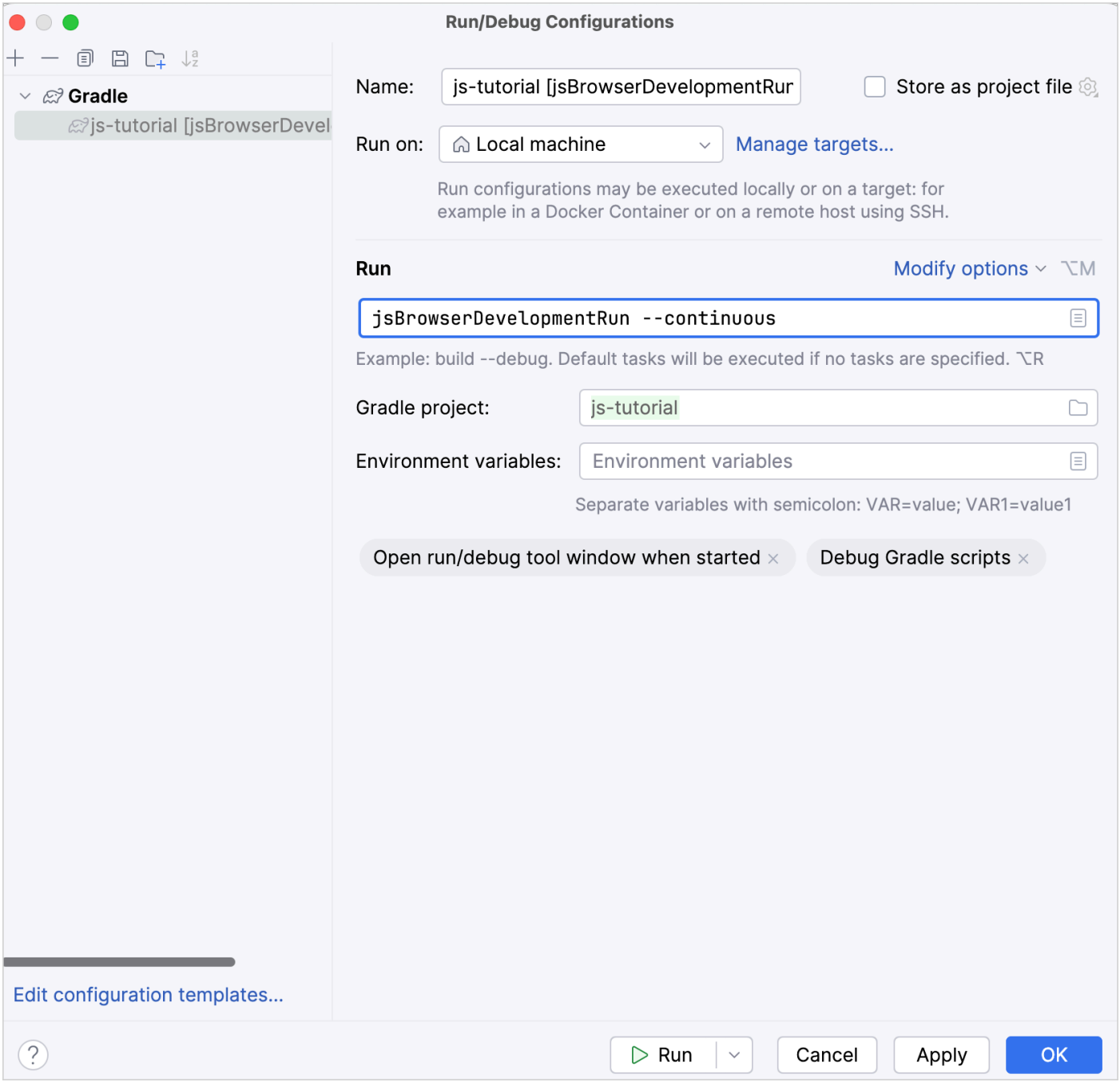
 # For `nodejs` project
./gradlew jsNodeDevelopmentRun --continuous
```

If you are working in IntelliJ IDEA, you can pass the same flag via the list of run configurations. After running the `jsBrowserDevelopmentRun` Gradle task for the first time from the IDE, IntelliJ IDEA automatically generates a run configuration for it, which you can edit at the top toolbar:



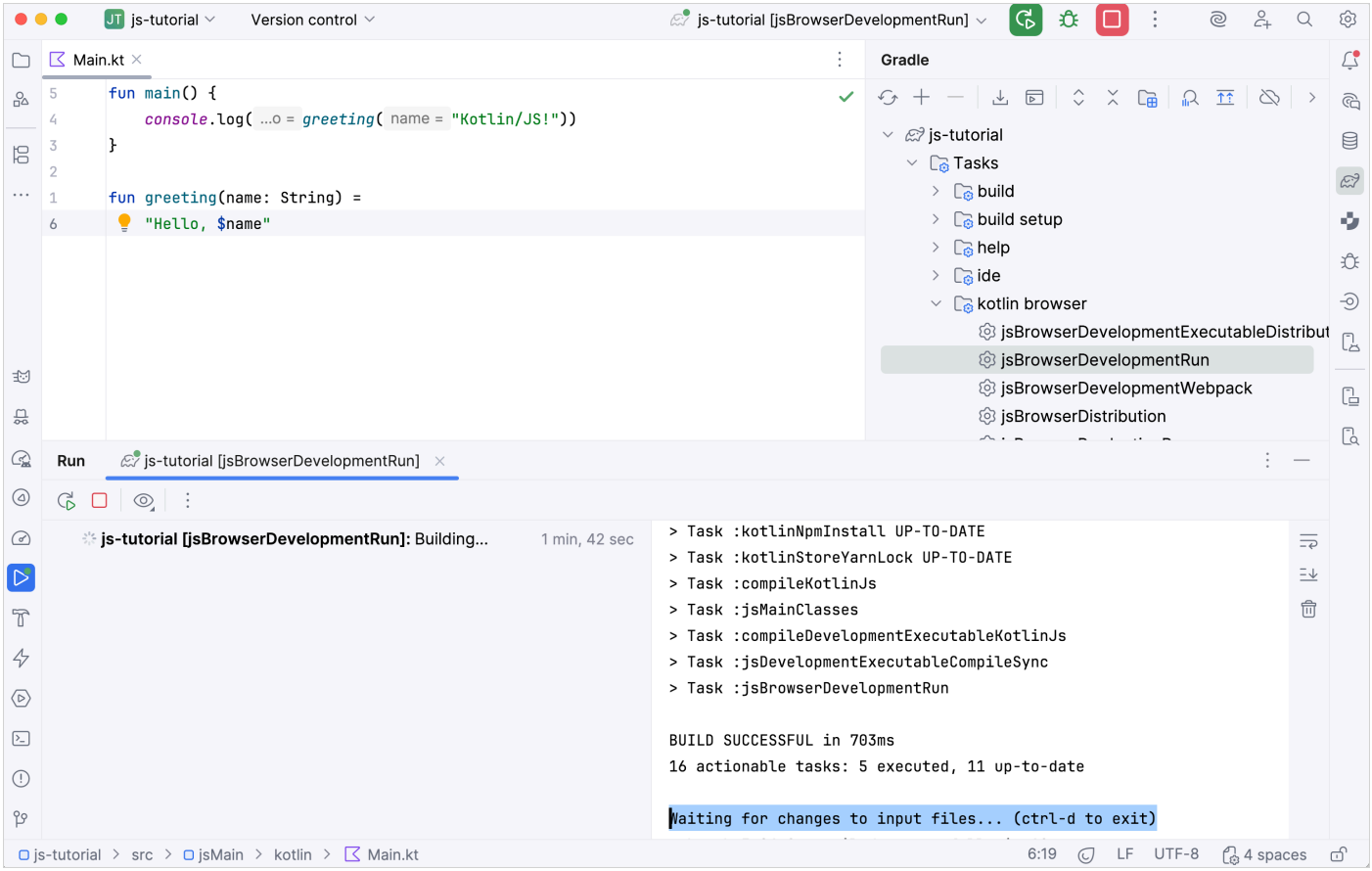
Editing run configurations in IntelliJ IDEA

Enable continuous mode via the **Run/Debug Configurations** dialog by adding the `--continuous` flag to the arguments for the run configuration:



Adding the continuous flag to a run configuration in IntelliJ IDEA

When executing this run configuration, you can note that the Gradle process continues watching for changes to the program:



Gradle waiting for changes

Once a change has been detected, the program is recompiled automatically. If you still have the web page open in the browser, the development server triggers an automatic reload of the page, and the changes become visible. This is thanks to the integrated `webpack-dev-server` that is managed by the Kotlin Multiplatform Gradle plugin.

Debug Kotlin/JS code

JavaScript source maps provide mappings between the minified code produced by bundlers or minifiers and the actual source code a developer works with. This way, the source maps enable support for debugging the code during its execution.

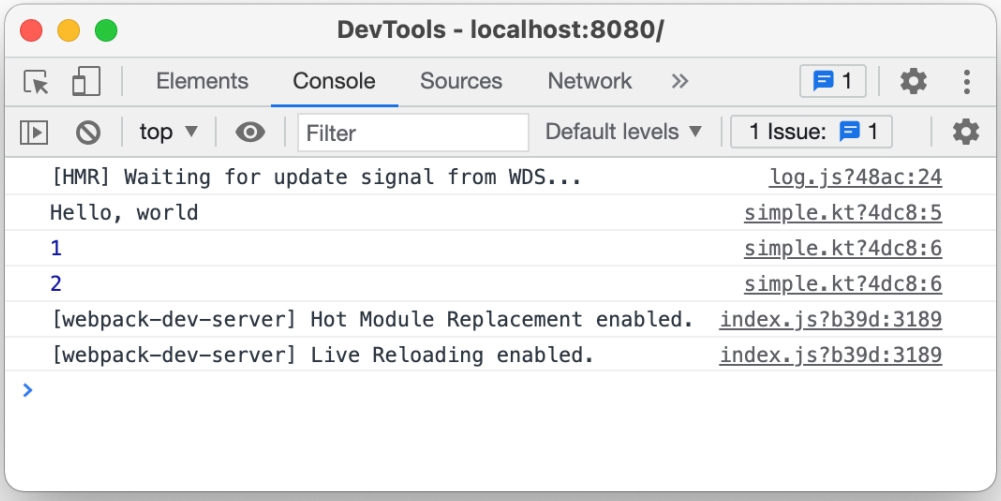
The Kotlin Multiplatform Gradle plugin automatically generates source maps for the project builds, making them available without any additional configuration.

Debug in browser

Most modern browsers provide tools that allow inspecting the page content and debugging the code that executes on it. Refer to your browser's documentation for more details.

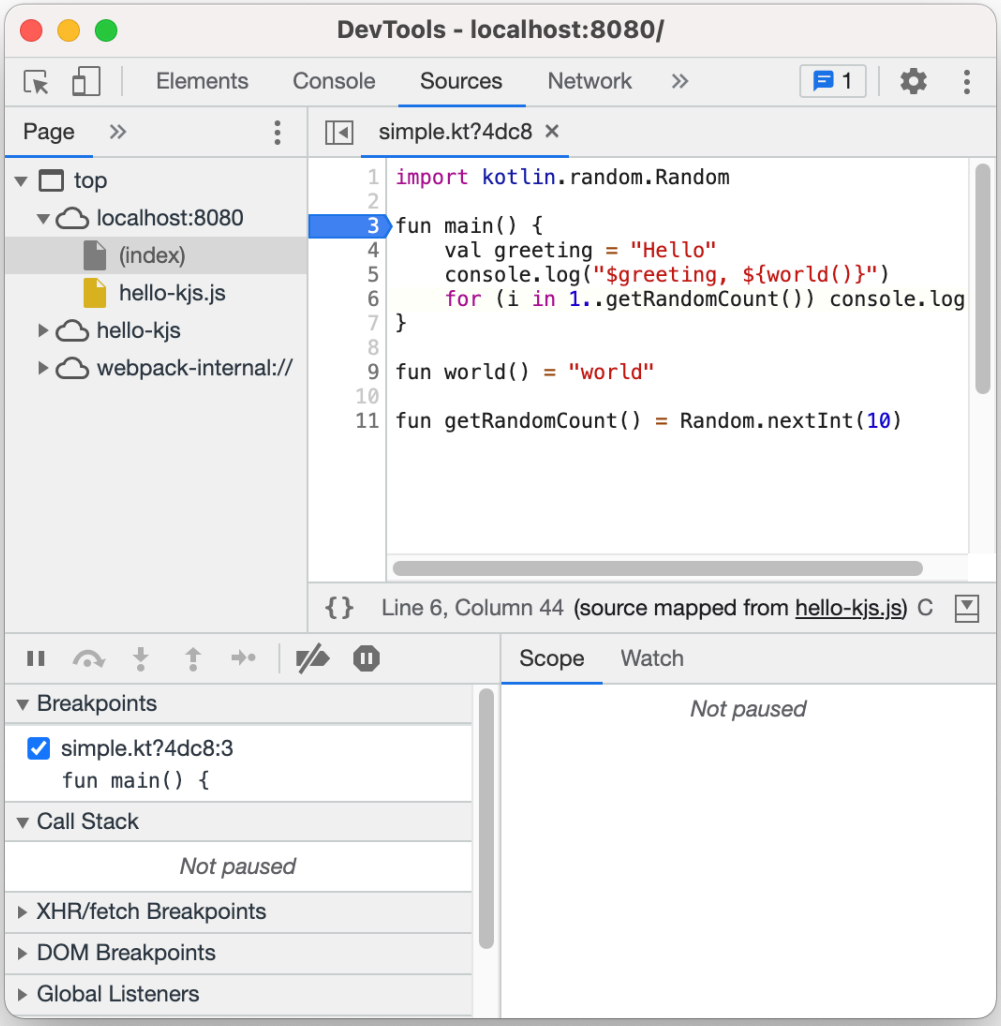
To debug Kotlin/JS in the browser:

1. Run the project by calling one of the available *run* Gradle tasks, for example, `jsBrowserDevelopmentRun`. Learn more about running Kotlin/JS.
2. Navigate to the page in the browser and launch its developer tools (for example, by right-clicking and selecting the **Inspect** action). Learn how to find the developer tools in popular browsers.
3. If your program is logging information to the console, navigate to the **Console** tab to see this output. Depending on your browser, these logs can reference the Kotlin source files and lines they come from:



Chrome DevTools console

4. Click the file reference on the right to navigate to the corresponding line of code. Alternatively, you can manually switch to the **Sources** tab and find the file you need in the file tree. Navigating to the Kotlin file shows you the regular Kotlin code (as opposed to minified JavaScript):



Debugging in Chrome DevTools

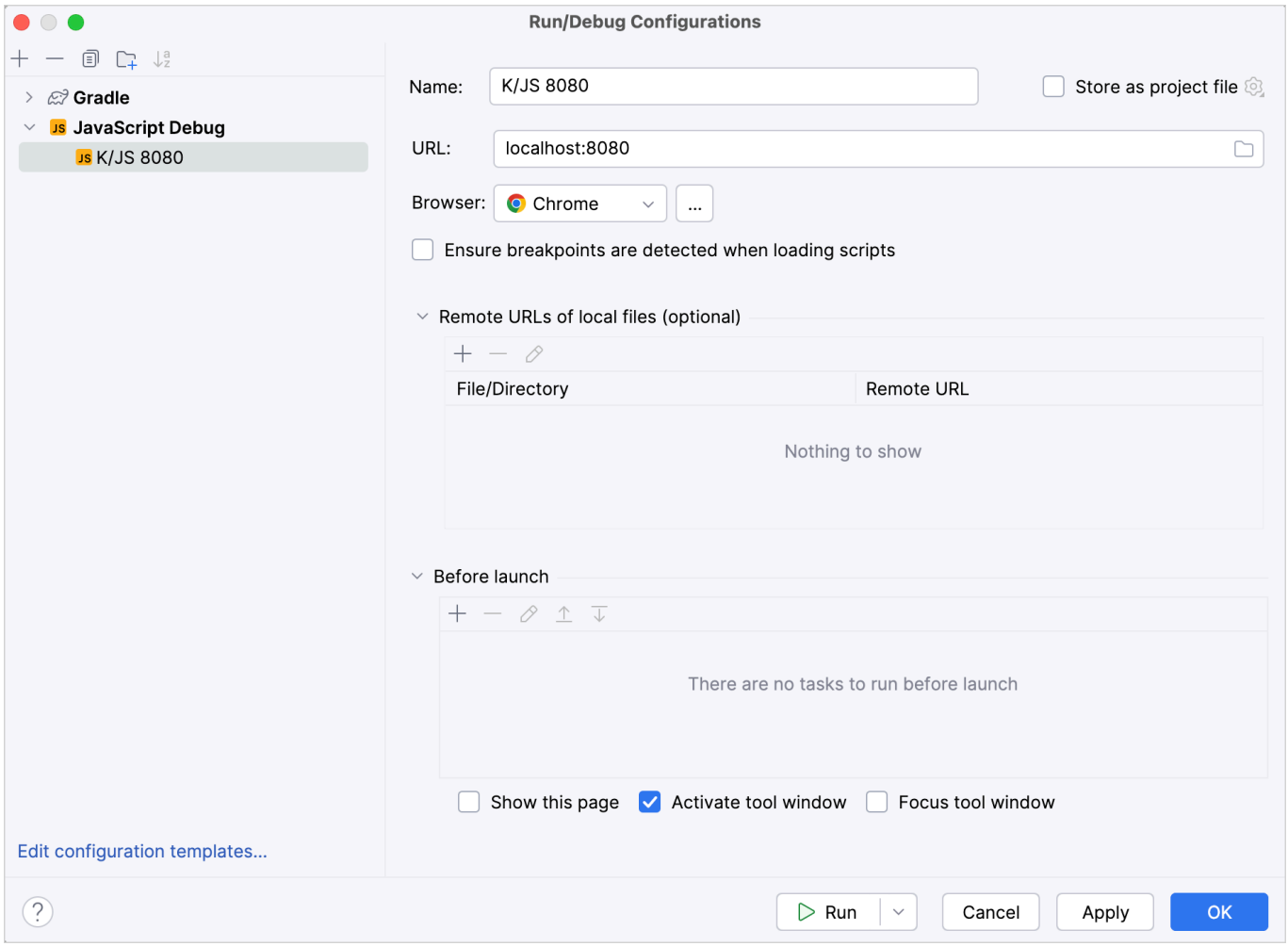
You can now start debugging the program. Set a breakpoint by clicking on one of the line numbers. The developer tools even support setting breakpoints within a statement. As with regular JavaScript code, any set breakpoints will persist across page reloads. This also makes it possible to debug Kotlin's `main()` method which is executed when the script is loaded for the first time.

Debug in the IDE

IntelliJ IDEA with an Ultimate subscription provides a powerful set of tools for debugging code during development.

For debugging Kotlin/JS in IntelliJ IDEA, you'll need a **JavaScript Debug** configuration. To add such a debug configuration:

1. Go to **Run | Edit Configurations**.
2. Click **+** and select **JavaScript Debug**.
3. Specify the configuration **Name** and provide the **URL** on which the project runs (`http://localhost:8080` by default).



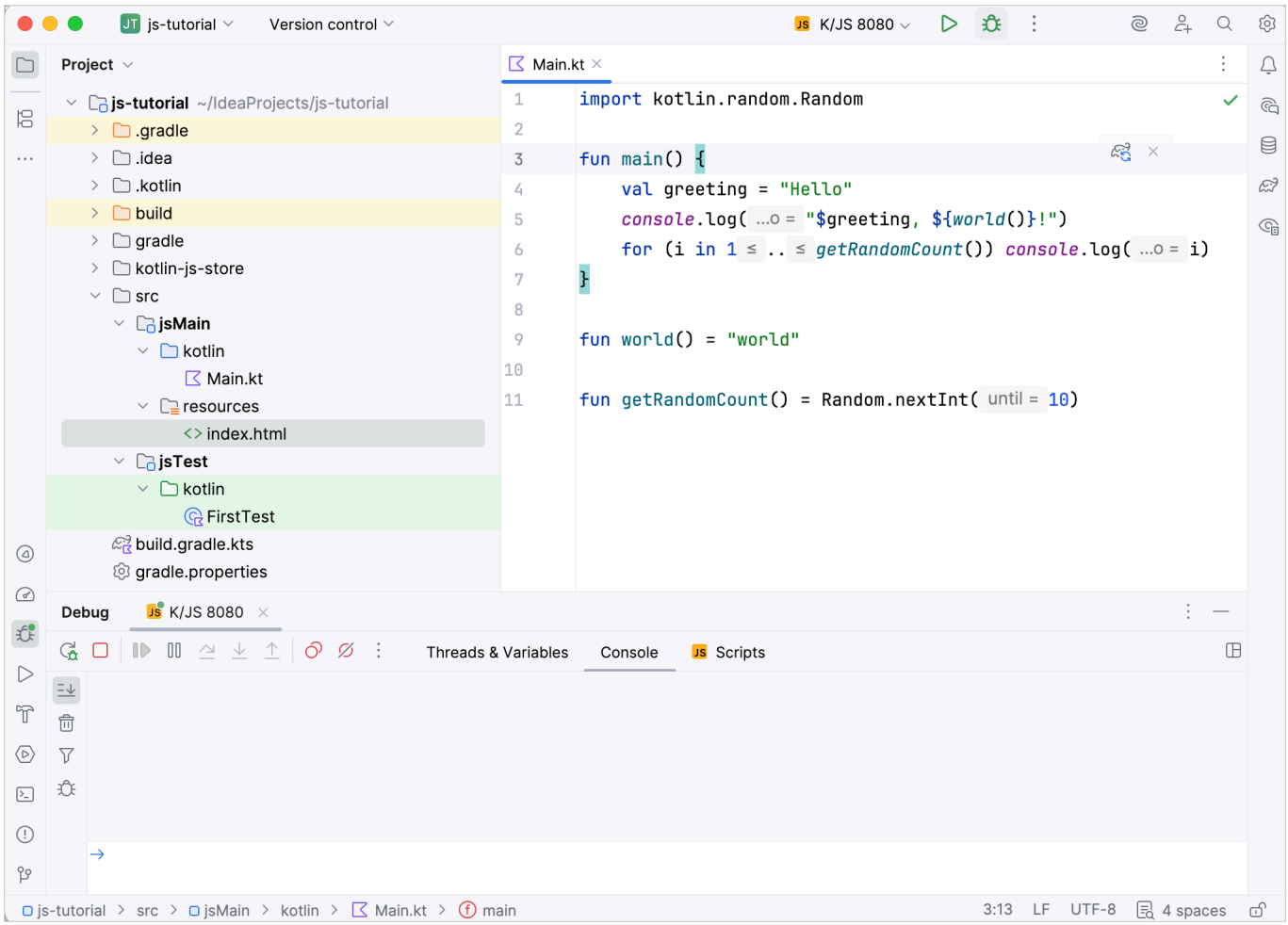
JavaScript debug configuration

4. Save the configuration.

Learn more about setting up JavaScript debug configurations.

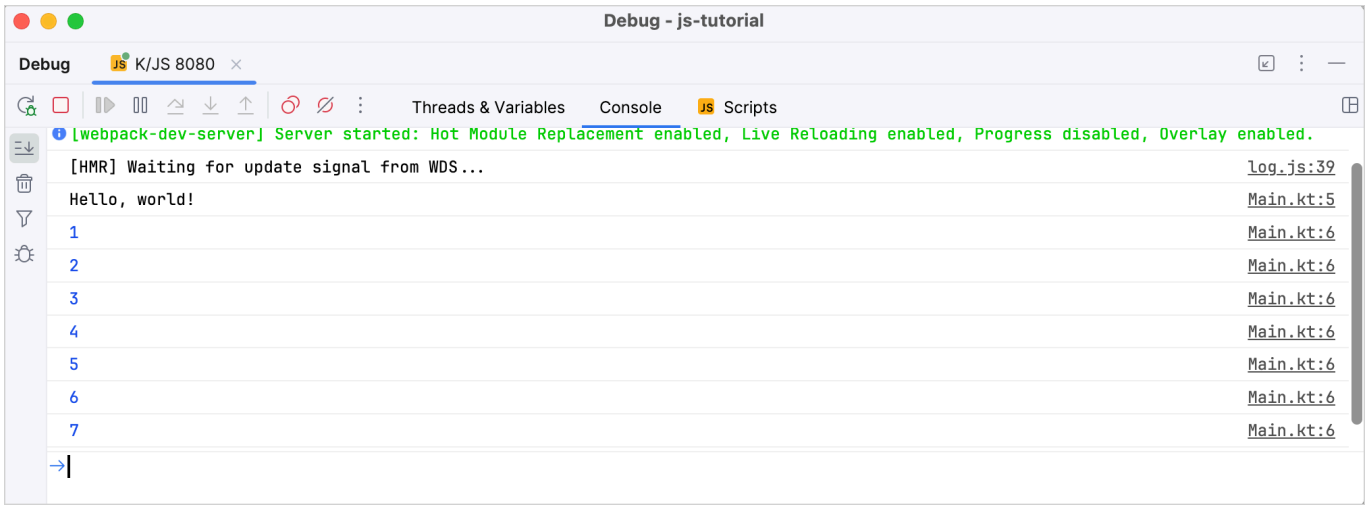
Now you're ready to debug your project!

1. Run the project by calling one of the available *run* Gradle tasks, for example, `jsBrowserDevelopmentRun`. Learn more about running Kotlin/JS.
2. Start the debugging session by running the JavaScript debug configuration you've created previously:



JavaScript debug configuration

3. You can see the console output of your program in the **Debug** window in IntelliJ IDEA. The output items reference the Kotlin source files and lines they come from:



JavaScript debug output in the IDE

4. Click the file reference on the right to navigate to the corresponding line of code.

You can now start debugging the program using the whole set of tools that the IDE offers: breakpoints, stepping, expression evaluation, and more. Learn more about debugging in IntelliJ IDEA.

Because of the limitations of the current JavaScript debugger in IntelliJ IDEA, you may need to rerun the JavaScript debug to make the execution stop on breakpoints.

Debug in Node.js

If your project targets Node.js, you can debug it in this runtime.

To debug a Kotlin/JS application targeting Node.js:

1. Build the project by running the `build` Gradle task.
2. Find the resulting `.js` file for Node.js in the `build/js/packages/your-module/kotlin/` directory inside your project's directory.
3. Debug it in Node.js as described in the Node.js Debugging Guide.

What's next?

Now that you know how to start debug sessions with your Kotlin/JS project, learn to make efficient use of the debugging tools:

- Learn how to debug JavaScript in Google Chrome
- Get familiar with IntelliJ IDEA JavaScript debugger
- Learn how to debug in Node.js.

If you run into any problems

If you face any issues with debugging Kotlin/JS, please report them to our issue tracker, YouTrack

Run tests in Kotlin/JS

The Kotlin Multiplatform Gradle plugin lets you run tests through a variety of test runners that can be specified via the Gradle configuration.

The general workflow for running tests in Kotlin/JS is to add test dependencies, configure the test task in the build file, add tests, and run them.

For browser testing, you can choose between:

- The Karma test runner.
- The new DSL for browser testing.

The Karma project has been deprecated. No new features or bug fixes are expected. As an alternative, try out the new Kotlin DSL for browser testing.

The new DSL for browser testing is currently Experimental. It may be changed at any time. Opt-in is required with the `@OptIn(ExperimentalJsTestDsl::class)` annotation.

Add test dependencies

When you create a multiplatform project, you can add test dependencies to all the source sets, including the JavaScript target, by using a single dependency in `commonTest`:

Kotlin

```
// build.gradle.kts
kotlin {
    sourceSets {
        commonTest.dependencies {
            implementation(kotlin("test")) // Enables test annotations and functionality in JS
        }
    }
}
```

Groovy

```
// build.gradle
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // Enables test annotations and functionality in JS
            }
        }
    }
}
```

Configure browsers

You can run tests in Kotlin/JS against specific browsers. To do so, adjust the settings in the `browser {}` configuration block of the Gradle build file.

By default, the plugin uses Headless Chrome to run browser tests. No browsers are bundled with the Kotlin Multiplatform Gradle plugin by default. To enable additional browsers, use the `testTask {}` block for Karma and the `test {}` block for the new DSL for browser testing. See all available options here:

Karma

```
kotlin {
    js {
        browser {
            testTask {
                useKarma {
                    useIe()
                    useSafari()
                    useFirefox()
                    useChrome()
                    useChromeCanary()
                    useChromeHeadless()
                    usePhantomJS()
                    useOpera()
                }
            }
        }
    }
}
```

With Karma, you need to install all necessary browsers on your target system (locally or in CI).

For more information on Karma functionality, see Set up a Kotlin/JS project.

DSL for browser testing

```
import org.jetbrains.kotlin.gradle.ExperimentalJsTestDsl

kotlin {
    js {
        browser {
            @OptIn(ExperimentalJsTestDsl::class)
            test {
                chromium()
                firefox()
                webkit() // Safari browser
            }
        }
    }
}
```

With the new DSL for browser testing, the Kotlin Multiplatform Gradle plugin installs the necessary browsers on the first run by using the `playwright install` command. Playwright then manages the location of these browsers and doesn't use locally installed browsers.

For additional settings available in the new DSL for browser testing, see Advanced configuration.

Add a test

To check that tests are executed properly, create a file `src/jsTest/kotlin/AppTest.kt` with the following content:

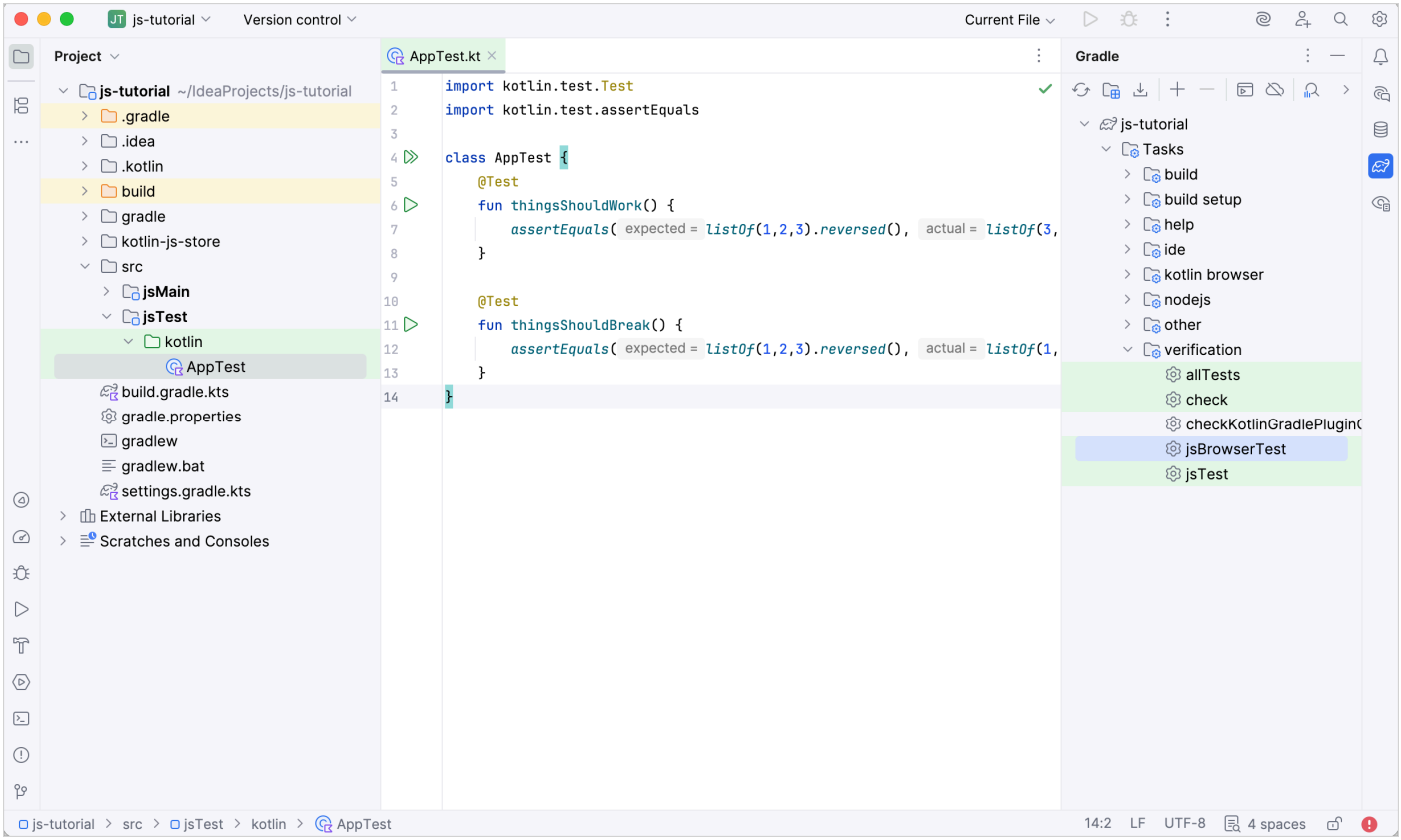
```
import kotlin.test.Test
import kotlin.test.assertEquals

@Test
fun thingsShouldWork() {
    assertEquals(listOf(3,2,1), listOf(1,2,3).reversed())
}

@Test
fun thingsShouldBreak() {
    assertEquals(listOf(1,2,3), listOf(1,2,3).reversed())
}
```

Run tests

To run the tests in the browser, execute the `jsBrowserTest` task or use the gutter icons in IntelliJ IDEA to execute all or individual tests:

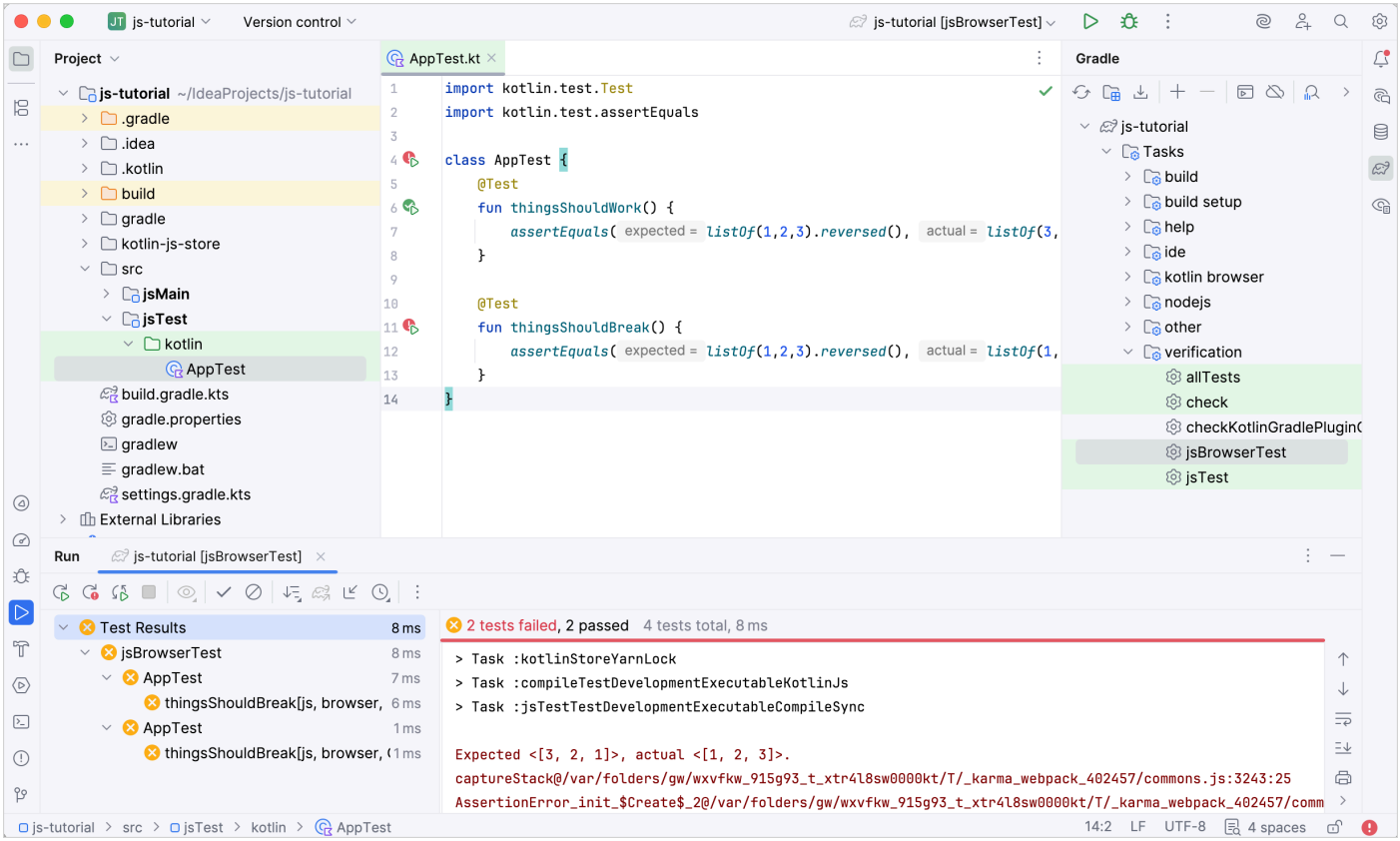


Gradle browserTest task

Alternatively, if you want to run the tests in the command line, use the Gradle wrapper:

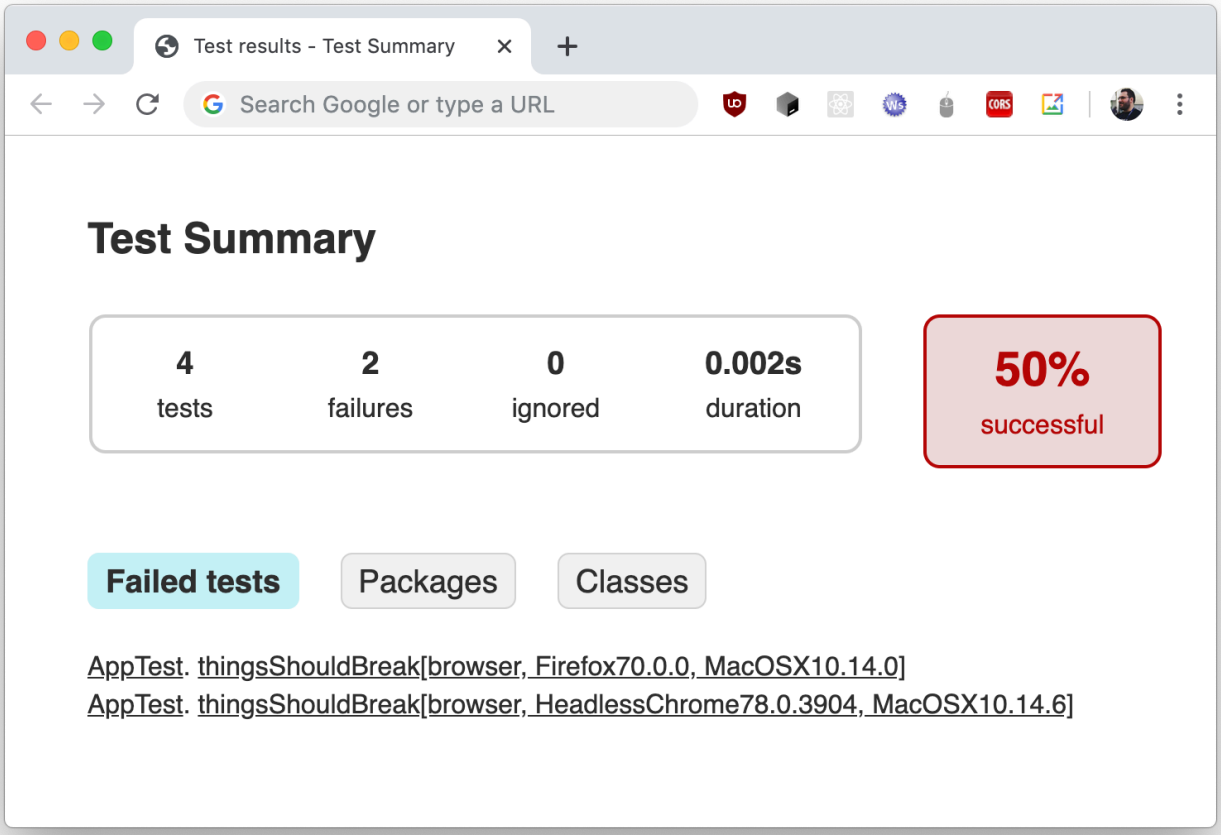
```
./gradlew jsBrowserTest
```

After running tests in IntelliJ IDEA, the **Run** tool window shows the test results. You can click failed tests to see their stack trace and navigate to the corresponding test implementation by double-clicking it.



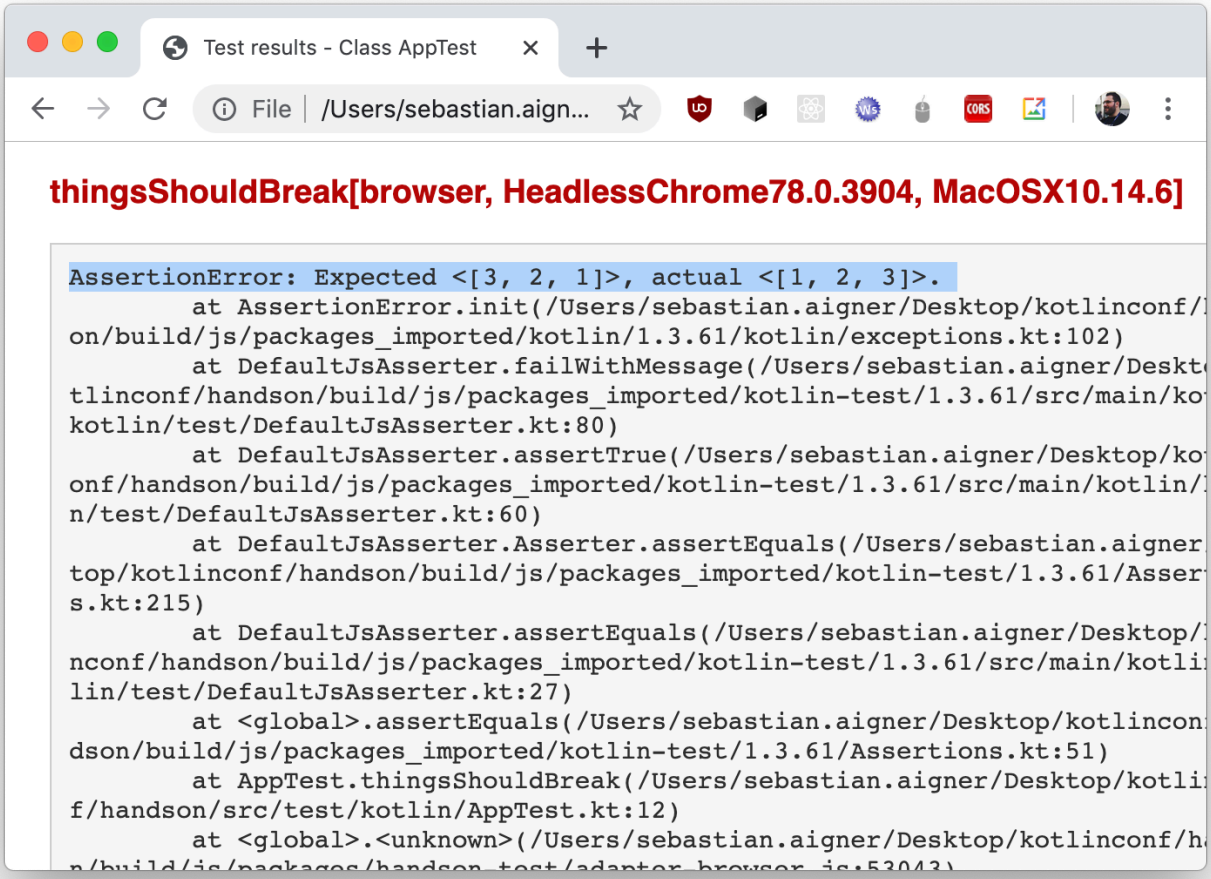
Test results in IntelliJ IDEA

After each test run, regardless of how you executed the test, you can find a properly formatted test report from Gradle in `build/reports/tests/jsBrowserTest/index.html`. Open this file in a browser to see another overview of the test results:



Gradle test summary

If you're using the set of example tests shown in the snippet above, one test passes, and one test fails, resulting in a 50% success rate. To get more information about individual test cases, use the provided links:



Stacktrace of a failed test in the Gradle summary

Experimental

Advanced configuration

This section applies only to the new experimental DSL for browser testing.

The new DSL for browser testing is designed to be minimalistic and tool-agnostic. The current implementation includes:

- Playwright acts as a browser driver and distribution manager that supports the Chromium, Firefox, and WebKit (Safari) browser engines.
- Mocha acts as a test runner.
- webpack acts as a bundler (will be replaced with Vite in future releases).

The DSL exposes timeouts, headless mode, and per-runner options as Gradle properties, so you can share defaults between runners, override them for a specific browser, and compute values lazily with providers:

```
import org.jetbrains.kotlin.gradle.ExperimentalJsTestDsl
import kotlin.time.Duration.Companion.seconds

kotlin {
    js {
        browser {
            @OptIn(ExperimentalJsTestDsl::class)
            test {
                // Configures the default timeout for all runners with kotlin.Duration
                timeout = 30.seconds

                // Configures headless mode using Gradle providers
                headless = providers
                    .environmentVariable("IS_IN_CI")
                    .map { it.toBoolean() }
                    .orElse(false)

                // Enables and configures the Chromium runner with a custom name
                chromium("chromium-no-webgl2") {
                    // Overrides the default timeout for this runner
                    timeout = 10.seconds

                    // Chromium-specific extra launch argument
                    launchArgs.add("--disable-webgl2")
                }

                // Enables the Firefox runner
                firefox()

                // Enables and configures the WebKit runner
                webkit("safari") {
                    timeout = 35.seconds
                }
            }
        }
    }
}
```

You can set the options for all test runners directly in the `test {}` block. To override these common options for a particular runner, use a custom name for it and provide different values inside a runner block. In this example, Chromium and WebKit (Safari) browsers use timeouts of 10 and 35 seconds, respectively, while Firefox uses the common timeout of 30 seconds.

Each runner is registered under its own name, so the test report tells you which browser a particular result comes from.

Experimental

Configuration for plugin authors

This section applies only to the new experimental DSL for browser testing.

If you write a Gradle plugin on top of the Kotlin Multiplatform Gradle plugin, the new DSL for browser testing also gives you access to the browser runners and to the location of the generated test bundle.

Kotlin generates the test bundle for running browser tests using the default test runner page. You can replace it by pointing to a different location in the `testsLocation` property:

```
kotlin {
    js {
        browser {
            @OptIn(ExperimentalJsTestDsl::class)
            test {
                // Implement customJsTestsLocation to modify or replace the default JS test bundle
                @OptIn(DelicateKotlinGradlePluginApi::class)
                testsLocation = customJsTestsLocation(extendFrom = defaultTestsLocationProvider)

                chromium()
            }
        }
    }
}
```

Your custom test bundler can include your own development server, bundler, or test runner. The `defaultTestsLocationProvider` property gives you access to the default location, so you can build on top of it instead of implementing everything from scratch.

Each test location exposes the directory with the generated test bundle (`bundleLocation`), the name of the test page (`testHtmlFileName`), and the URL that the browser opens (`url`) through the `KotlinJsTestsLocation` interface.

With access to these APIs, you can:

- Customize the URL that the browser opens. Each browser runner has its own test location, so you can override it either for all runners in the `test {}` block or for a particular runner.
- Override the bundle location itself, for example, to add extra files to the bundle.
- Post-process the generated test bundle. Register your own task and modify the files there before the browser opens them, for example, to inject your own configuration into `test.html` .

Keep the following limitations in mind when you build plugins using these APIs:

- Configuring `subtarget.test` enables the new test pipeline and disables Karma. There is currently no reliable way to detect which pipeline the user has chosen.
- There is no reliable way to configure a specific browser runner lazily, so the configuration has to happen in `afterEvaluate` . Consider asking your users to set the test location explicitly or exposing decorating functions such as `myPluginChromium()` instead.

Leave feedback

The new DSL for browser testing is in active development. New features, for example debugging, are planned for the next Kotlin releases.

We would appreciate your feedback in YouTrack or in the #javascript Slack channel.

Kotlin/JS frameworks

Take advantage of available Kotlin/JavaScript frameworks that simplify web development. These frameworks provide ready-to-use components, routing, state management, and other tools for building modern web applications.

Here are some Kotlin/JS web frameworks from the community:

Kobweb

Kobweb is a Kotlin framework for creating websites and web applications with Compose HTML. It supports live-reloading for fast development. Inspired by Next.js, Kobweb promotes a standard structure for adding widgets, layouts, and pages.

Out of the box, Kobweb provides page routing, light/dark mode, CSS styling, Markdown support, backend APIs, and more. It also includes Silk, a UI library with a set of versatile widgets for modern UIs.

Kobweb also supports site export by generating page snapshots for SEO and automatic search indexing. Additionally, it enables the creation of DOM-based UIs that efficiently update in response to state changes.

For more information and examples, see the Kobweb documentation.

For updates and discussions about the framework, join the #kobweb and #compose-web channels in Kotlin Slack.

Kilua

Kilua is a composable web framework built on the Compose Runtime and is similar to the compose-html library. Unlike compose-html, Kilua supports both Kotlin/Wasm and Kotlin/JS targets.

Kilua provides a modular API for creating declarative UI components and managing their state. It also includes a set of ready-to-use components for common web application use cases.

Kilua is a successor to the KVision framework. Kilua is designed to be familiar to both Compose users (`@Composable` functions, state management, coroutines/flow integration) and KVision users (a component-based API that allows some imperative interaction with UI components).

For more information and examples, see the Kilua documentation.

For updates and discussions about the framework, join the #kilua channel in Kotlin Slack.

Summon

Summon is a Kotlin Multiplatform frontend framework for building high-performance applications across JVM, Kotlin/JS, and Kotlin/Wasm. It supports component-based UI development, first-class server-side rendering (SSR) with HTML streaming, and modifier-driven styling that compiles to clean CSS.

Summon also focuses on runtime-aware components for accessibility and advanced behavior, while keeping a unified codebase across platforms.

For more information and examples, see the Summon documentation.

Kinetica

Kinetica is a Kotlin UI framework focused on practical outcomes for web apps: fast initial page delivery through server rendering, SEO-friendly HTML output, and a smooth handoff to interactive client-side UI after the page loads. It also supports headless testing, which helps teams test UI behavior quickly and reliably without depending on full browser environments.

Out of the box, Kinetica includes routing, forms, and persistence modules, so teams can ship common app features faster. Its update model applies minimal DOM patches, which helps keep interactions responsive as state changes, and its consistent reactive model is designed to reduce unexpected behavior in larger apps.

For more information and examples, see the Kinetica documentation.

Kotlin React

React is a component-based library, widely used for web and native user interfaces. It offers a large ecosystem of components, learning materials, and an active community.

Kotlin React is a Kotlin wrapper for React that combines the React ecosystem with the type-safety and expressiveness of Kotlin.

For updates and discussions about the library, join the #react and #javascript channels in Kotlin Slack.

KVision

KVision is an object-oriented web framework for building Kotlin/JS applications with ready-to-use UI components. These components can be building blocks for your application's user interface.

With this framework, you can use both reactive and imperative programming models to build your frontend. You can also integrate it with your backend applications by using connectors for Ktor, Spring Boot, and other frameworks. In addition, you can share code using Kotlin Multiplatform.

For more information, tutorials, and examples, see the KVision documentation.

For updates and discussions about the framework, join the #kvision and #javascript channels in Kotlin Slack.

fritz2

fritz2 is a standalone framework for building reactive web user interfaces. It provides its own type-safe DSL for building and rendering HTML elements and uses Kotlin's coroutines and flows to define components and their data bindings.

Out of the box, fritz2 offers state management, validation, routing, and more. It also integrates with Kotlin Multiplatform projects.

For more information, tutorials, and examples, see the fritz2 documentation.

For updates and discussions about the framework, join the #fritz2 and #javascript channels in Kotlin Slack.

Doodle

Doodle is a vector-based UI framework for Kotlin/JS. Doodle applications use the browser's graphics capabilities to draw user interfaces instead of relying on DOM, CSS, or JavaScript. This approach gives you control over the rendering of arbitrary UI elements, vector shapes, gradients, and custom visualizations.

For more information, tutorials, and examples, see the Doodle documentation.

For updates and discussions about the framework, join the #doodle and #javascript channels in Kotlin Slack.

Kotlin/JS compiler features

Kotlin/JS includes compiler features to optimize code for performance, size, and development speed. This works through the compilation process, which transforms Kotlin code into an intermediate representation (IR) before generating the JavaScript code.

Lazy initialization of top-level properties

For better application startup performance, the Kotlin/JS compiler initializes top-level properties lazily. This way, the application loads without initializing all the top-level properties used in its code. It initializes only the ones needed at startup; other properties receive their values later when the code that uses them actually runs.

```
val a = run {
    val result = // intensive computations
    println(result)
    result
} // value is computed upon the first usage
```

If for some reason you need to initialize a property eagerly (upon the application start), mark it with the `@EagerInitialization` annotation.

Incremental compilation for development binaries

The Kotlin/JS compiler provides the *incremental compilation mode for development binaries* that speeds up the development process. In this mode, the compiler caches the results of `compileDevelopmentExecutableKotlinJs` Gradle task on the module level. It uses the cached compilation results for unchanged source files during subsequent compilations, making them complete faster, especially with small changes.

Incremental compilation is enabled by default. To disable incremental compilation for development binaries, add the following line to the project's `gradle.properties` or `local.properties`:

```
kotlin.incremental.js.ir=false // true by default
```

The clean build in the incremental compilation mode is usually slower because of the need to create and populate the caches.

Minification of member names in production

The Kotlin/JS compiler uses its internal information about the relationships of your Kotlin classes and functions to apply more efficient minification, shortening the names of functions, properties, and classes. This reduces the size of resulting bundled applications.

This type of minification is automatically applied when you build your Kotlin/JS application in production mode, and enabled by default. To disable member name minification, use the `-Xir-minimized-member-names` compiler option:

```
kotlin {
    js {
        compilations.all {
            compileTaskProvider.configure {
                compilerOptions.freeCompilerArgs.add("-Xir-minimized-member-names=false")
            }
        }
    }
}
```

Dead code elimination

Dead code elimination (DCE) reduces the size of the resulting JavaScript code by removing unused properties, functions, and classes.

Unused declarations can appear in cases like:

- A function is inlined and never gets called directly (which always happens except for a few cases).
- A module uses a shared library. Without DCE, parts of the library that you don't use are still included in the resulting bundle. For example, the Kotlin standard library contains functions for manipulating lists, arrays, char sequences, adapters for DOM, and so on. All of this functionality would require about 1.3 MB as a JavaScript file. A simple "Hello, world" application only requires console routines, which are only a few kilobytes for the entire file.

In the Kotlin/JS compiler, DCE is handled automatically:

- DCE is disabled in *development* bundling tasks, which correspond to the following Gradle tasks:
  - `jsBrowserDevelopmentRun`
  - `jsBrowserDevelopmentWebpack`
  - `jsNodeDevelopmentRun`
  - `compileDevelopmentExecutableKotlinJs`
  - `compileDevelopmentLibraryKotlinJs`
  - Other Gradle tasks including "development" in their name
- DCE is enabled if you build a *production* bundle, which corresponds to the following Gradle tasks:
  - `jsBrowserProductionRun`
  - `jsBrowserProductionWebpack`
  - `compileProductionExecutableKotlinJs`
  - `compileProductionLibraryKotlinJs`
  - Other Gradle tasks including "production" in their name

With the `@JsExport` annotation, you can specify the declarations you want DCE to treat as roots.

Experimental

JS plain objects compiler plugin

The JavaScript (JS) plain objects compiler plugin (`js-plain-objects`) lets you create and copy plain JS objects in a type-safe way.

Here you can find information about plain JS objects and how to use the `js-plain-objects` compiler plugin for your Kotlin/JS projects.



The `js-plain-objects` plugin only works with the new K2 Kotlin compiler.

Plain JS objects

A plain object is a simple JS object created via an object literal (`{}`) that contains data properties. Many JS APIs accept/return Plain JS objects for configuration or data exchange.

With the `js-plain-objects` plugin, you declare a Kotlin external interface to describe the object shape and annotate it with `@JsPlainObject`. The compiler then generates convenient functions to build and copy such objects while preserving Kotlin type safety.

Enable the plugin

Add the `js-plain-objects` plugin to your project's Gradle configuration file, as the following Kotlin DSL shows:

Kotlin

```
// build.gradle.kts
plugins {
    kotlin("multiplatform") version "2.4.20"
    kotlin("plugin.js-plain-objects") version "2.4.20"
}

kotlin {
    js {
        browser() // or nodejs()
    }
}
```

Groovy

```
// build.gradle
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.4.20'
    id 'org.jetbrains.kotlin.plugin.js-plain-objects' version '2.4.20'
}

kotlin {
    js {
        browser() // or nodejs()
    }
}
```

Declare a plain object type

Once you have enabled the `js-plain-objects` plugin, you can declare a plain object type. Annotate an external interface with `@JsPlainObject`. For example:

```
@JsPlainObject
external interface User {
    val name: String
    val age: Int
    // You can use nullable types to declare a property as optional
    val email: String?
}
```

When the plugin processes such an interface, it generates a companion object with two helper functions for creating and copying objects:

```
@JsPlainObject
external interface User {
    val name: String
    val age: Int
    val email: String?

    // Generated by the plugin
    @JsExport.Ignore
    companion object {
        inline operator fun invoke(name: String, age: Int, email: String? = NOTHING): User =
            js("({ name: name, age: age, email: email })")

        inline fun copy(source: User, name: String = NOTHING, age: Int = NOTHING, email: String? =
NOTHING): User =
            js("Object.assign({}, source, { name: name, age: age, email: email })")
    }
}
```

From the previous example:

- `name` and `age` are declared without a nullability mark, so they are required.
- `email` is declared as nullable, so it's optional and can be skipped during creation.
- The operator `invoke` builds a new plain JS object with the provided properties.
- The `copy` function creates a new object by shallow-copying `source` and overriding any specified properties.
- The companion is marked with `@JsExport.Ignore` to avoid leaking these helpers into JS exports.

Use plain objects

Create and copy objects using the generated helpers:

```
fun main() {
    val user = User(name = "Name", age = 10)
    val copy = User.copy(user, age = 11, email = "some@user.com")

    println(JSON.stringify(user))
    // { "name": "Name", "age": 10 }
    println(JSON.stringify(copy))
    // { "name": "Name", "age": 11, "email": "some@user.com" }
}
```

The Kotlin code compiles to JavaScript:

```
function main () {
    var user = { name: "Name", age: 10 };
    var copy = Object.assign({}, user, { age: 11, email: "some@user.com" });

    println(JSON.stringify(user));
    // { "name": "Name", "age": 10 }
    println(JSON.stringify(copy));
    // { "name": "Name", "age": 11, "email": "some@user.com" }
}
```

Any JavaScript objects created with this approach are safe. When you use a wrong property name or value type, you encounter compile-time errors. This approach is also zero-cost since the generated code is inlined as a simple object literal and `Object.assign` calls.

What's next

Learn more about interoperability with JavaScript in the Use JavaScript code from Kotlin and dynamic type documentation.

Beta

Kotlin/Wasm

Kotlin/Wasm has the power to compile your Kotlin code into WebAssembly (Wasm) format. With Kotlin/Wasm, you can create applications that run on different environments and devices, which support Wasm and meet Kotlin's requirements.

Wasm is a binary instruction format for a stack-based virtual machine. This format is platform-independent because it runs on its own virtual machine. Wasm provides Kotlin and other languages with a compilation target.

You can use Kotlin/Wasm in different target environments, such as browsers, for developing web applications built with Compose Multiplatform, or outside the browser in standalone Wasm virtual machines. In the outside-of-browser case, WebAssembly System Interface (WASI) provides access to platform APIs, which you can also utilize.

> To run applications built with Kotlin/Wasm in a browser, your users need a browser version that supports WebAssembly's garbage collection and legacy exception handling proposals. To check the browser support status, see the WebAssembly roadmap.

Kotlin/Wasm and Compose Multiplatform

With Kotlin, you have the power to build applications and reuse mobile and desktop user interfaces (UIs) in your web projects through Compose Multiplatform and Kotlin/Wasm.

Compose Multiplatform is a declarative framework based on Kotlin and Jetpack Compose that allows you to implement the UI once and share it across all the platforms you target.

For web platforms, Compose Multiplatform uses Kotlin/Wasm as its compilation target. Applications built with Kotlin/Wasm and Compose Multiplatform use a `wasm-js` target and run in browsers.

Additionally, you can use the most popular Kotlin libraries in Kotlin/Wasm out of the box. Like in other Kotlin and Multiplatform projects, you can include dependency declarations in the build script. For more information, see Adding dependencies on multiplatform libraries.

Would you like to try it yourself?

Get started with Kotlin/Wasm

Kotlin/Wasm and WASI

Kotlin/Wasm uses the WebAssembly System Interface (WASI) for backend applications. Applications built with Kotlin/Wasm and WASI use a Wasm-WASI target, allowing you to call the WASI API and run applications outside the browser environment.

Kotlin/Wasm leverages WASI to abstract away platform-specific details, allowing the same Kotlin code to run across diverse platforms. This expands the reach of Kotlin/Wasm beyond web applications without requiring custom handling for each runtime.

WASI provides a secure, standardized interface for running Kotlin applications compiled to WebAssembly across different environments.

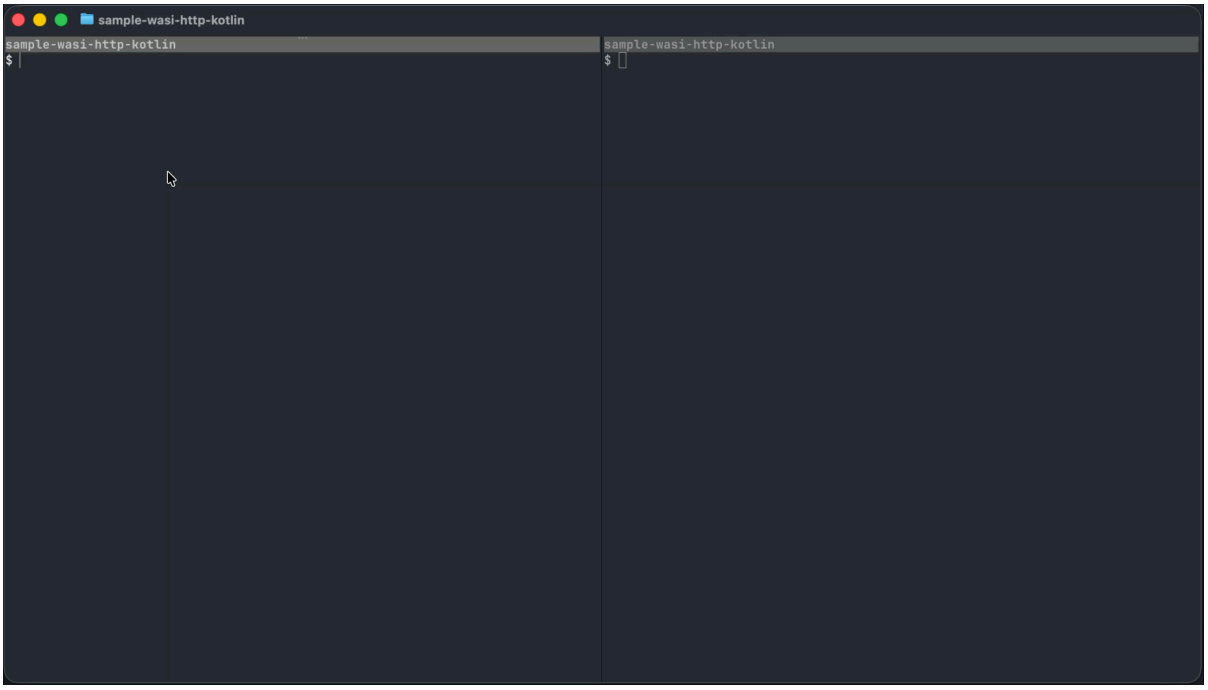
> To see Kotlin/Wasm and WASI in action, check the Get started with Kotlin/Wasm and WASI tutorial.

Experimental

WebAssembly Component Model

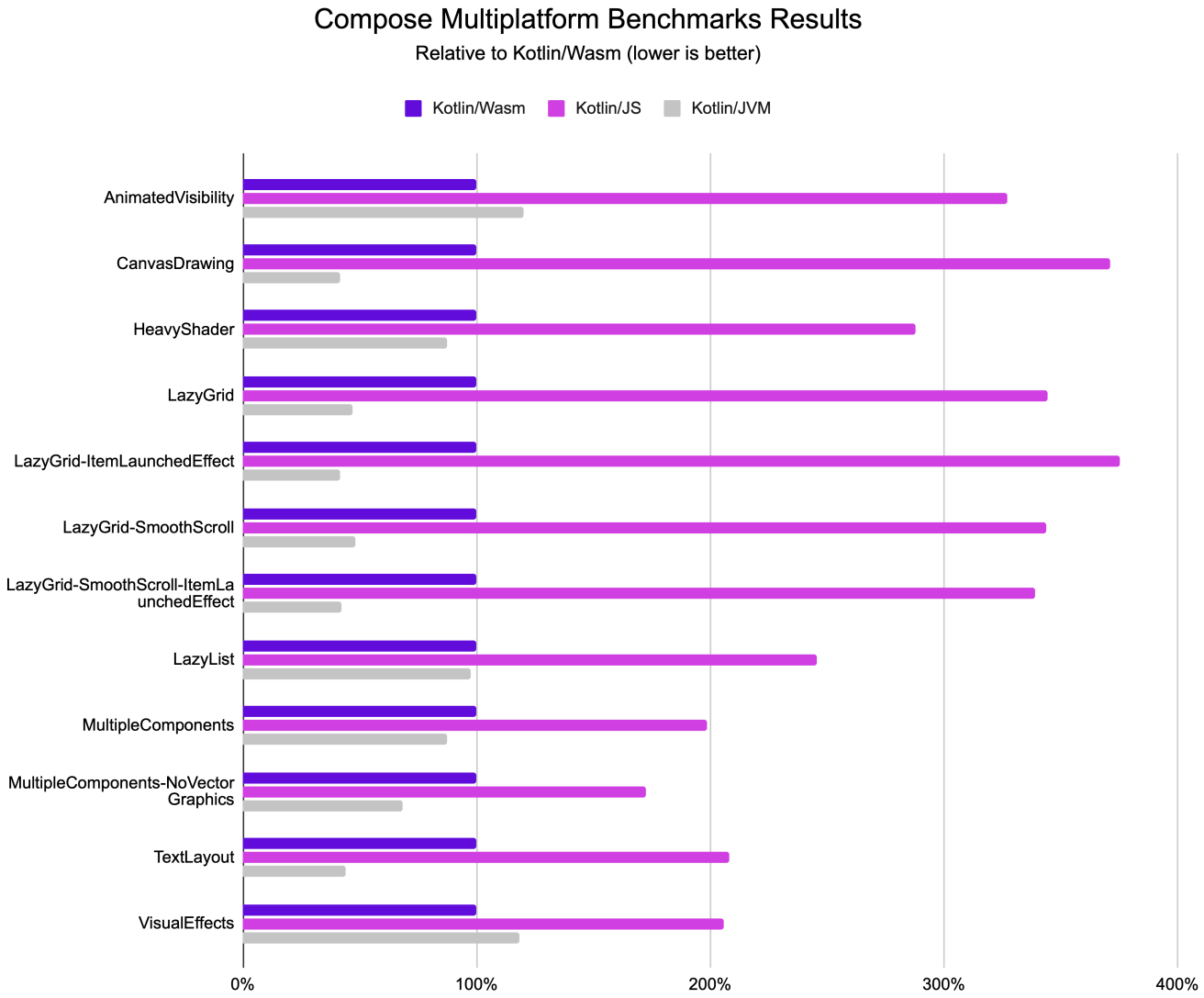
WASI 0.2 is built on the WebAssembly Component Model, which defines a way to build components from Wasm modules using standardized interfaces and types. The model lets you define language-agnostic components in applications or libraries. You can also compose Wasm modules and existing components into new components.

To explore what's possible with the WebAssembly Component Model and Kotlin/Wasm, check out this demo of a simple server built with `wasi:http`.



Kotlin/Wasm performance

Although Kotlin/Wasm is still in Beta, Compose Multiplatform running on Kotlin/Wasm already shows encouraging performance traits. You can see that its execution speed outperforms JavaScript and is approaching that of the JVM:



Kotlin/Wasm performance

We regularly run benchmarks on Kotlin/Wasm, and these results come from our testing in a recent version of Google Chrome.

Browser API support

The Kotlin/Wasm standard library provides declarations for browser APIs, including the DOM API. With these declarations, you can directly use the Kotlin API to access and utilize various browser functionalities. For example, in your Kotlin/Wasm applications, you can use manipulation with DOM elements or fetch the API without defining these declarations from scratch. To learn more, see our Kotlin/Wasm browser example.

The declarations for browser API support are defined using JavaScript interoperability capabilities. You can use the same capabilities to define your own declarations. In addition, Kotlin/Wasm–JavaScript interoperability allows you to use Kotlin code from JavaScript. For more information, see Use Kotlin code in JavaScript.

Leave feedback

Kotlin/Wasm feedback

- Slack: Get a Slack invite and provide your feedback directly to developers in our #webassembly channel.
- Report any issues in YouTrack.

Compose Multiplatform feedback

- Slack: provide your feedback in the #compose-web public channel.
- Report any issues in GitHub.

Learn more

- Learn more about Kotlin/Wasm in this YouTube playlist.
- Explore the Kotlin/Wasm examples in our GitHub repository.

Beta

Get started with Kotlin/Wasm and Compose Multiplatform

This tutorial demonstrates how to run a Compose Multiplatform app with Kotlin/Wasm in IntelliJ IDEA and generate artifacts you can publish as a website.

Create a project

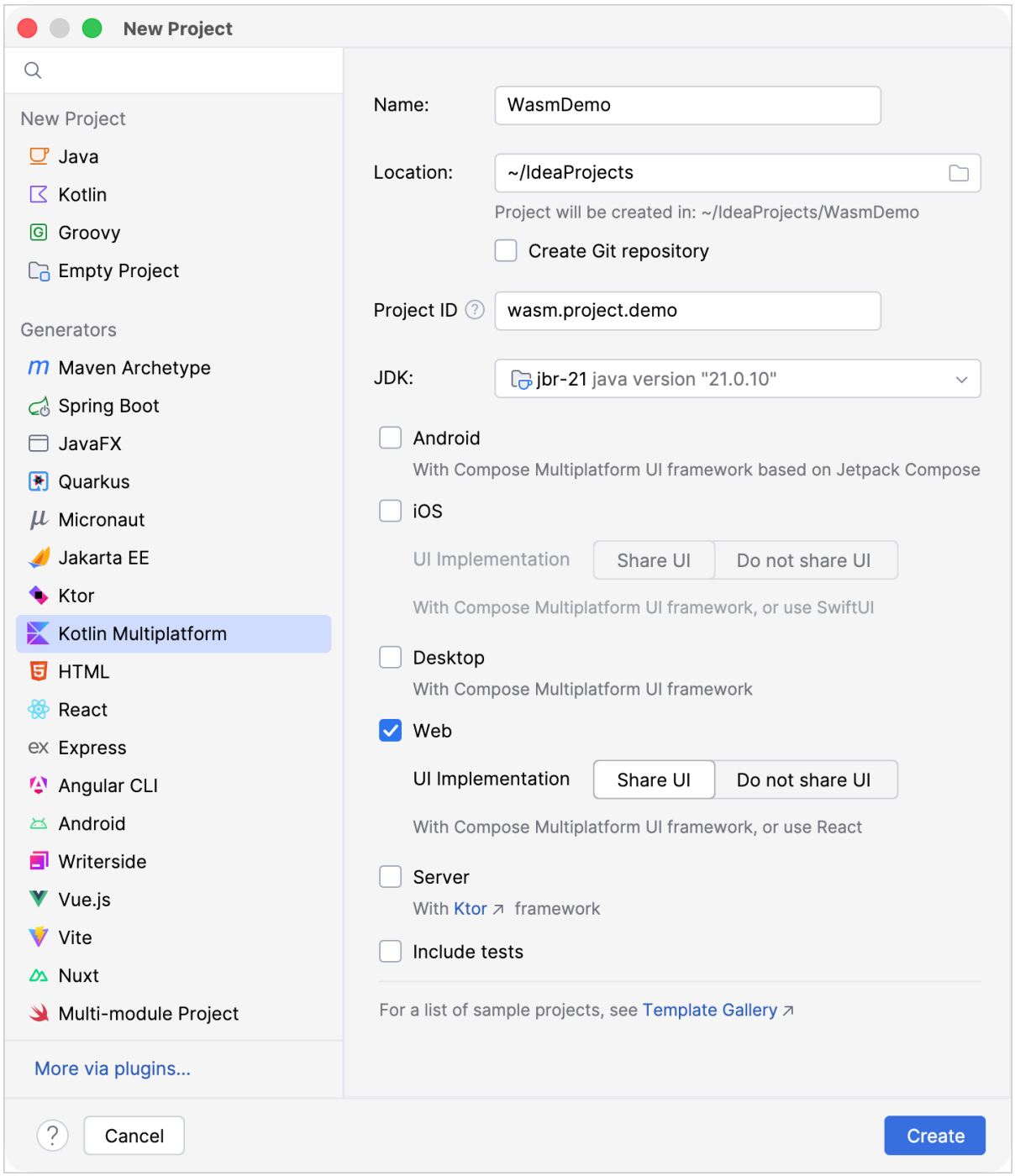
1. Set up your environment for Kotlin Multiplatform development.
2. In IntelliJ IDEA, select **File | New | Project**.
3. In the list of project templates, select **Kotlin Multiplatform**.

   If you're not using the Kotlin Multiplatform IDE plugin, you can generate the same project using the KMP web wizard.

4. Specify the following fields in the **New Project** window:
   - **Name:** WasmDemo
   - **Project ID:** wasm.project.demo

   This tutorial uses `wasm.project.demo` as the Project ID for consistency. However, we recommend keeping your usual group ID, such as `org.example`. Whatever you enter here will be suggested as the default in future projects.

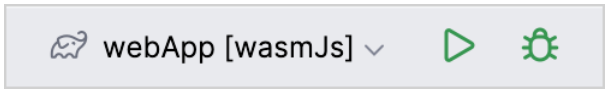
5. Select the **Web** target and the **Share UI** tab. Make sure that no other options are selected.
6. Click **Create**.



Kotlin Multiplatform wizard

Run the application

1. Once the project loads, select **webApp [wasmJs]** in the list of run configurations and click **Run**.



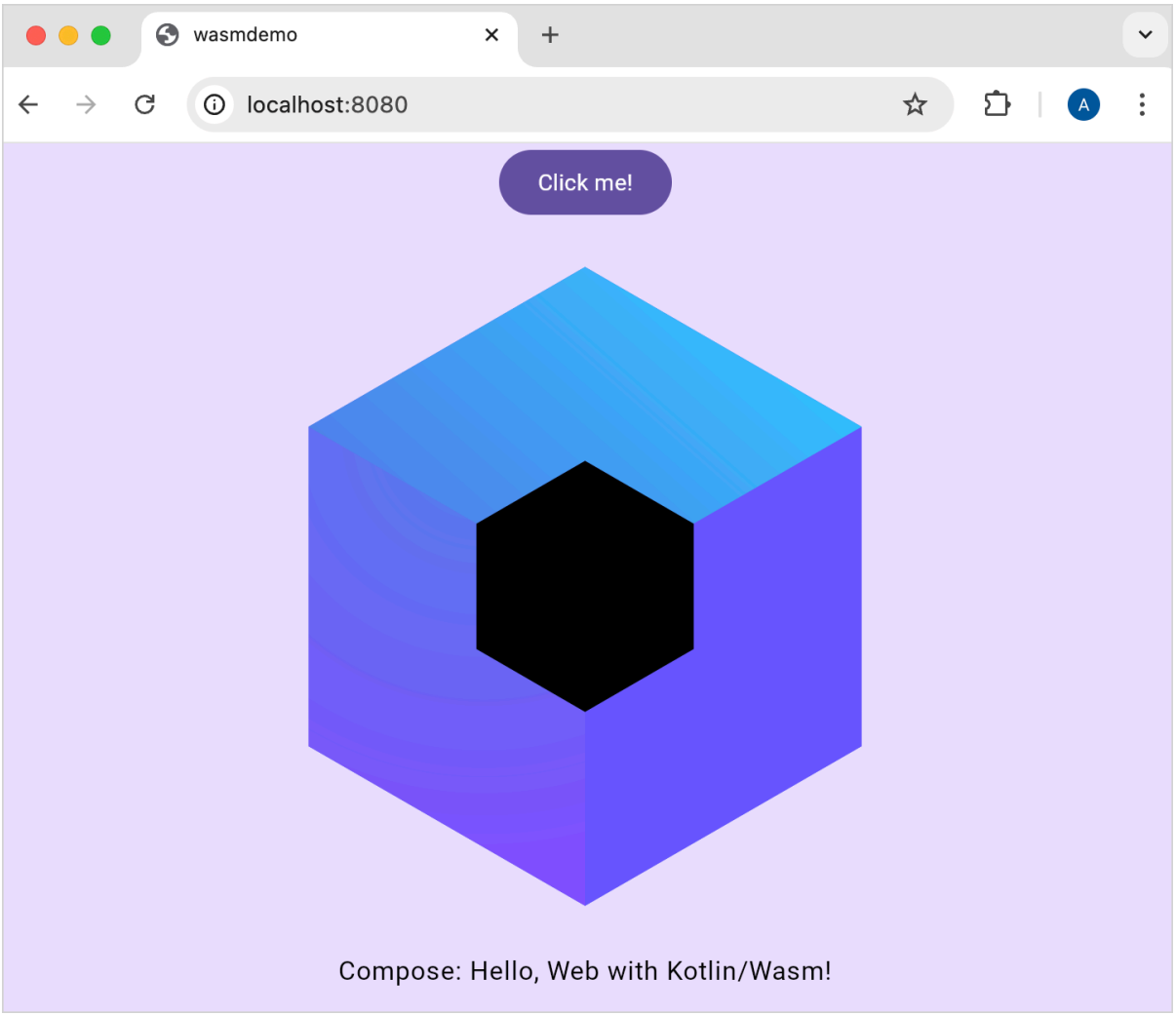
Run the Compose Multiplatform app on web

The web application opens automatically in your browser. Alternatively, once the build finishes, you can open the following URL manually:

```
http://localhost:8080/
```

The port number may vary if port `8080` is already in use. You can find the actual port number in the output of the Gradle build.

2. Click the **Click me!** button. This reveals the Compose Multiplatform logo:



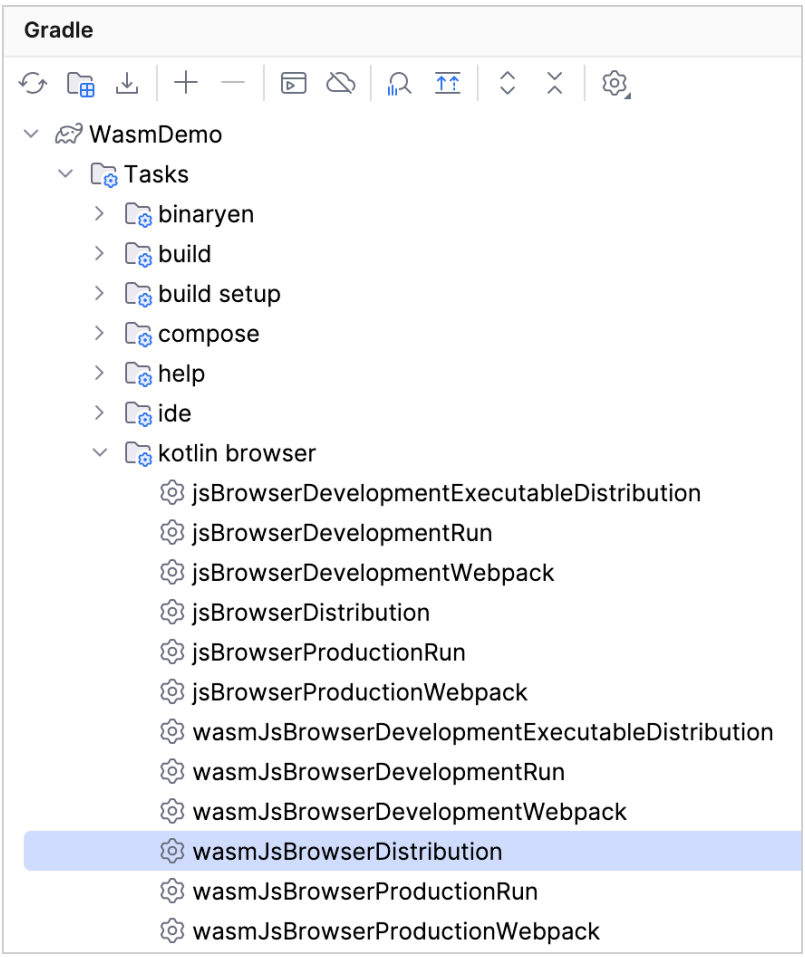
Compose app in browser

Generate artifacts

Generate your project's artifacts to publish on a website:

1. Open the **Gradle** tool window by selecting **View** | **Tool Windows** | **Gradle**.
2. In **WasmDemo** | **Tasks** | **kotlin browser**, select and run the **wasmJsBrowserDistribution** task.

> You need at least Java 11 as your Gradle JVM for the tasks to load successfully. For Compose Multiplatform projects in general, we recommend Java 17 or later.

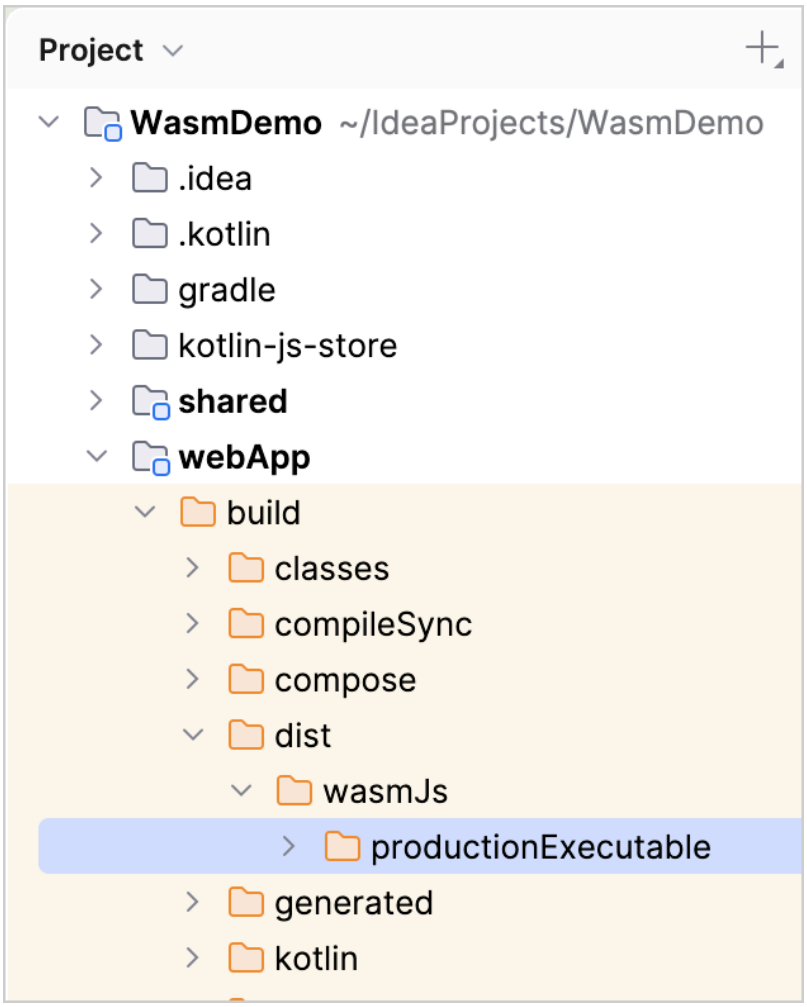


Run the Gradle task

Alternatively, you can run the following command in the terminal from the `WasmDemo` root directory:

```
./gradlew wasmJsBrowserDistribution
```

Once the task completes, you can find the generated artifacts in the `webApp/build/dist/wasmJs/productionExecutable` directory:



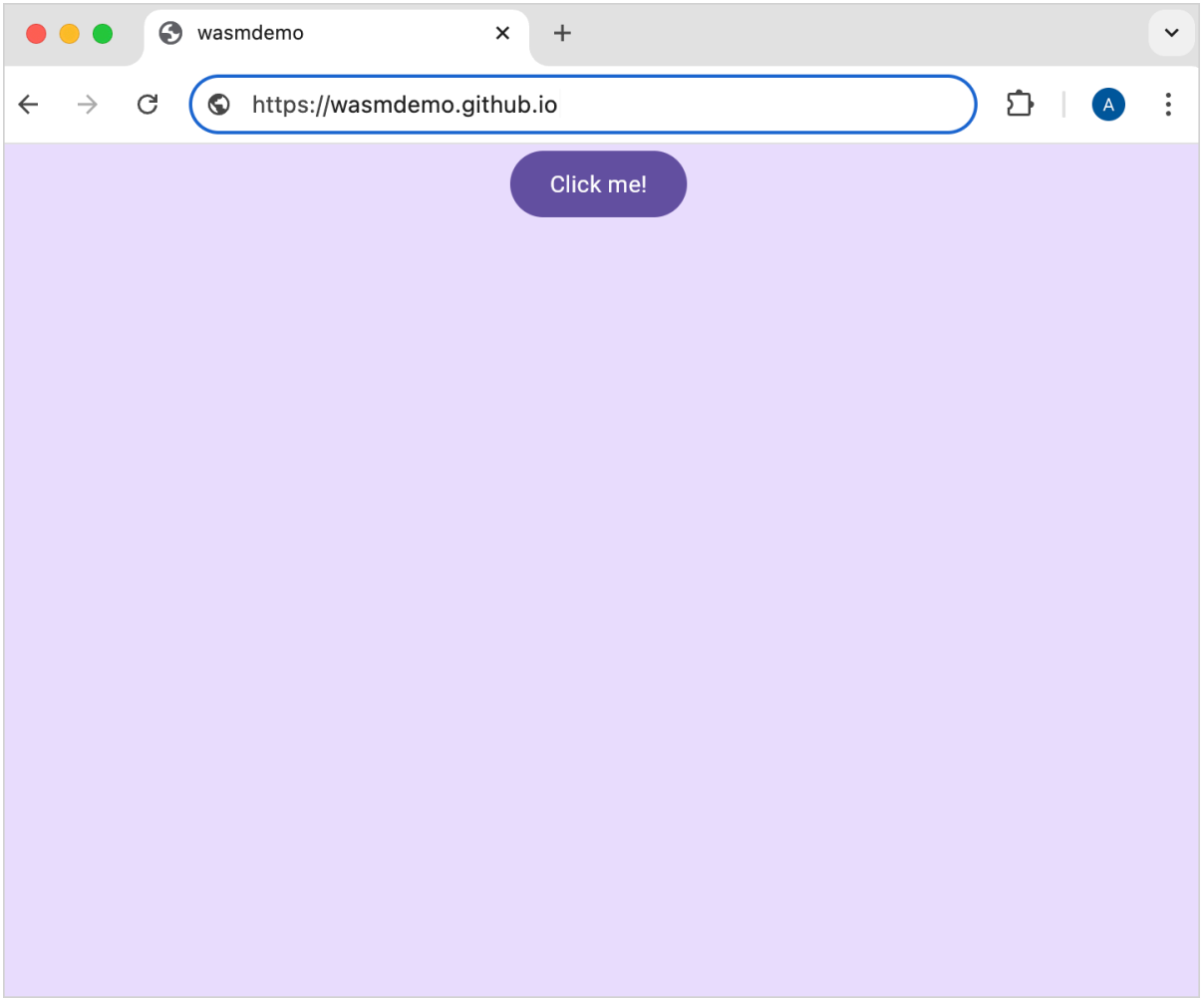
Artifacts directory

Publish the application

Use the generated artifacts to deploy your Kotlin/Wasm application. Select a publishing option that you prefer and follow the instructions:

- GitHub pages
- Cloudflare
- Apache HTTP Server

Once your site is created, open the browser and navigate to your platform's page domain. For example, GitHub pages:



Navigate to GitHub pages

Congratulations! You have published your artifacts.

What's next?

- Learn how to share UIs between iOS and Android using Compose Multiplatform
- Try more Kotlin/Wasm examples:
  - KotlinConf application
  - Compose image viewer
  - Node.js example
  - WASI example
  - Compose example
- Join the Kotlin/Wasm community in Kotlin Slack:

  Join the Kotlin/Wasm community

Beta

Get started with Kotlin/Wasm and WASI

This tutorial demonstrates how to run a simple Kotlin/Wasm application using the WebAssembly System Interface (WASI) in various WebAssembly virtual machines.

You can find examples of an application running on Node.js, Wasmtime Deno, and WasmEdge virtual machines. The output is a simple application that uses the standard WASI API.

Currently, Kotlin/Wasm supports WASI 0.1, also known as Preview 1. Support for WASI 0.2 is planned for future releases. Follow this YouTrack issue for updates on WASI 0.2 support.

The `wasmWasi` target uses the new exception handling proposal by default, ensuring better compatibility with modern WebAssembly runtimes.

> The Kotlin/Wasm toolchain provides Node.js tasks (`wasmWasiNode*`) and Wasmtime (`wasmWasiWasmtime*`) out of the box. Other task variants in the project, such as those utilizing Deno or WasmEdge, are included as custom tasks.

Before you start

1. Download and install the latest version of IntelliJ IDEA.
2. Clone the Kotlin/Wasm WASI template repository by selecting **File | New | Project from Version Control** in IntelliJ IDEA.

   You can also clone it from the command line:

   ```
   git clone git@github.com:Kotlin/kotlin-wasm-wasi-template.git
   ```

Run the application

1. Open the **Gradle** tool window by selecting **View** | **Tool Windows** | **Gradle**.

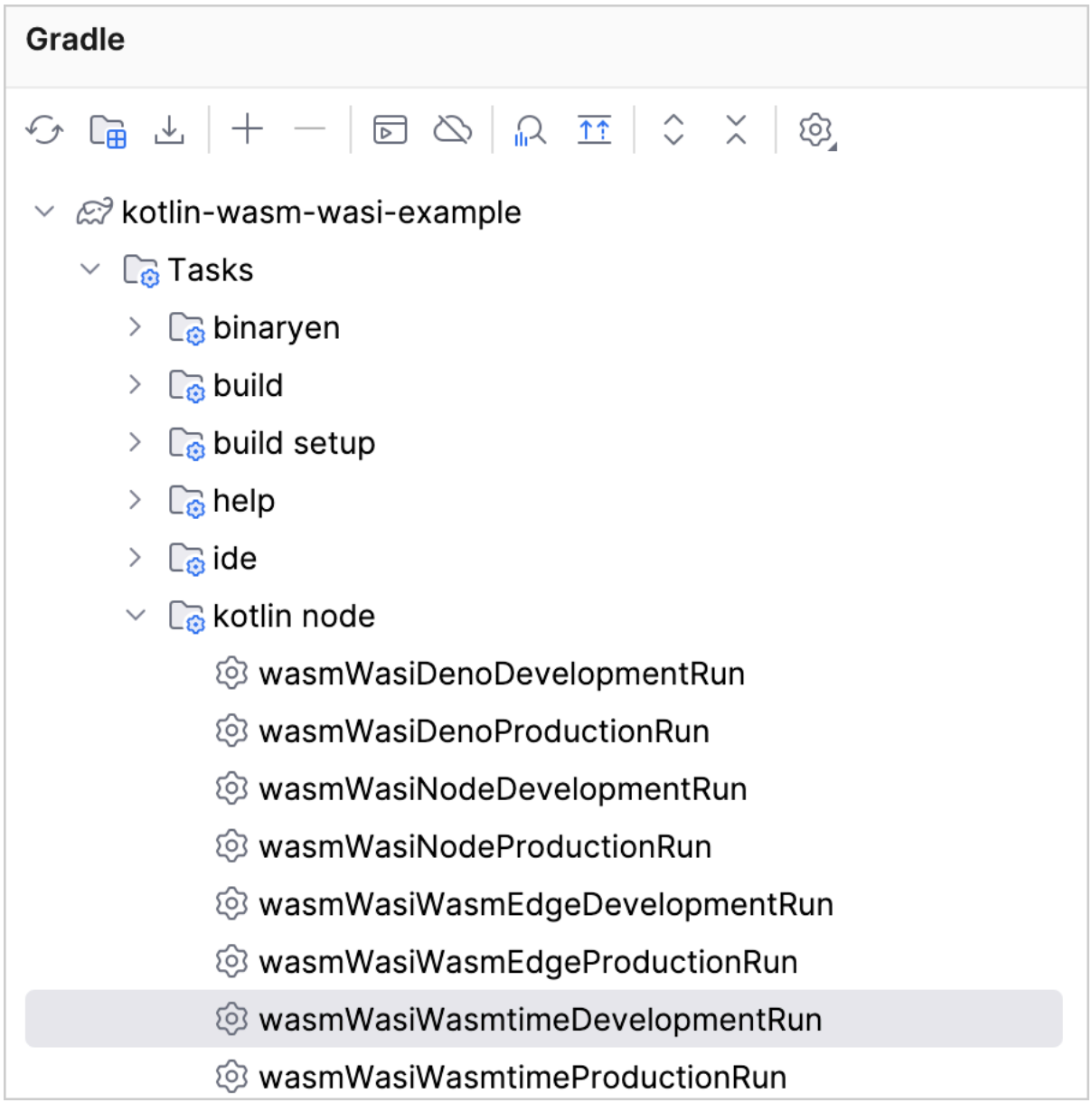
   In the **Gradle** tool window, you can find the Gradle tasks under **kotlin-wasm-wasi-example** once the project loads.

   > You need at least Java 11 as your Gradle JVM for the tasks to load successfully.

2. From **kotlin-wasm-wasi-example** | **Tasks** | **kotlin node**, select and run one of the following Gradle tasks:

   - **wasmWasiNodeDevelopmentRun** to run the application in Node.js.
   - **wasmWasiWasmtimeDevelopmentRun** to run the application in Wasmtime.
   - **wasmWasiDenoDevelopmentRun** to run the application in Deno.
   - **wasmWasiWasmEdgeDevelopmentRun** to run the application in WasmEdge.

   > When using Deno on a Windows platform, ensure `deno.exe` is installed. For more information, see Deno's installation documentation.



Kotlin/Wasm and WASI tasks

Alternatively, run one of the following commands in the terminal from the `kotlin-wasm-wasi-template` root directory:

- To run the application in Node.js:

```
./gradlew wasmWasiNodeDevelopmentRun
```

- To run the application in Wasmtime:

```
./gradlew wasmWasiWasmtimeDevelopmentRun
```

- To run the application in Deno:

```
./gradlew wasmWasiDenoDevelopmentRun
```

- To run the application in WasmEdge:

```
./gradlew wasmWasiWasmEdgeDevelopmentRun
```

The terminal displays a message when your application is built successfully:



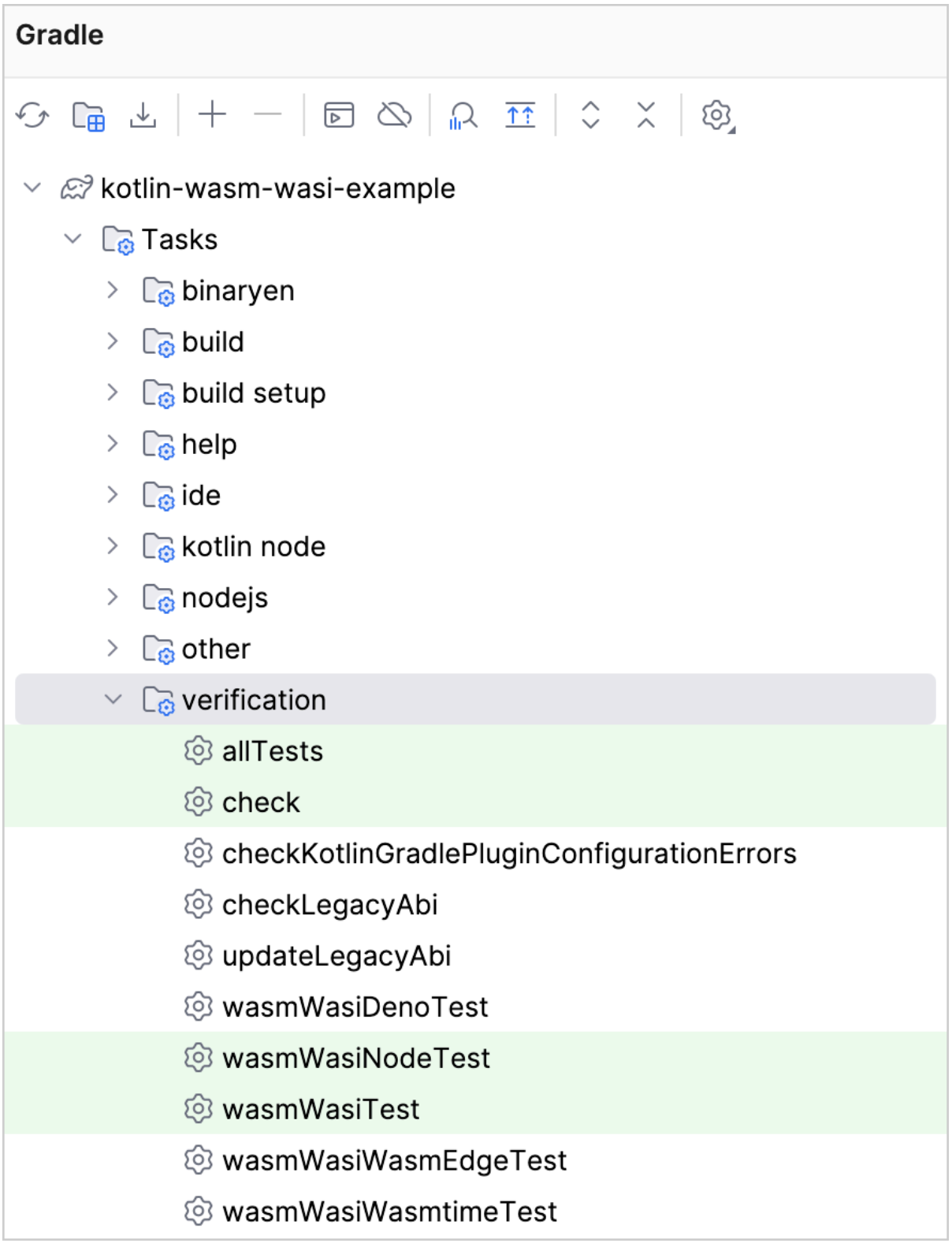
Kotlin/Wasm and WASI app

Test the application

You can also test that the Kotlin/Wasm application works correctly across various virtual machines.

In the Gradle tool window, run one of the following Gradle tasks from **kotlin-wasm-wasi-example** | **Tasks** | **verification**:

- **wasmWasiNodeTest** to test the application in Node.js.
- **wasmWasiWasmtimeTest** to test the application in Wasmtime.
- **wasmWasiDenoTest** to test the application in Deno.
- **wasmWasiWasmEdgeTest** to test the application in WasmEdge.



Kotlin/Wasm and WASI test tasks

Alternatively, run one of the following commands in the terminal from the `kotlin-wasm-wasi-template` root directory:

- To test the application in Node.js:

```
./gradlew wasmWasiNodeTest
```

- To test the application in Wasmtime:

```
./gradlew wasmWasiWasmtimeTest
```

- To test the application in Deno:

```
./gradlew wasmWasiDenoTest
```

- To test the application in WasmEdge:

```
./gradlew wasmWasiWasmEdgeTest
```

What's next?

Join the Kotlin/Wasm community in Kotlin Slack:

Join the Kotlin/Wasm community

Try more Kotlin/Wasm examples:

- Compose image viewer
- Node.js example
- Compose example

Beta

Debug Kotlin/Wasm code

This tutorial demonstrates how to use IntelliJ IDEA and the browser to debug your Compose Multiplatform application built with Kotlin/Wasm.

Before you start

1. Set up your environment for Kotlin Multiplatform development.
2. Follow the instructions to create a Kotlin Multiplatform project targeting Kotlin/Wasm.

- Debugging Kotlin/Wasm code in IntelliJ IDEA is available starting from version 2025.3 of the IDE, currently in the Early Access Program (EAP) and on the way to becoming stable. If you created the `WasmDemo` project in a different version of IntelliJ IDEA, switch to version 2025.3 and open the project there to continue this tutorial.
- To debug Kotlin/Wasm code in IntelliJ IDEA, you must have the JavaScript Debugger plugin installed. See more information about the plugin and how to install it.

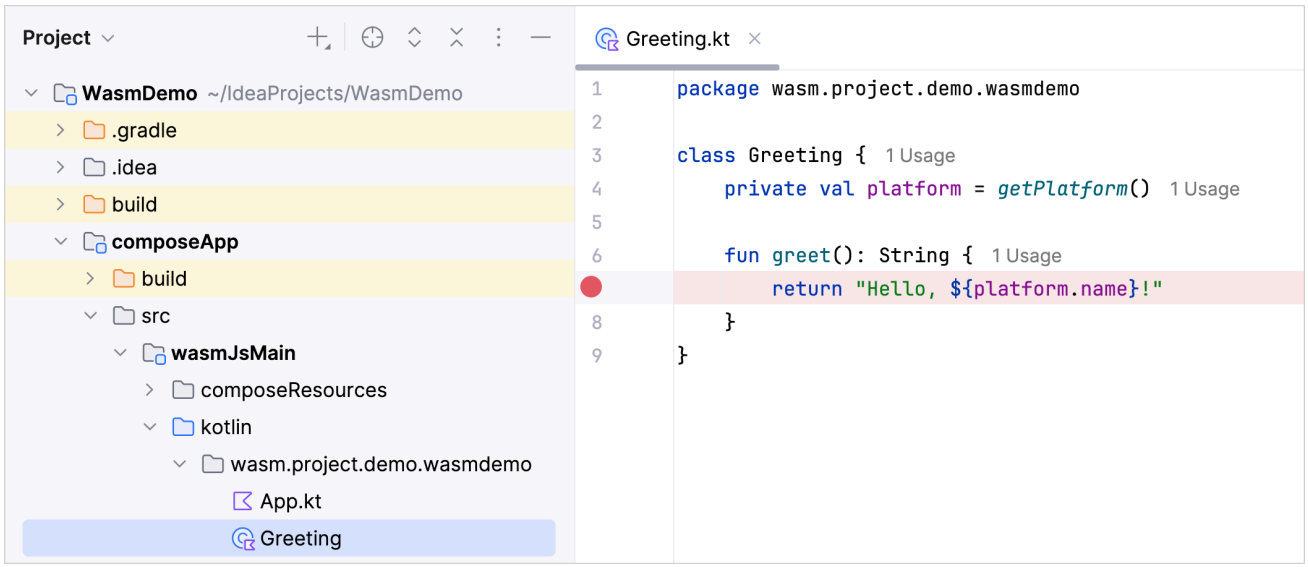
Debug in IntelliJ IDEA

The Kotlin Multiplatform project you created contains a Compose Multiplatform application powered by Kotlin/Wasm. You can debug this application in IntelliJ IDEA out of the box, without additional configuration.

1. In IntelliJ IDEA, open the Kotlin file to debug. In this tutorial, we'll work with the `Greeting.kt` file from the following directory of the Kotlin Multiplatform project:

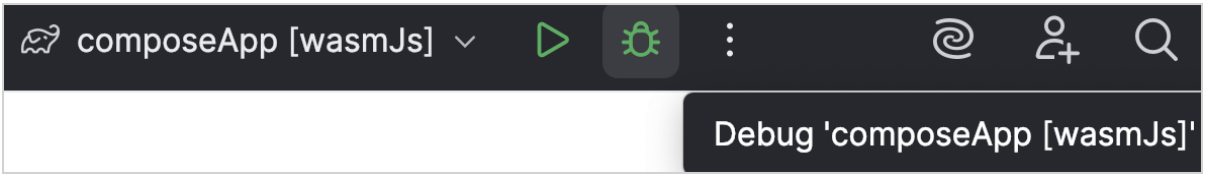
   `WasmDemo/composeApp/src/wasmJsMain/kotlin/wasm.project.demo.wasmdemo`

2. Click on the line numbers to set breakpoints on the code that you want to inspect.



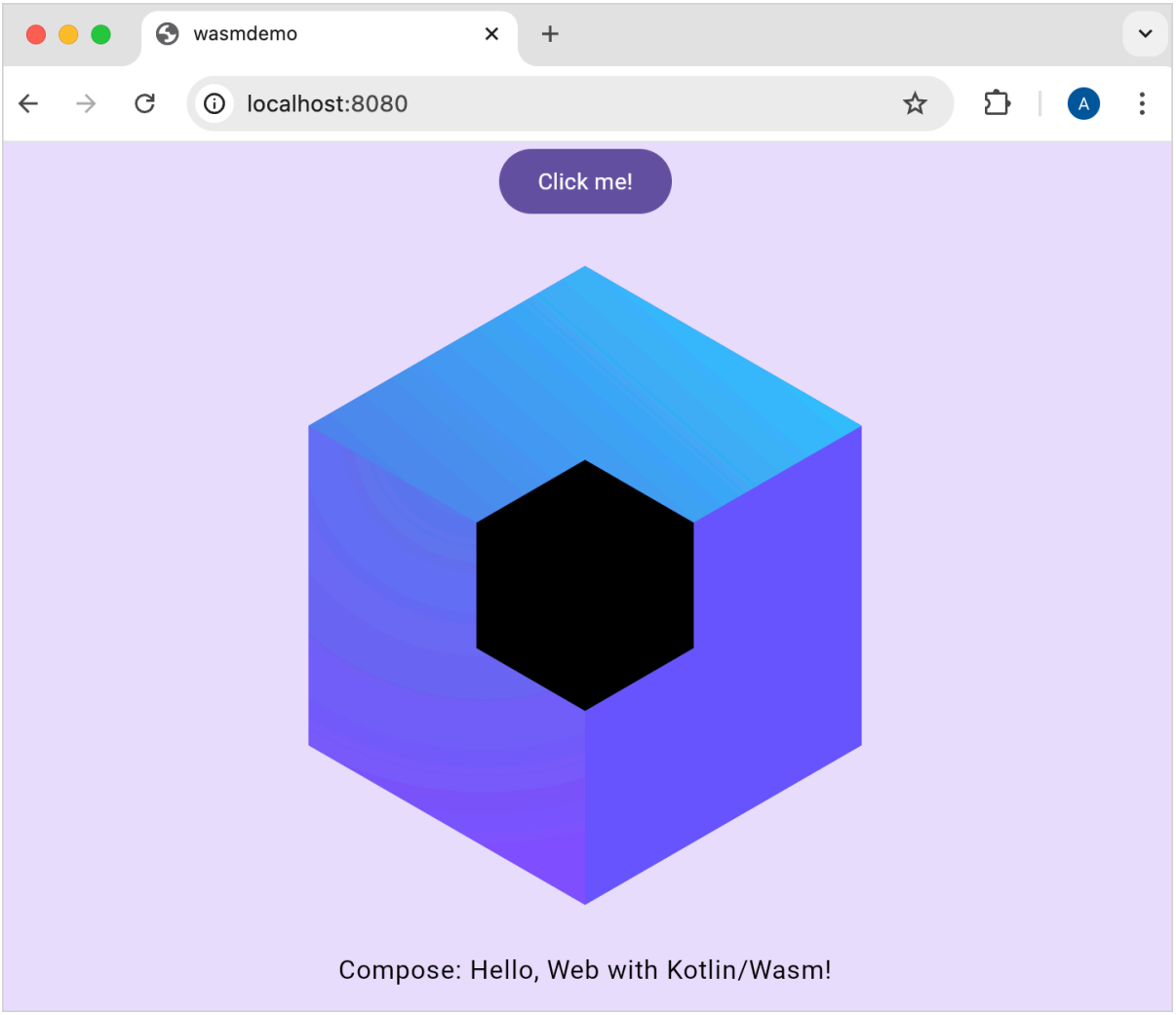
Set breakpoints

3. In the list of run configurations, select **composeApp[wasmJs]**.
4. Run the code in debug mode by clicking the debug icon at the top of the screen.



Run in debug mode

Once the application starts, it opens in a new browser window.



Compose app in browser

Also, the **Debug** panel opens automatically in IntelliJ IDEA.



Compose app debugger

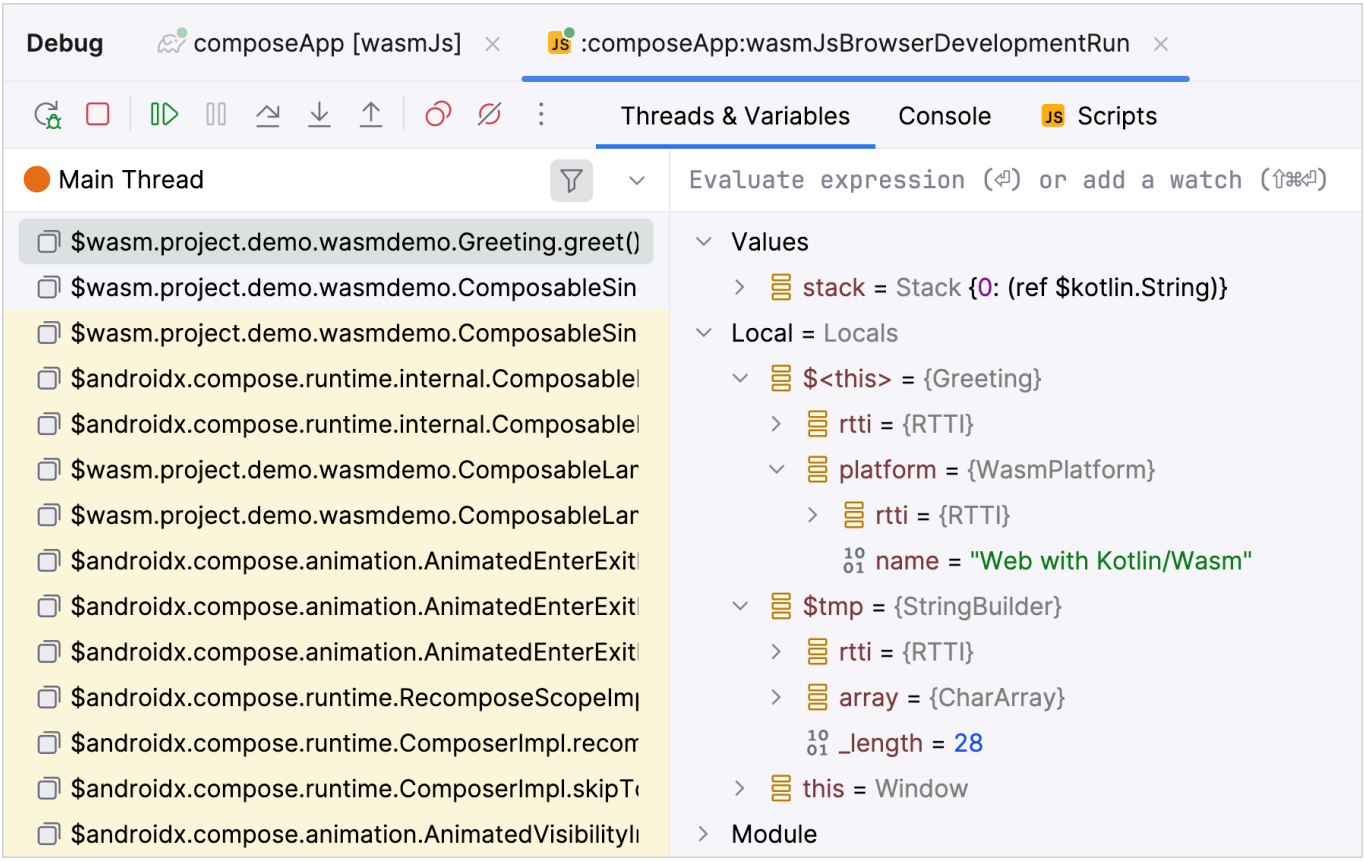
Inspect your application

If you're debugging in your browser, you can follow the same steps for inspecting your application.

1. In the application's browser window, click on the **Click me!** button to interact with the application. This action triggers the execution of the code, and the debugger pauses when the execution reaches a breakpoint.
2. In the debugging pane, use the debugging control buttons to inspect variables and code execution at the breakpoints:
   - Step over to execute the current line and pause on the next line.
   - Step into to investigate a function more deeply.

- Step out to execute the code until it exits the current function.

3. Check the **Threads & Variables** pane. It helps you trace the sequence of function calls and pinpoint the location of any errors.



Check Threads & Variables

4. Make changes to your code and run the application again to verify how it works.
5. When you're done debugging, click on the line numbers with breakpoints to remove the breakpoints.

Debug in your browser

You can also debug this Compose Multiplatform application in your browser without additional configuration.

When you run development Gradle tasks (`*DevRun`), Kotlin automatically serves the source files to the browser, allowing you to set breakpoints, inspect variables, and step through Kotlin code.

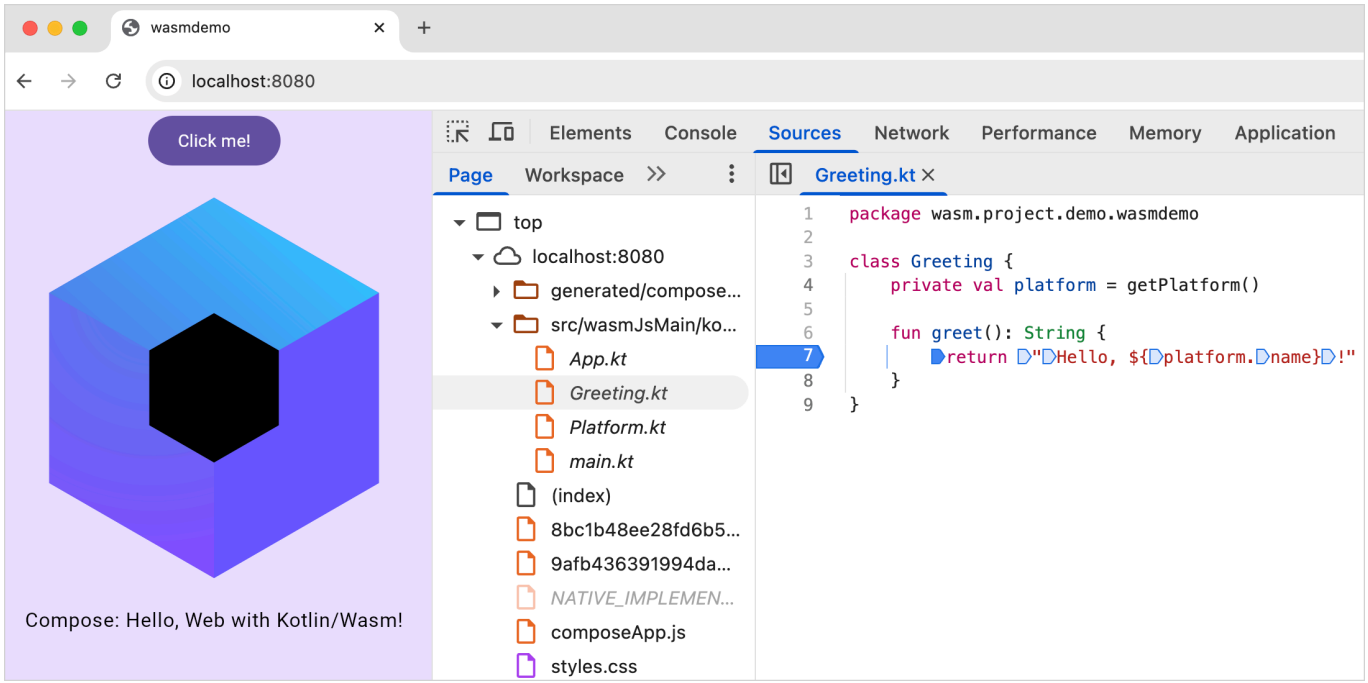
The configuration to serve the Kotlin/Wasm project sources in the browser is now included in the Kotlin Gradle plugin. If you previously added this configuration to your `build.gradle.kts` file, you should remove it to avoid conflicts.

> This tutorial uses the Chrome browser, but you should be able to follow these steps with other browsers. For more information, see Browser versions.

1. Follow the instructions to run the Compose Multiplatform application.
2. In the application's browser window, right-click and select the **Inspect** action to access developer tools. Alternatively, you can use the **F12** shortcut or select **View** | **Developer** | **Developer Tools**.

3. Switch to the **Sources** tab and select the Kotlin file to debug. In this tutorial, we'll work with the `Greeting.kt` file.

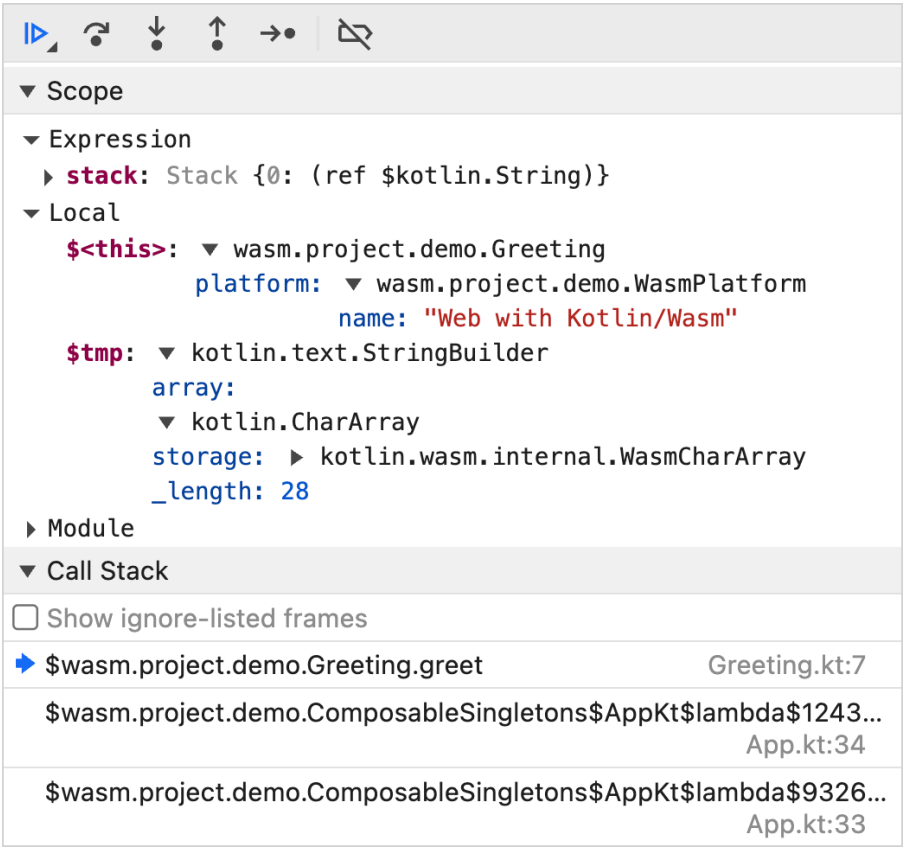
4. Click on the line numbers to set breakpoints on the code that you want to inspect. Only the lines with darker numbers can have breakpoints — in this example, 4, 7, 8, and 9.



Set breakpoints

5. Inspect your application similar to debugging in IntelliJ IDEA.

   When debugging in your browser, the panes to trace the sequence of function calls and pinpoint any errors are **Scope** and **Call Stack**.



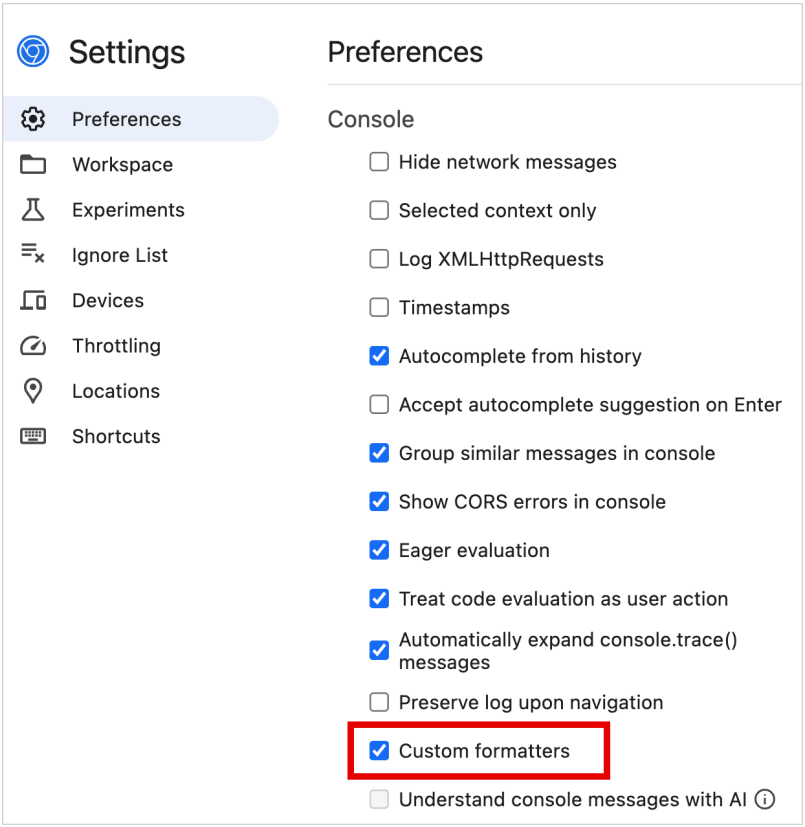
Check the call stack

Use custom formatters

Custom formatters help display and locate variable values in a more user-friendly and comprehensible manner when debugging Kotlin/Wasm code in your browser.

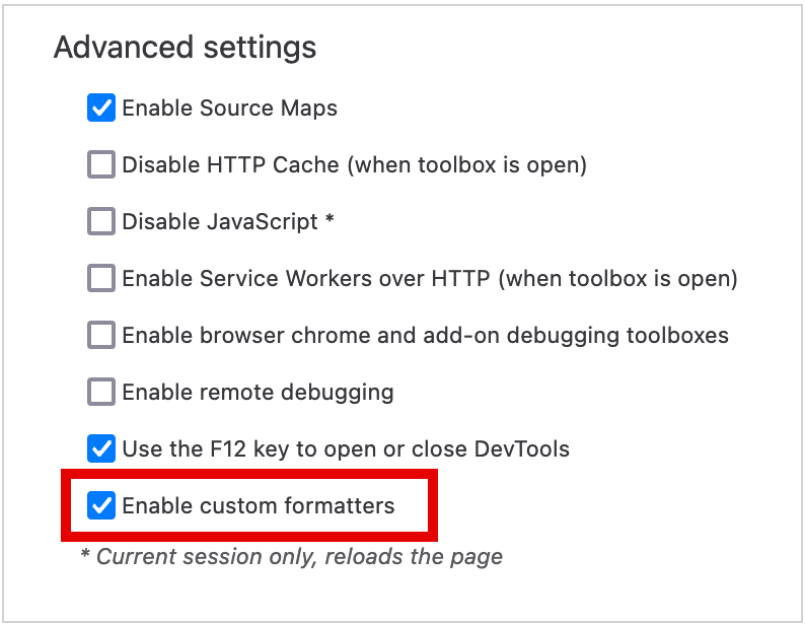
Custom formatters are enabled by default in Kotlin/Wasm development builds, but you still need to ensure that custom formatters are enabled in your browser's developer tools:

- In Chrome DevTools, find the **Custom formatters** checkbox in **Settings | Preferences | Console**:



Enable custom formatters in Chrome

- In Firefox DevTools, find the **Enable custom formatters** checkbox in **Settings | Advanced settings**:



Enable custom formatters in Firefox

This feature uses the custom formatters API, and is supported in Firefox and Chromium-based browsers.

Given that custom formatters work by default only for Kotlin/Wasm development builds, you need to adjust your Gradle configuration if you want to use them for production builds. Add the following compiler option to the `wasmJs {}` block:

```
// build.gradle.kts
kotlin {
    wasmJs {
        // ...

        compilerOptions {
            freeCompilerArgs.add("-Xwasm-debugger-custom-formatters")
        }
    }
}
```

Leave feedback

We would appreciate any feedback you may have on your debugging experience!

- Slack: get a Slack invite and provide your feedback directly to the developers in our #webassembly channel.
- Provide your feedback in YouTrack.

What's next?

- See Kotlin/Wasm debugging in action in this YouTube video.
- Try more Kotlin/Wasm examples:
  - KotlinConf application
  - Compose image viewer
  - Node.js example
  - WASI example
  - Compose example

Beta

Interoperability with JavaScript

Kotlin/Wasm allows you to use both JavaScript code in Kotlin and Kotlin code in JavaScript.

As with Kotlin/JS, the Kotlin/Wasm compiler also has interoperability with JavaScript. If you are familiar with Kotlin/JS interoperability, you can notice that Kotlin/Wasm interoperability is similar. However, there are key differences to consider.

Kotlin/Wasm is Beta. It may be changed at any time. Use it in scenarios before production. We would appreciate your feedback in YouTrack.

Use JavaScript code in Kotlin

Learn how to use JavaScript code in Kotlin by using `external` declarations, functions with JavaScript code snippets, and the `@JsModule` annotation.

External declarations

External JavaScript code is not visible in Kotlin by default. To use JavaScript code in Kotlin, you can describe its API with `external` declarations.

JavaScript functions

Consider this JavaScript function:

```
function greet (name) {
    console.log("Hello, " + name + "!");
}
```

You can declare it in Kotlin as an `external` function:

```
external fun greet(name: String)
```

External functions don't have bodies, and you can call it as a regular Kotlin function:

```
fun main() {
    greet("Alice")
}
```

JavaScript properties

Consider this global JavaScript variable:

```
let globalCounter = 0;
```

You can declare it in Kotlin using external `var` or `val` properties:

```
external var globalCounter: Int
```

These properties are initialized externally. The properties can't have `= value` initializers in Kotlin code.

JavaScript classes

Consider this JavaScript class:

```
class Rectangle {
    constructor (height, width) {
        this.height = height;
        this.width = width;
    }

    area () {
        return this.height * this.width;
    }
}
```

You can use it in Kotlin as an external class:

```
external class Rectangle(height: Double, width: Double) : JsAny {
    val height: Double
    val width: Double
    fun area(): Double
}
```

All declarations inside the `external` class are implicitly considered external.

External interfaces

You can describe the shape of a JavaScript object in Kotlin. Consider this JavaScript function and what it returns:

```
function createUser (name, age) {
    return { name: name, age: age };
}
```

See how its shape can be described in Kotlin with an `external interface User` type:

```
external interface User : JsAny {
    val name: String
    val age: Int
}

external fun createUser(name: String, age: Int): User
```

External interfaces don't have runtime type information and are a compile-time-only concept. Therefore, external interfaces have some restrictions compared to regular interfaces:

- You can't use them on the right-hand side of `is` checks.
- You can't use them in class literal expressions (such as `User::class`).
- You can't pass them as reified type arguments.
- Casting with `as` to external interfaces always succeeds.

External objects

Consider these JavaScript variables holding an object:

```
let Counter = {
    value: 0,
    step: 1,
    increment () {
        this.value += this.step;
    }
};
```

You can use them in Kotlin as an external object:

```
external object Counter : JsAny {
    fun increment()
    val value: Int
    var step: Int
}
```

External type hierarchy

Similar to regular classes and interfaces, you can declare external declarations to extend other external classes and implement external interfaces. However, you can't mix external and non-external declarations in the same type hierarchy.

Experimental

Callable JavaScript objects with `@nativeInvoke`

You can use the `@nativeInvoke` annotation on a Kotlin member function of an `external` declaration (a class or an interface) to make it callable as a JavaScript function.

With this annotation, each call to that function in Kotlin translates to a direct call to the JavaScript object:

```
import kotlin.js.nativeInvoke

@OptIn(ExperimentalWasmJsInterop::class)
external class JsAction {
    @nativeInvoke
    operator fun invoke(data: String)
}

fun main() {
    val action = JsAction()
    action("Run task")
}
```

The `@nativeInvoke` annotation is a temporary solution until there's a design for stable interoperability. Currently, when you use `@nativeInvoke`, the compiler reports a warning.

Kotlin functions with JavaScript code

You can add a JavaScript snippet to Kotlin/Wasm code by defining a function with `= js("code")` body:

```
fun getCurrentURL(): String =
    js("window.location.href")
```

If you want to run a block of JavaScript statements, surround your code inside the string with curly brackets `{}`:

```
fun setLocalSettings(value: String): Unit = js(
    """{
        localStorage.setItem('settings', value);
}"""
)
```

If you want to return an object, surround the curly brackets `{}` with parentheses `()`:

```
fun createJsUser(name: String, age: Int): JsAny =
    js("({ name: name, age: age })")
```

Kotlin/Wasm treats calls to the `js()` function in a special way, and the implementation has some restrictions:

- A `js()` function call must be provided with a string literal argument.
- A `js()` function call must be the only expression in the function body.

- The `js()` function is only allowed to be called from package-level functions.
- The function return type must be provided explicitly.
- Types are restricted, similar to `external fun`.

The Kotlin compiler puts the code string into a function in the generated JavaScript file and imports it into WebAssembly format. The Kotlin compiler doesn't verify these JavaScript snippets. If there are JavaScript syntax errors, they are reported when you run your JavaScript code.



The `@JsFun` annotation has similar functionality and will likely be deprecated.

JavaScript modules

By default, external declarations correspond to the JavaScript global scope. If you annotate a Kotlin file with the `@JsModule` annotation, then all external declarations within it are imported from the specified module.

Consider this JavaScript code sample:

```
// users.mjs
export let maxUsers = 10;

export class User {
    constructor (username) {
        this.username = username;
    }
}
```

Use this JavaScript code in Kotlin with the `@JsModule` annotation:

```
// Kotlin
@file:JsModule("./users.mjs")

external val maxUsers: Int

external class User : JsAny {
    constructor(username: String)

    val username: String
}
```

Array interoperability

You can copy JavaScript's `JsArray<T>` into Kotlin's native `Array` or `List` types; likewise, you can copy these Kotlin types to `JsArray<T>`.

To convert `JsArray<T>` to `Array<T>` or the other way around, use one of the available adapter functions.

Here's an example of conversion between generic types:

```
val list: List<JsString> =
    listOf("Kotlin", "Wasm").map { it.toJsString() }

// Uses .toJsArray() to convert List or Array to JsArray
val jsArray: JsArray<JsString> = list.toJsArray()

// Uses .toArray() and .toList() to convert it back to Kotlin types
val kotlinArray: Array<JsString> = jsArray.toArray()
val kotlinList: List<JsString> = jsArray.toList()
```

Similar adapter functions are available for converting typed arrays to their Kotlin equivalents (for example, `IntArray` and `Int32Array`). For detailed information and implementation, see the `kotlinx-browser` repository.

Here's an example of conversion between typed arrays:

```
import org.khronos.webgl.*

    // ...

    val intArray: IntArray = intArrayOf(1, 2, 3)

    // Uses .toInt32Array() to convert Kotlin IntArray to JavaScript Int32Array
    val jsInt32Array: Int32Array = intArray.toInt32Array()

    // Uses toIntArray() to convert JavaScript Int32Array back to Kotlin IntArray
    val kotlinIntArray: IntArray = jsInt32Array.toIntArray()
```

Use Kotlin code in JavaScript

Learn how to use your Kotlin code in JavaScript by using the `@JsExport` annotation.

Functions with the @JsExport annotation

To make a Kotlin/Wasm function available to JavaScript code, use the `@JsExport` annotation:

```
// Kotlin/Wasm

@JsExport
fun addOne(x: Int): Int = x + 1
```

Kotlin/Wasm functions marked with the `@JsExport` annotation are visible as properties on a `default` export of the generated `.mjs` module. You can then use this function in JavaScript:

```
// JavaScript

import exports from "./module.mjs"

exports.addOne(10)
```

The Kotlin/Wasm compiler is capable of generating TypeScript definitions from any `@JsExport` declarations in your Kotlin code. These definitions can be used by IDEs and JavaScript tools to provide code autocompletion, help with type-checks, and make it easier to consume Kotlin code from JavaScript and TypeScript.

The Kotlin/Wasm compiler collects any top-level functions marked with the `@JsExport` annotation and automatically generates TypeScript definitions in a `.d.ts` file.

To generate TypeScript definitions, in your `build.gradle.kts` file in the `wasmJs{}` block, add the `generateTypeScriptDefinitions()` function:

```
kotlin {
    wasmJs {
        binaries.executable()
        browser {
        }
        generateTypeScriptDefinitions()
    }
}
```

Generating TypeScript declaration files in Kotlin/Wasm is Experimental. It may be dropped or changed at any time.

Type correspondence

Kotlin/Wasm allows only certain types in signatures of JavaScript interop declarations. These limitations apply uniformly to declarations with `external`, `= js("code")` or `@JsExport`.

See how Kotlin types correspond to JavaScript types:

Kotlin	JavaScript
`Byte`, `Short`, `Int`, `Char`, `UByte`, `UShort`, `UInt`,	`Number`
`Float`, `Double`,	`Number`
`Long`, `ULong`,	`BigInt`
`Boolean`,	`Boolean`
`String`,	`String`
`Unit` in return position	`undefined`
Function type, for example `(String) -> Int`	Function
`JsAny` and subtypes	Any JavaScript value
`JsReference`	Opaque reference to Kotlin object
Other types	Not supported

You can use nullable versions of these types as well.

JsAny type

JavaScript values are represented in Kotlin using the `JsAny` type and its subtypes.

The Kotlin/Wasm standard library provides representation for some of these types:

- Package `kotlin.js`:
  - `JsAny`
  - `JsBoolean`, `JsNumber`, `JsString`
  - `JsArray`
  - `Promise`

You can also create custom `JsAny` subtypes by declaring an `external` interface or class.

JsReference type

Kotlin values can be passed to JavaScript as opaque references using the `JsReference` type.

For example, if you want to expose this Kotlin class `User` to JavaScript:

```
class User(var name: String)
```

You can use the `toJsReference()` function to create `JsReference<User>` and return it to JavaScript:

```
@JsExport
fun createUser(name: String): JsReference<User> {
    return User(name).toJsReference()
}
```

These references aren't directly available in JavaScript and behave like empty frozen JavaScript objects. To operate on these objects, you need to export more functions to JavaScript using the `get()` method where you unwrap the reference value:

```
@JsExport
fun setUserName(user: JsReference<User>, name: String) {
    user.get().name = name
}
```

You can create a class and change its name from JavaScript:

```
import UserLib from "./userlib.mjs"

let user = UserLib.createUser("Bob");
UserLib.setUserName(user, "Alice");
```

Type parameters

JavaScript interop declarations can have type parameters if they have an upper bound of `JsAny` or its subtypes. For example:

```
external fun <T : JsAny> processData(data: JsArray<T>): T
```

Exception handling

You can use Kotlin `try-catch` expressions to catch JavaScript exceptions in Kotlin/Wasm code. Exception handling works as follows:

- Exceptions thrown from JavaScript: detailed information is available on the Kotlin side. If such an exception propagates back to JavaScript, it's no longer wrapped into WebAssembly.
- Exceptions thrown from Kotlin: they can be caught on JavaScript's side as regular JS errors.

Here's an example that demonstrates catching a JavaScript exception on the Kotlin side:

```
external object JSON {
    fun <T: JsAny> parse(json: String): T
}

fun main() {
    try {
        JSON.parse("an invalid JSON")
    } catch (e: JsException) {
        println("Thrown value is: ${e.thrownValue}")
        // SyntaxError: Unexpected token 'a', "an invalid JSON" is not valid JSON

        println("Message: ${e.message}")
        // Message: Unexpected token 'a', "an invalid JSON" is not valid JSON

        println("Stacktrace:")
        // Stacktrace:

        // Prints the full JavaScript stack trace
        e.printStackTrace()
    }
}
```

This exception handling works automatically in modern browsers that support the `WebAssembly.JSTag` feature:

- Chrome 115+
- Firefox 129+
- Safari 18.4+

Kotlin/Wasm and Kotlin/JS interoperability differences

Although Kotlin/Wasm interoperability shares similarities with Kotlin/JS interoperability, there are key differences to consider:

	Kotlin/Wasm	Kotlin/JS
External enums	Doesn't support external enum classes.	Supports external enum classes.
Type extensions	Doesn't support non-external types to extend external types.	Supports non-external types.

	Kotlin/Wasm	Kotlin/JS
`JsName` annotation	Only has an effect when annotating external declarations.	Can be used to change names of regular non-external declarations.
`js()` function	`js("code")` function calls are allowed as a single expression body of package-level functions.	The `js("code")` function can be called in any context and returns a `dynamic` value.
Module systems	Supports ES modules only. There is no analog of the `@JsNonModule` annotation. Provides its exports as properties on the `default` object. Allows exporting package-level functions only.	Supports ES modules and legacy module systems. Provides named ESM exports. Allows exporting classes and objects.
Types	Applies stricter type restrictions uniformly to all interop declarations `external`, `= js("code")`, and `@JsExport`. Allows a select number of built-in Kotlin types and `JsAny` subtypes.	Allows all types in `external` declarations. Restricts types that can be used in `@JsExport`.
Long	Type corresponds to JavaScript `BigInt`.	Visible as a custom class in JavaScript.
Arrays	Not supported in interop directly yet. You can use the new `JsArray` type instead.	Implemented as JavaScript arrays.
Other types	Requires `JsReference<>` to pass Kotlin objects to JavaScript.	Allows the use of non-external Kotlin class types in external declarations.
Exception handling	You can catch any JavaScript exception with the `JsException` and `Throwable` types.	Can catch JavaScript `Error` using the `Throwable` type. It can catch any JavaScript exception using the `dynamic` type.
Dynamic types	Does not support the `dynamic` type. Use `JsAny` instead (see sample code below).	Supports the `dynamic` type.

Kotlin/JS dynamic type for interoperability with untyped or loosely typed objects is not supported in Kotlin/Wasm. Instead of the `dynamic` type, you can use the `JsAny` type:

```
// Kotlin/JS
fun processUser(user: dynamic, age: Int) {
    // ...
    user.profile.updateAge(age)
    // ...
}

// Kotlin/Wasm
private fun updateUserAge(user: JsAny, age: Int): Unit =
    js("{ user.profile.updateAge(age); }")

fun processUser(user: JsAny, age: Int) {
    // ...
    updateUserAge(user, age)
    // ...
}
```

Web-related browser APIs

The `kotlinx-browser` library is a standalone library that provides JavaScript browser APIs, including:

- Package `org.khronos.webgl`:
  - Typed arrays, like `Int8Array`.
  - WebGL types.
- Packages `org.w3c.dom.*`:
  - DOM API types.
- Package `kotlinx.browser`:
  - DOM API global objects, like `window` and `document`.

To use the declarations from the `kotlinx-browser` library, add it as a dependency in your project's build configuration file:

```
val wasmJsMain by getting {
    dependencies {
        implementation("org.jetbrains.kotlinx:kotlinx-browser:0.3")
    }
}
```

Beta

Supported versions and configuration

This page provides details about the WebAssembly proposals, supported browsers, and configuration recommendations for efficient development with Kotlin/Wasm.

Browser versions

Kotlin/Wasm relies on the latest WebAssembly proposals, such as garbage collection (WasmGC) and exception handling, to introduce improvements and new features within WebAssembly.

To make sure these features work properly, provide an environment that supports the latest proposals. Check if your browser version supports the new WasmGC by default or if you need to make changes to the environment.

Chrome

- **For version 119 or later:**

  Works by default.

- **For older versions:**

  > To run applications in an older browser, you need a Kotlin version older than 1.9.20.

  1. In your browser, go to `chrome://flags/#enable-webassembly-garbage-collection`.
  2. Enable **WebAssembly Garbage Collection**.
  3. Relaunch your browser.

Chromium-based

Including Chromium-based browsers such as Edge, Brave, Opera, or Samsung Internet.

- **For version 119 or later:**

  Works by default.

- **For older versions:**

  > To run applications in an older browser, you need a Kotlin version older than 1.9.20.

  Run the application with the `--js-flags=--experimental-wasm-gc` command line argument.

Firefox

- **For version 120 or later:**

  Works by default.

- **For version 119:**

  1. In your browser, go to `about:config`.
  2. Enable `javascript.options.wasm_gc` option.
  3. Refresh the page.

Safari/WebKit

- **For version 18.2 or later:**

  Works by default.

- **For older versions:**

  Not supported.

> Safari 18.2 is available for iOS 18.2, iPadOS 18.2, visionOS 2.2, macOS 15.2, macOS Sonoma, and macOS Ventura. On iOS and iPadOS, Safari 18.2 is bundled with the operating system. To get it, update your device to version 18.2 or later.
>
> For more information, see the Safari release notes.

Wasm proposals support

Kotlin/Wasm improvements are based on WebAssembly proposals. Here you can find details about the support for WebAssembly's garbage collection and (legacy) exception handling proposals.

Garbage collection proposal

Since Kotlin 1.9.20, the Kotlin toolchain uses the latest version of the Wasm garbage collection (WasmGC) proposal.

For this reason, we strongly recommend that you update your Wasm projects to the latest version of Kotlin. We also recommend you use the latest versions of browsers with the Wasm environment.

Exception handling proposal

The Kotlin toolchain supports both the legacy and the new versions of the exception handling proposal. This allows Kotlin-produced Wasm binaries to run in a wider range of environments.

The `wasmJs` target uses the legacy exception handling proposal by default. To enable the new exception handling proposal for the `wasmJs` target, use the `-Xwasm-use-new-exception-proposal` compiler option.

By contrast, the `wasmWasi` target uses the new proposal by default, ensuring better compatibility with modern WebAssembly runtimes. To switch to the legacy proposal, use the `-Xwasm-use-new-exception-proposal=false` compiler option.

For the `wasmWasi` target, it is safe to adopt the new exception handling proposal. Applications targeting this environment usually run in a less diverse runtime environment (often running on a single specific VM) that is typically controlled by the user, reducing the risk of compatibility issues.

> Learn more about setting up projects, using dependencies, and other tasks with our Kotlin/Wasm examples.

Use default import

Importing Kotlin/Wasm code into Javascript has shifted to named exports, moving away from default exports.

If you still want to use a default import, generate a new JavaScript wrapper module. Create a `.mjs` file with the following snippet:

```
// Specifies the path to the main .mjs file
import * as moduleExports from "./wasm-test.mjs";

export { moduleExports as default };
```

You can place your new `.mjs` file in the resources folder, and it will automatically be placed next to the main `.mjs` file during the build process.

You can also place your `.mjs` file in a custom location. In this case, you need to either manually move it next to the main `.mjs` file or adjust the path in the import statement to match its location.

Kotlin/Wasm incremental compilation

Kotlin/Wasm targets support incremental compilation, which allows the compiler to recompile only the files affected by recent changes. This helps reduce compilation time.

Incremental compilation for the Wasm targets is enabled by default. To disable it, add the following line to your project's `local.properties` or `gradle.properties` file:

```
kotlin.incremental.wasm=false
```

Diagnostics in fully qualified class names

On Kotlin/Wasm, the compiler does not store fully qualified names (FQNs) of classes in the generated binary by default, to avoid increasing the application size.

For this reason, the compiler reports an error when you call the `KClass::qualifiedName` property in Kotlin/Wasm projects, unless you explicitly enable the qualified names feature.

This diagnostic is enabled by default, and errors are reported automatically. To disable the diagnostic and allow `qualifiedName` in Kotlin/Wasm, instruct the compiler to store fully qualified names for all classes by adding the following option to your `build.gradle.kts` file:

```
// build.gradle.kts
kotlin {
   wasmJs {
       ...
       compilerOptions {
           freeCompilerArgs.add("-Xwasm-kclass-fqn")
       }
   }
}
```

Keep in mind that enabling this option increases the application size.

Fully qualified names

On Kotlin/Wasm targets, fully qualified names (FQNs) are available at runtime without any additional configuration. This means that the `KClass.qualifiedName` property is enabled by default.

Using FQNs improves code portability from JVM to Wasm targets and makes runtime errors more informative by showing the full qualified name.

Array out-of-bounds access and traps

In Kotlin/Wasm, accessing an array with an index outside its bounds triggers a WebAssembly trap instead of a regular Kotlin exception. The trap immediately stops the current stack of execution.

When running in a JavaScript environment, these traps appear as `WebAssembly.RuntimeError` and can be caught on the JavaScript side.

You can avoid such traps in Kotlin/Wasm environments by using the following compiler option in the command line when linking an executable:

```
-Xwasm-enable-array-range-checks
```

Or add it to the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xwasm-enable-array-range-checks")
    }
}
```

With the compiler option enabled, an `IndexOutOfBoundsException` is thrown instead of a trap.

See more details and share your feedback in this YouTrack issue.

Experimental annotations

Kotlin/Wasm provides several experimental annotations for general WebAssembly interoperability.

`@WasmImport` and `@WasmExport` let you call functions defined outside a Kotlin/Wasm module and expose Kotlin functions to the host or other Wasm modules, respectively.

Because these mechanisms are still evolving, all annotations are marked as experimental. You must explicitly opt in to use them, and their design or behavior may change in future Kotlin versions.

Reloads during debugging

Debugging your applications in modern browsers works out of the box. When you run development Gradle tasks (`*DevRun`), Kotlin automatically serves the source files to the browser.

However, serving sources by default may cause repeated reloads of the application in the browser before Kotlin compilation and bundling are complete. As a workaround, adjust your webpack configuration to ignore Kotlin source files and disable watching for served static files. Add a `.js` file with the following content into the `webpack.config.d` directory at the root of your project:

```
config.watchOptions = config.watchOptions || {
    ignored: ["**/*.kt", "**/node_modules"]
}

if (config.devServer) {
    config.devServer.static = config.devServer.static.map(file => {
        if (typeof file === "string") {
            return {
                directory: file,
                watch: false,
            }
        } else {
            return file
        }
    })
}
```

Kotlin/Native

Kotlin/Native is a technology for compiling Kotlin code to native binaries that can run without a virtual machine. Kotlin/Native includes an LLVM-based backend for the Kotlin compiler and a native implementation of the Kotlin standard library.

Why Kotlin/Native?

Kotlin/Native is primarily designed to allow compilation for platforms on which *virtual machines* are not desirable or possible, such as embedded devices or iOS. It's ideal for situations when you need to produce a self-contained program that doesn't require an additional runtime or a virtual machine.

It's easy to include compiled Kotlin code in existing projects written in C, C++, Swift, Objective-C, and other languages. You can also use existing native code, static or dynamic C libraries, Swift/Objective-C frameworks, graphical engines, and anything else directly from Kotlin/Native.

Get started with Kotlin/Native

Target platforms

Kotlin/Native supports the following platforms:

- Linux
- Windows (through MinGW)
- Android NDK
- Apple targets for macOS, iOS, tvOS, and watchOS

> To compile Apple targets, you need to install Xcode and its command-line tools.

See the full list of supported targets and hosts.

Interoperability

Kotlin/Native supports two-way interoperability with native programming languages for different operating systems. The compiler can create executables for many platforms, static or dynamic C libraries, and Swift/Objective-C frameworks.

Interoperability with C

Kotlin/Native provides interoperability with C. You can use existing C libraries directly from Kotlin code.

To learn more, complete the following tutorials:

- Create a dynamic library with C headers for C/C++ projects
- Learn how C types are mapped into Kotlin
- Create a native HTTP client using C interop and libcurl

Interoperability with Swift/Objective-C

Kotlin/Native provides interoperability with Swift through Objective-C. You can use Kotlin code directly from Swift/Objective-C applications on macOS and iOS.

To learn more, complete the Kotlin/Native as an Apple framework tutorial.

Sharing code between platforms

Kotlin/Native includes a set of prebuilt platform libraries that help share Kotlin code between projects. POSIX, gzip, OpenGL, Metal, Foundation, and many other popular libraries and Apple frameworks are pre-imported and included as Kotlin/Native libraries in the compiler package.

Kotlin/Native is a part of the Kotlin Multiplatform technology that helps share common code across multiple platforms, including Android, iOS, JVM, web, and native. Multiplatform libraries provide the necessary APIs for common Kotlin code and allow writing shared parts of projects in Kotlin all in one place.

Memory manager

Kotlin/Native uses an automatic memory manager that is similar to the JVM and Go. It has its own tracing garbage collector, which is also integrated with Swift/Objective-C's ARC.

The memory consumption is controlled by a custom memory allocator. It optimizes memory usage and helps prevent sudden surges in memory allocation.

Get started with Kotlin/Native

In this tutorial, you'll learn how to create a Kotlin/Native application. Choose the tool that works best for you and create your app using:

- The IDE. Here, you can clone the project template from a version control system and use it in IntelliJ IDEA.
- The Gradle build system. To better understand how things work under the hood, create build files for your project manually.

- The command line tool. You can use the Kotlin/Native compiler, which is shipped as a part of the standard Kotlin distribution, and create the app directly in the command line tool.

  Console compilation may seem easy and straightforward, but it doesn't scale well for larger projects with hundreds of files and libraries. For such projects, we recommend using an IDE or a build system.

With Kotlin/Native, you can compile for different targets, including Linux, macOS, and Windows. While cross-platform compilation is possible, which means using one platform to compile for a different one, in this tutorial, you'll be targeting the same platform you're compiling on.

> If you use a Mac and want to create and run applications for macOS or other Apple targets, you also need to install Xcode Command Line Tools, launch it, and accept the license terms first.

In IDE

In this section, you'll learn how to use IntelliJ IDEA to create a Kotlin/Native application.

Create the project

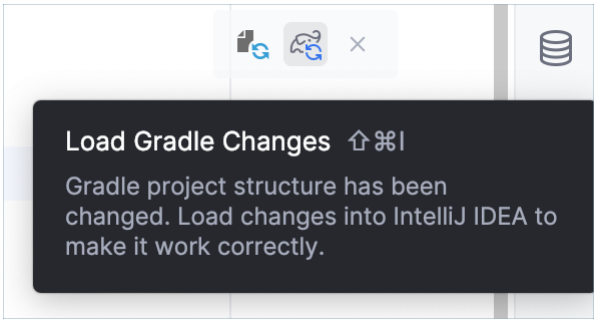
1. Download and install the latest version of IntelliJ IDEA.
2. Clone the project template by selecting **File** | **New** | **Project from Version Control** in IntelliJ IDEA and using this URL:

   ```
   https://github.com/Kotlin/kmp-native-wizard
   ```

3. Open the `gradle/libs.versions.toml` file, which is the version catalog for project dependencies. To create Kotlin/Native applications, you need the Kotlin Multiplatform Gradle plugin, which has the same version as Kotlin. Ensure that you use the latest Kotlin version:

   ```
   [versions]
   kotlin = "2.4.20"
   ```

4. Follow the suggestion to reload Gradle files:



Load Gradle changes button

For more information about these settings, see the Multiplatform Gradle DSL reference.

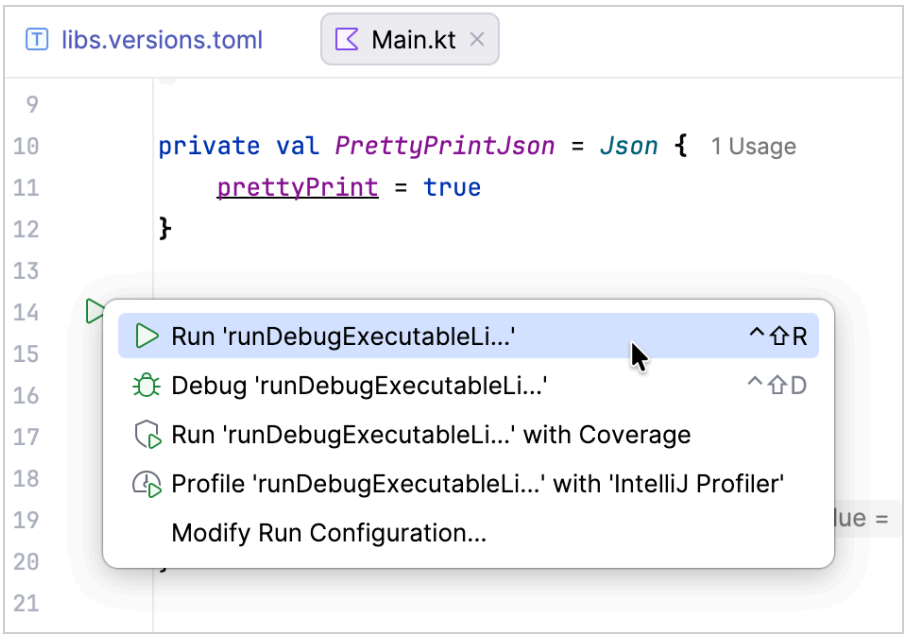
Build and run the application

Open the `Main.kt` file in the `src/nativeMain/kotlin/` directory:

- The `src` directory contains Kotlin source files.

- The `Main.kt` file includes code that prints "Hello, Kotlin/Native!" using the `println()` function.

Press the green icon in the gutter to run the code:



Run the application

IntelliJ IDEA runs the code using the Gradle task and outputs the result in the **Run** tab:

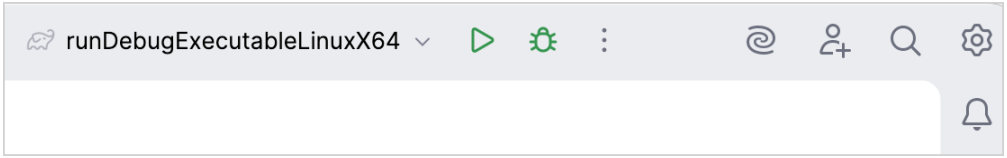
```
> Task :runDebugExecutableMacosArm64
{
    "topic": "Kotlin/Native",
    "content": "Hello!"
}

> Task :compileKotlinMingwX64 UP-TO-DATE
> Task :linkDebugExecutableMingwX64 UP-TO-DATE
> Task :runDebugExecutableMingwX64 SKIPPED

BUILD SUCCESSFUL in 1s
```

Application output

After the first run, the IDE creates the corresponding run configuration at the top:



Gradle run configuration

IntelliJ IDEA users with an Ultimate subscription can install the Native Debugging Support plugin that allows debugging compiled native executables and also automatically creates run configurations for imported Kotlin/Native projects.

You can configure IntelliJ IDEA to build your project automatically:

1. Go to **Settings | Build, Execution, Deployment | Compiler**.
2. On the **Compiler** page, select **Build project automatically**.
3. Apply the changes.

Now, when you make changes in the class files or save the file (Ctrl + S/Cmd + S), IntelliJ IDEA automatically performs an incremental build of the project.

Update the application

Let's add a feature to your application so it can count the number of letters in your name:

1. In the `Main.kt` file, add code to read the input. Use the `readln()` function to read the input value and assign it to the `name` variable:

```
fun main() {
    // Read the input value.
    println("Hello, enter your name:")
    val name = readln()
}
```

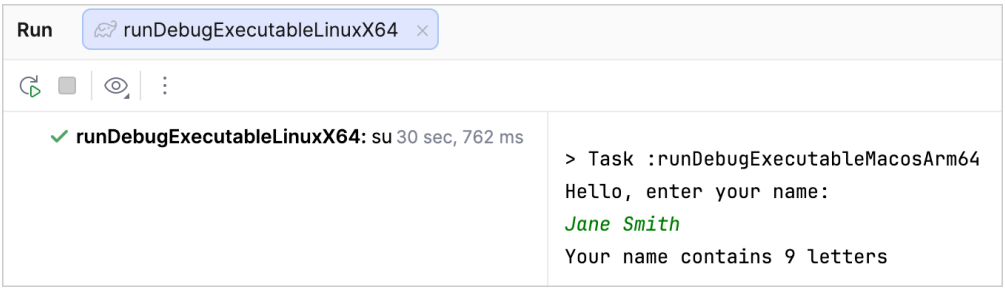
2. To run this app using Gradle, specify `System.in` as the input to use in the `build.gradle.kts` file and load the Gradle changes:

```
kotlin {
    // ...
    targets.withType<KotlinNativeTarget>().configureEach {
        binaries {
            executable {
                entryPoint = "main"
                runTaskProvider?.configure { standardInput = System.`in` }
            }
        }
    }
    // ...
}
```

3. Eliminate the whitespaces and count the letters:
   - Use the `replace()` function to remove the empty spaces in the name.
   - Use the scope function `let` to run the function within the object context.
   - Use a string template to insert your name length into the string by adding a dollar sign `$` and enclosing it in curly braces – `${it.length}`. `it` is the default name of a lambda parameter.

```
fun main() {
    // Read the input value.
    println("Hello, enter your name:")
    val name = readln()
    // Count the letters in the name.
    name.replace(" ", "").let {
        println("Your name contains ${it.length} letters")
    }
}
```

4. Run the application.
5. Enter your name and enjoy the result:



Application output

Now let's count only the unique letters in your name:

1. In the `Main.kt` file, declare the new extension function `.countDistinctCharacters()` for `String`:
   - Convert the name to lowercase using the `.lowercase()` function.
   - Convert the input string to a list of characters using the `toList()` function.
   - Select only the distinct characters in your name using the `distinct()` function.
   - Count the distinct characters using the `count()` function.

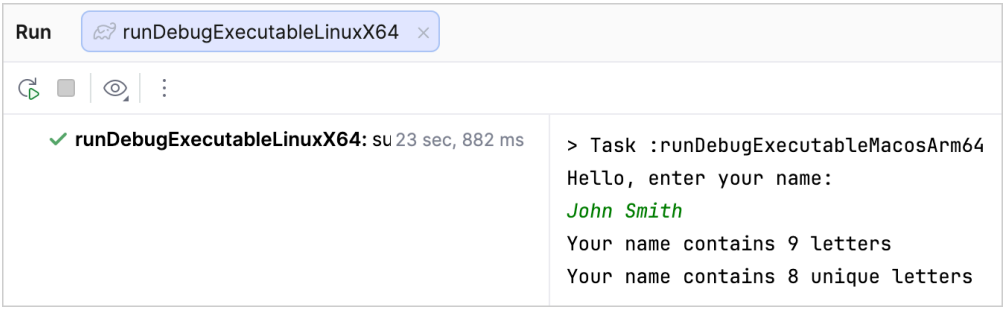
```
fun String.countDistinctCharacters() = lowercase().toList().distinct().count()
```

2. Use the `.countDistinctCharacters()` function to count the unique letters in your name:

```
fun String.countDistinctCharacters() = lowercase().toList().distinct().count()

fun main() {
    // Read the input value.
    println("Hello, enter your name:")
    val name = readln()
    // Count the letters in the name.
    name.replace(" ", "").let {
        println("Your name contains ${it.length} letters")
        // Print the number of unique letters.
        println("Your name contains ${it.countDistinctCharacters()} unique letters")
    }
}
```

3. Run the application.
4. Enter your name and see the result:



Application output

Using Gradle

In this section, you'll learn how to manually create a Kotlin/Native application using Gradle. It's the default build system for Kotlin/Native and Kotlin Multiplatform projects, which is also commonly used in Java, Android, and other ecosystems.

When building Kotlin/Native projects, the Kotlin Gradle plugin downloads the following artifacts:

- The main Kotlin/Native bundle, which includes different tools like `konanc`, `cinterop`, and `jsinterop`. By default, the Kotlin/Native bundle is downloaded from the Maven Central repository as a simple Gradle dependency.
- Dependencies required for `konanc` itself, like `llvm`. They are downloaded from the JetBrains CDN using custom logic.

You can change the source for the main bundle download in the `repositories {}` block in your Gradle build script.

Create project files

1. To get started, install a compatible version of Gradle. See the compatibility table to check the Kotlin Gradle plugin (KGP) compatibility with available Gradle versions.
2. Create an empty project directory. Inside it, create a `build.gradle(.kts)` file with the following content:

Kotlin

```kotlin
// build.gradle.kts
import org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget

plugins {
    kotlin("multiplatform") version "2.4.20"
}

repositories {
    // Specify the source to download the main bundle
    // Maven Central is used by default
    mavenCentral()
}

kotlin {
    macosArm64()    // on macOS
    // linuxArm64() // on Linux
    // mingwX64()   // on Windows

    targets.withType<KotlinNativeTarget>().configureEach {
        binaries {
            executable()
        }
    }
}

tasks.withType<Wrapper> {
    gradleVersion = "9.7.0"
    distributionType = Wrapper.DistributionType.BIN
}
```

Groovy

```
// build.gradle
import org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget

plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.4.20'
}

repositories {
    // Specify the source to download the main bundle
    // Maven Central is used by default
    mavenCentral()
}

kotlin {
    macosArm64()    // on macOS
    // linuxArm64() // on Linux
    // mingwX64()   // on Windows

    targets.withType(KotlinNativeTarget).configureEach {
        binaries {
            executable()
        }
    }
}

wrapper {
    gradleVersion = '9.7.0'
    distributionType = 'BIN'
}
```

You can use different target names, such as `macosArm64`, `iosArm64` `linuxArm64`, and `mingwX64` to define the targets for which you are compiling your code. The target name is used to generate the source paths and task names in the project.

3. Create an empty `settings.gradle(.kts)` file in the project directory.
4. Create a `src/nativeMain/kotlin` directory and place a `hello.kt` file inside with the following content:

```
fun main() {
    println("Hello, Kotlin/Native!")
}
```

By convention, all sources are located in the `src/<platform name>[Main|Test]/kotlin` directories, where `Main` is for the source code and `Test` is for tests. In this case, `<platform name>` is `native`.

Build and run the project

1. From the root project directory, run the `<yourTargetName>Binaries` build command for your target, for example:

```
./gradlew macosArm64Binaries
```

This command creates the `build/bin/<yourTargetName>` directory with two directories inside: `debugExecutable` and `releaseExecutable`. They contain the corresponding binary files.

By default, the name of the binary file is the same as the project directory.

2. To run the project, execute the `build/bin/<yourTargetName>/debugExecutable/<project_name>.kexe` command for your target, for example:

```
build/bin/macosArm64/DebugExecutable/hello.kexe
```

The terminal prints "Hello, Kotlin/Native!".

Open the project in IDE

Now, you can open your project in any IDE that supports Gradle. If you use IntelliJ IDEA:

1. Select **File** | **Open**.
2. Select the project directory and click **Open**. IntelliJ IDEA automatically detects if it's a Kotlin/Native project.

If you encounter a problem with the project, IntelliJ IDEA displays the error message in the **Build** tab.

Using the command-line compiler

In this section, you'll learn how to create a Kotlin/Native application using the Kotlin compiler in the command line tool.

Download and install the compiler

To install the compiler:

1. Go to the Kotlin's GitHub releases page and scroll down to the **Assets** section.
2. Look for a file with `kotlin-native` in the name and download one that is suitable for your operating system, for example `kotlin-native-prebuilt-linux-x86_64-2.4.20.tar.gz`.
3. Unpack the archive to a directory of your choice.
4. Open your shell profile and add the path to the compiler's `/bin` directory to the `PATH` environment variable:

```
export PATH="/<path to the compiler>/kotlin-native/bin:$PATH"
```

Although the compiler output has no dependencies or virtual machine requirements, the compiler itself requires Java 1.8 or higher runtime. It's supported by JDK 8 (JAVA SE 8) or later versions.

Create the program

Choose a working directory and create a file named `hello.kt`. Update it with the following code:

```
fun main() {
    println("Hello, Kotlin/Native!")
}
```

Compile the code from the console

To compile the application, execute the following command with the downloaded compiler:

```
kotlinc-native hello.kt -o hello
```

The value of the `-o` option specifies the name of the output file, so this call generates the `hello.kexe` binary file on macOS and Linux (and `hello.exe` on Windows).

For the full list of available options, see Kotlin compiler options.

Run the program

To run the program, in your command line tool, navigate to the directory containing the binary file and run the following command:

macOS and Linux

```
./hello.kexe
```

Windows

```
./hello.exe
```

The application prints "Hello, Kotlin/Native" to the standard output.

What's next?

- Complete the Create an app using C interop and libcurl tutorial that explains how to create a native HTTP client and interoperate with C libraries.
- Learn how to write Gradle build scripts for real-life Kotlin/Native projects.
- Read more about the Gradle build system in the documentation.

Kotlin/Native libraries

Library compilation

You can use your project's build file or the Kotlin/Native compiler to produce a `*.klib` artifact for your library.

Using Gradle build file

You can compile a `*.klib` library artifact by specifying a Kotlin/Native target in your Gradle build file:

1. In your `build.gradle(.kts)` file, declare at least one Kotlin/Native target. For example:

```
// build.gradle.kts
plugins {
    kotlin("multiplatform") version "2.4.20"
}

kotlin {
    macosArm64()    // on macOS
    // linuxArm64() // on Linux
    // mingwX64()   // on Windows
}
```

2. Run the `<target>Klib` task. For example:

```
./gradlew macosArm64Klib
```

Gradle automatically compiles source files for that target and produces the `.klib` artifact in the project's `build/libs` directory.

Using Kotlin/Native compiler

To produce a library with the Kotlin/Native compiler:

1. Download and install the Kotlin/Native compiler.
2. To compile a Kotlin/Native source file into a library, use the `-produce library` or `-p library` option:

   ```
   kotlinc-native foo.kt -p library -o bar
   ```

   This command compiles the contents of the `foo.kt` file into a library with the name `bar`, producing a `bar.klib` artifact.
3. To link another file to a library, use the `-library <name>` or `-l <name>` option. For example:

   ```
   kotlinc-native qux.kt -l bar
   ```

   This command compiles the contents of the `qux.kt` source file and the `bar.klib` library and produces the `program.kexe` final executable binary.

klib utility

The **klib** library management utility allows you to inspect libraries using the following syntax:

```
klib <command> <library path> [<option>]
```

The following commands are currently available:

Command	Description
`info`	General information about the library.

Command	Description
`dump-abi`	Dump the ABI snapshot of the library. Each line in the snapshot corresponds to one declaration. In case an ABI-incompatible change happens to a declaration, it'll be visible in the corresponding line of the snapshot.
`dump-ir`	Dump the intermediate representation (IR) of library declarations to the output. Use it only for debugging.
`dump-ir-signatures`	Dump IR signatures of all non-private library declarations and all non-private declarations consumed by this library (as two separate lists). This command relies purely on the data in IR.
`dump-ir-inlinable-functions`	Dump the IR of inlinable functions in the library to the output. Use it only for debugging.
`dump-metadata`	Dump the metadata of all library declarations to the output. Use it only for debugging.
`dump-metadata-signatures`	Dump IR signatures of all non-private library declarations based on the library metadata. In most cases, the output is the same as for the `dump-ir-signatures` command, which renders signatures based on IR. However, if IR-transforming compiler plugins (such as Compose) are used during compilation, patched declarations may have different signatures.

All the above dumping commands accept an additional `-signature-version {N}` argument that instructs the klib utility which IR signature version to render when dumping signatures. If not provided, it uses the most up-to-date version supported by the library. For example:

```
klib dump-metadata-signatures mylib.klib -signature-version 1
```

In addition, the `dump-metadata` command accepts the `-print-signatures {true|false}` argument that instructs the klib utility to print the IR signatures for every declaration in the output.

Creating and using a library

1. Create a library by placing the source code into `kotlinizer.kt`:

   ```
   package kotlinizer

   val String.kotlinized
       get() = "Kotlin $this"
   ```

2. Compile the library into a `.klib`:

   ```
   kotlinc-native kotlinizer.kt -p library -o kotlinizer
   ```

3. Check the current directory for the created library:

```
ls kotlinizer.klib
```

4. Check out general information about the library:

```
klib info kotlinizer.klib
```

5. Create a short program in the `use.kt` file:

```
import kotlinizer.*

fun main(args: Array<String>) {
    println("Hello, ${"world".kotlinized}!")
}
```

6. Compile the program, linking the `use.kt` source file to your library:

```
kotlinc-native use.kt -l kotlinizer -o kohello
```

7. Run the program:

```
./kohello.kexe
```

You should see `Hello, Kotlin world!` in the output.

Library search sequence

> The library search mechanism will change soon. Expect updates to this section and avoid relying on deprecated flags.

When given a `-library foo` option, the compiler searches the `foo` library in the following order:

1. Current compilation directory or an absolute path.
2. Libraries installed in the default repository.

> The default repository is `~/.konan`. You can change it by setting the `konan.data.dir` Gradle property.
>
> Alternatively, you can use the `-Xkonan-data-dir` compiler option to configure your custom path to the directory via the `cinterop` and `konanc` tools.

3. Libraries installed in the `$installation/klib` directory.

Library format

Kotlin/Native libraries are zip files containing a predefined directory structure, with the following layout:

`foo.klib` when unpacked as `foo/` gives us:

```
- foo/
  - $component_name/
    - ir/
      - Serialized Kotlin IR.
    - targets/
      - $platform/
        - kotlin/
          - Kotlin compiled to LLVM bitcode.
        - native/
          - Bitcode files of additional native objects.
      - $another_platform/
        - There can be several platform specific kotlin and native pairs.
    - linkdata/
      - A set of ProtoBuf files with serialized linkage metadata.
    - resources/
      - General resources such as images. (Not used yet).
    - manifest - A file in the java property format describing the library.
```

You can find an example layout in the `klib/common/stdlib` directory of your Kotlin/Native compiler installation.

Using relative paths in klibs

A serialized IR representation of source files is a part of a `klib` library. It includes paths of files for generating proper debug information. By default, stored paths are absolute.

With the `-Xklib-relative-path-base` compiler option, you can change the format and use only relative paths in the artifact. To make it work, pass one or multiple base paths of source files as an argument:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named<KotlinCompilationTask<*>>("compileKotlin").configure {
    // $base is a base path of source files
    compilerOptions.freeCompilerArgs.add("-Xklib-relative-path-base=$base")
}
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        // $base is a base path of source files
        freeCompilerArgs.add("-Xklib-relative-path-base=$base")
    }
}
```

What's next?

Learn how to use the cinterop tool to produce `*.klib` artifacts

Platform libraries

To provide access to native services of operating systems, the Kotlin/Native distribution includes a set of prebuilt libraries specific to each target. These are called *platform libraries*.

The packages from platform libraries are available by default. You don't need to specify additional link options to use them. The Kotlin/Native compiler automatically detects which platform libraries are accessed and links the necessary ones.

However, platform libraries in the compiler distribution are merely wrappers and bindings to the native libraries. That means you need to install native libraries themselves (`.so` , `.a` , `.dylib` , `.dll` , and so on) on your local machine.

POSIX bindings

Kotlin provides the POSIX platform library for all UNIX- and Windows-based targets, including Android and iOS. These platform libraries contain bindings to the platform's implementation, which follows the POSIX standard.

To use the library, import it into your project:

```
import platform.posix.*
```

The `platform.posix` contents differ across platforms because of variations in POSIX implementations.

You can explore the contents of the `posix.def` file for each supported platform here:

- iOS
- macOS
- tvOS
- watchOS
- Linux
- Windows (MinGW)
- Android

The POSIX platform library is not available for the WebAssembly target.

Popular native libraries

Kotlin/Native provides bindings for various popular native libraries that are commonly used on different platforms, such as OpenGL, zlib, and Foundation.

On Apple platforms, the `objc` library is included to enable interoperability with Objective-C APIs.

You can explore the native libraries available for Kotlin/Native targets in your compiler distribution, depending on your setup:

- If you installed a standalone Kotlin/Native compiler:
  1. Go to the unpacked archive with the compiler distribution, for example, `kotlin-native-prebuilt-macos-aarch64-2.1.0` .

2. Navigate to the `klib/platform` directory.
3. Choose the folder with the corresponding target.

- If you use the Kotlin plugin in your IDE (bundled with IntelliJ IDEA and Android Studio):
  1. In your command line tool, run the following to navigate to the `.konan` folder:

     macOS and Linux

     ```
     ~/.konan/
     ```

     Windows

     ```
     %USERPROFILE%\.konan
     ```

  2. Open the Kotlin/Native compiler distribution, for example, `kotlin-native-prebuilt-macos-aarch64-2.1.0`.
  3. Navigate to the `klib/platform` directory.
  4. Choose the folder with the corresponding target.

> If you'd like to explore the definition file for each supported platform library: in the compiler distribution folder, navigate to the `konan/platformDef` directory and choose the necessary target.

What's next

Learn more about interoperability with Swift/Objective-C

Kotlin/Native memory management

Kotlin/Native uses a modern memory manager that is similar to the JVM, Go, and other mainstream technologies, including the following features:

- Objects are stored in a shared heap and can be accessed from any thread.
- Tracing garbage collection is performed periodically to collect objects that are not reachable from the "roots", like local and global variables.

Garbage collector

Kotlin/Native's garbage collector (GC) algorithm is constantly evolving. Currently, it functions as a concurrent mark and sweep (CMS) collector that does not separate the heap into generations.

The GC is executed on a separate thread and started based on the memory pressure heuristics or by a timer. Alternatively, it can be called manually.

The GC processes the mark queue on several threads in parallel, including application threads, the GC thread, and optional marker threads. Application threads and at least one GC thread participate in the marking process. By

default, the marking phase runs concurrently with application threads, which reduces the GC pause time. You can monitor GC performance through the GC logs.

> You can disable the parallelization of the mark phase with the `kotlin.native.binary.gcMarkSingleThreaded=true` compiler option. However, this may increase the garbage collector's pause time on large heaps.

When the marking phase is completed, the GC processes weak references and nullifies reference points to an unmarked object. By default, weak references are processed concurrently to decrease the GC pause time.

If you encounter problems with CMS, switch back to the parallel mark concurrent sweep (PMCS) setup. To do that, set the following binary option in your `gradle.properties` file:

```
kotlin.native.binary.gc=pmcs
```

Enable garbage collection manually

To force-start the garbage collector, call `kotlin.native.internal.GC.collect()`. This method triggers a new collection and waits for its completion.

Monitor GC performance

To monitor the GC performance, you can look through its logs and diagnose issues. To enable logging, set the following compiler option in your Gradle build script:

```
-Xruntime-logs=gc=info
```

Currently, the logs are only printed to `stderr`.

On Apple platforms, you can take advantage of the Xcode Instruments toolkit to debug iOS app performance. The garbage collector reports pauses with signposts available in Instruments. Signposts enable custom logging within your app, allowing you to check if a GC pause corresponds to an application freeze.

To track GC-related pauses in your app:

1. To enable the feature, set the following compiler option in your `gradle.properties` file:

   ```
   kotlin.native.binary.enableSafepointSignposts=true
   ```

2. Open Xcode, go to **Product** | **Profile** or press Cmd + I. This action compiles your app and launches Instruments.
3. In the template selection, select **os_signpost**.
4. Configure it by specifying `org.kotlinlang.native.runtime` as **subsystem** and `safepoint` as **category**.
5. Click the red record button to run your app and start recording signpost events:



Tracking GC pauses as signposts

Here, each blue blob on the lowest graph represents a separate signpost event, which is a GC pause.

Disable garbage collection

It's recommended to keep the GC enabled. However, you can disable it in certain cases, such as for testing purposes or if you encounter issues and have a short-lived program. To do so, set the following binary option in your `gradle.properties` file:

```
kotlin.native.binary.gc=noop
```

With this option enabled, the GC doesn't collect Kotlin objects, so memory consumption will keep rising as long as the program runs. Be careful not to exhaust the system memory.

Memory consumption

Kotlin/Native uses its own memory allocator. It divides system memory into pages, allowing independent sweeping in consecutive order. Each allocation becomes a memory block within a page, and the page keeps track

of block sizes. Different page types are optimized for various allocation sizes. The consecutive arrangement of memory blocks ensures efficient iteration through all allocated blocks.

When a thread allocates memory, it searches for a suitable page based on the allocation size. Threads maintain a set of pages for different size categories. Typically, the current page for a given size can accommodate the allocation. If not, the thread requests a different page from the shared allocation space. This page may already be available, require sweeping, or have to be created first.

The Kotlin/Native memory allocator comes with protection against sudden spikes in memory allocations. It prevents situations where the mutator starts to allocate a lot of garbage quickly and the GC thread cannot keep up with it, making the memory usage grow endlessly. In this case, the GC forces a stop-the-world phase until the iteration is completed.

You can monitor memory consumption yourself, check for memory leaks, and adjust memory consumption.

Monitor memory consumption

To debug memory issues, you can check memory manager metrics. In addition, it's possible to track Kotlin's memory consumption on Apple platforms.

Check for memory leaks

To access the memory manager metrics, call `kotlin.native.internal.GC.lastGCInfo()`. This method returns statistics for the last run of the garbage collector. The statistics can be useful for:

- Debugging memory leaks when using global variables
- Checking for leaks when running tests

```
import kotlin.native.internal.*
import kotlin.test.*

class Resource

val global = mutableListOf<Resource>()

@OptIn(ExperimentalStdlibApi::class)
fun getUsage(): Long {
    GC.collect()
    return GC.lastGCInfo!!.memoryUsageAfter["heap"]!!.totalObjectsSizeBytes
}

fun run() {
    global.add(Resource())
    // The test will fail if you remove the next line
    global.clear()
}

@Test
fun test() {
    val before = getUsage()
    // A separate function is used to ensure that all temporary objects are cleared
    run()
    val after = getUsage()
    assertEquals(before, after)
}
```

Track memory consumption on Apple platforms

When debugging memory issues on Apple platforms, you can see how much memory is reserved by Kotlin code. Kotlin's share is tagged with an identifier and can be tracked through tools like VM Tracker in Xcode Instruments.

The feature is available only for the default Kotlin/Native memory allocator when *all* the following conditions are met:

- **Tagging enabled**. The memory should be tagged with a valid identifier. Apple recommends numbers between 240 and 255; the default value is 246.

  If you set up the `kotlin.native.binary.mmapTag=0` Gradle property, tagging is disabled.

- **Allocation with mmap**. The allocator should use the `mmap` system call to map files into memory.

  If you set up the `kotlin.native.binary.disableMmap=true` Gradle property, the default allocator uses `malloc` instead of `mmap`.

- **Paging enabled**. Paging of allocations (buffering) should be enabled.

  If you set up the `kotlin.native.binary.pagedAllocator=false` Gradle property, the memory is reserved on a per-object basis instead.

Adjust memory consumption

If you experience unexpectedly high memory consumption, try the following solutions:

Update Kotlin

Update Kotlin to the latest version. We're constantly improving the memory manager, so even a simple compiler update might improve memory consumption.

Experimental

Disable allocator paging

You can disable paging of allocations (buffering) so that the memory allocator reserves memory on a per-object basis. In some cases, it may help you satisfy strict memory limitations or reduce memory consumption on the application's startup.

To do that, set the following option in your `gradle.properties` file:

```
kotlin.native.binary.pagedAllocator=false
```

With allocator paging disabled, tracking memory consumption on Apple platforms is not possible.

Experimental

Enable support for Latin-1 strings

By default, strings in Kotlin are stored using UTF-16 encoding, where each character is represented by two bytes. In some cases, it leads to strings taking up twice as much space in the binary compared to the source code and reading data taking up twice as much memory.

To reduce the application's binary size and adjust memory consumption, you can enable support for Latin-1-encoded strings. The Latin-1 (ISO 8859-1) encoding represents each of the first 256 Unicode characters by just one byte.

To enable it, set the following option in your `gradle.properties` file:

```
kotlin.native.binary.latin1Strings=true
```

With the Latin-1 support, strings are stored in Latin-1 encoding as long as all the characters fall within its range. Otherwise, the default UTF-16 encoding is used.

> While the feature is Experimental, the cinterop extension functions `String.pin`, `String.usePinned`, and `String.refTo` become less efficient. Each call to them may trigger an automatic string conversion to UTF-16.

If none of these options helped, create an issue in YouTrack.

Unit tests in the background

In unit tests, nothing processes the main thread queue, so don't use `Dispatchers.Main` unless it was mocked. Mocking it can be done by calling `Dispatchers.setMain` from `kotlinx-coroutines-test`.

If you don't rely on `kotlinx.coroutines` or if `Dispatchers.setMain` doesn't work for you for some reason, try the following workaround for implementing the test launcher:

```
package testlauncher

import platform.CoreFoundation.*
import kotlin.native.concurrent.*
import kotlin.native.internal.test.*
import kotlin.system.*

fun mainBackground(args: Array<String>) {
    val worker = Worker.start(name = "main-background")
    worker.execute(TransferMode.SAFE, { args.freeze() }) {
        val result = testLauncherEntryPoint(it)
        exitProcess(result)
    }
    CFRunLoopRun()
    error("CFRunLoopRun should never return")
}
```

Then, compile the test binary with the `-e testlauncher.mainBackground` compiler option.

What's next

- Migrate from the legacy memory manager
- Check the specifics of integration with Swift/Objective-C ARC

Integration with Swift/Objective-C ARC

Kotlin and Objective-C use different memory management strategies. Kotlin has a tracing garbage collector, while Objective-C relies on automatic reference counting (ARC).

The integration between these strategies is usually seamless and generally requires no additional work. However, there are some specifics you should keep in mind:

Threads

Deinitializers

Deinitialization on the Swift/Objective-C objects and the objects they refer to is called on the main thread if these objects are passed to Kotlin on the main thread, for example:

```
// Kotlin
class KotlinExample {
    fun action(arg: Any) {
        println(arg)
    }
}
```

```
// Swift
class SwiftExample {
    init() {
        print("init on \(Thread.current)")
    }

    deinit {
        print("deinit on \(Thread.current)")
    }
}

func test() {
    KotlinExample().action(arg: SwiftExample())
}
```

The resulting output:

```
init on <_NSMainThread: 0x600003bc0000>{number = 1, name = main}
shared.SwiftExample
deinit on <_NSMainThread: 0x600003bc0000>{number = 1, name = main}
```

Deinitialization on the Swift/Objective-C objects is called on a special GC thread instead of the main one if:

- Swift/Objective-C objects are passed to Kotlin on a thread other than main.
- The main dispatch queue isn't processed.

If you want to call deinitialization on a special GC thread explicitly, set `kotlin.native.binary.objcDisposeOnMain=false` in your `gradle.properties`. This option enables deinitialization on a special GC thread, even if Swift/Objective-C objects were passed to Kotlin on the main thread.

A special GC thread complies with the Objective-C runtime, meaning that it has a run loop and drain autorelease pools.

Completion handlers

When calling Kotlin suspending functions from Swift, completion handlers might be called on threads other than the main one, for example:

```kotlin
// Kotlin
// coroutineScope, launch, and delay are from kotlinx.coroutines
suspend fun asyncFunctionExample() = coroutineScope {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
```

```swift
// Swift
func test() {
    print("Running test on \(Thread.current)")
    PlatformKt.asyncFunctionExample(completionHandler: { _ in
        print("Running completion handler on \(Thread.current)")
    })
}
```

The resulting output:

```
Running test on <_NSMainThread: 0x600001b100c0>{number = 1, name = main}
Hello
World!
Running completion handler on <NSThread: 0x600001b45bc0>{number = 7, name = (null)}
```

Garbage collection and lifecycle

Object reclamation

An object is reclaimed only during the garbage collection. This applies to Swift/Objective-C objects that cross interop boundaries into Kotlin/Native, for example:

```kotlin
// Kotlin
class KotlinExample {
    fun action(arg: Any) {
        println(arg)
    }
}
```

```
// Swift
class SwiftExample {
    deinit {
        print("SwiftExample deinit")
    }
}

func test() {
    swiftTest()
    kotlinTest()
}

func swiftTest() {
    print(SwiftExample())
    print("swiftTestFinished")
}

func kotlinTest() {
    KotlinExample().action(arg: SwiftExample())
    print("kotlinTest finished")
}
```

The resulting output:

```
shared.SwiftExample
SwiftExample deinit
swiftTestFinished
shared.SwiftExample
kotlinTest finished
SwiftExample deinit
```

Objective-C objects lifecycle

The Objective-C objects might live longer than they should, which sometimes might cause performance issues. For example, when a long-running loop creates several temporary objects that cross the Swift/Objective-C interop boundary on each iteration.

In the GC logs, there's a number of stable refs in the root set. If this number keeps growing, it may indicate that the Swift/Objective-C objects are not freed up when they should. In this case, try the `autoreleasepool` block around loop bodies that do interop calls:

```
// Kotlin
fun growingMemoryUsage() {
    repeat(Int.MAX_VALUE) {
        NSLog("$it\n")
    }
}

fun steadyMemoryUsage() {
    repeat(Int.MAX_VALUE) {
        autoreleasepool {
            NSLog("$it\n")
        }
    }
}
```

Garbage collection of Swift and Kotlin objects' chains

Consider the following example:

```
// Kotlin
interface Storage {
    fun store(arg: Any)
}

class KotlinStorage(var field: Any? = null) : Storage {
    override fun store(arg: Any) {
        field = arg
    }
}

class KotlinExample {
    fun action(firstSwiftStorage: Storage, secondSwiftStorage: Storage) {
        // Here, we create the following chain:
        // firstKotlinStorage -> firstSwiftStorage -> secondKotlinStorage -> secondSwiftStorage.
        val firstKotlinStorage = KotlinStorage()
        firstKotlinStorage.store(firstSwiftStorage)
        val secondKotlinStorage = KotlinStorage()
        firstSwiftStorage.store(secondKotlinStorage)
        secondKotlinStorage.store(secondSwiftStorage)
    }
}
```

```
// Swift
class SwiftStorage : Storage {

    let name: String

    var field: Any? = nil

    init(_ name: String) {
        self.name = name
    }

    func store(arg: Any) {
        field = arg
    }

    deinit {
        print("deinit SwiftStorage \(name)")
    }
}

func test() {
    KotlinExample().action(
        firstSwiftStorage: SwiftStorage("first"),
        secondSwiftStorage: SwiftStorage("second")
    )
}
```

It takes some time between "deinit SwiftStorage first" and "deinit SwiftStorage second" messages to appear in the log. The reason is that `firstKotlinStorage` and `secondKotlinStorage` are collected in different GC cycles. Here's the sequence of events:

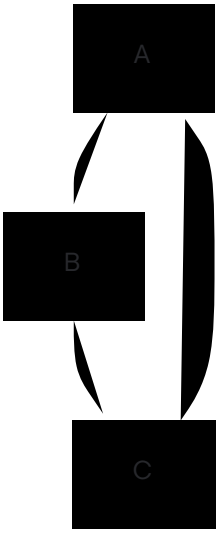
1. `KotlinExample.action` finishes. `firstKotlinStorage` is considered "dead" because nothing references it, while `secondKotlinStorage` is not because it is referenced by `firstSwiftStorage`.

2. First GC cycle starts, and `firstKotlinStorage` is collected.
3. There are no references to `firstSwiftStorage`, so it is "dead" as well, and `deinit` is called.
4. Second GC cycle starts. `secondKotlinStorage` is collected because `firstSwiftStorage` is no longer referencing it.
5. `secondSwiftStorage` is finally reclaimed.

It requires two GC cycles to collect these four objects because deinitialization of Swift and Objective-C objects happens after the GC cycle. The limitation stems from `deinit`, which can call arbitrary code, including the Kotlin code that cannot be run during the GC pause.

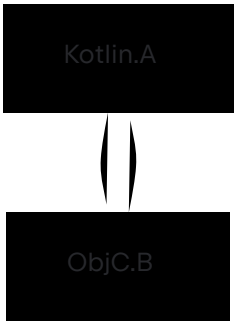
Retain cycles

In a *retain cycle*, a number of objects refer each other using strong references cyclically:



Kotlin's tracing GC and Objective-C's ARC handle retain cycles differently. When objects become unreachable, Kotlin's GC can properly reclaim such cycles, while Objective-C's ARC cannot. Therefore, retain cycles of Kotlin objects can be reclaimed, while retain cycles of Swift/Objective-C objects cannot.

Consider the case when a retain cycle contains both Objective-C and Kotlin objects:



This involves combining Kotlin's and Objective-C's memory management models that cannot handle (reclaim) retain cycles together. That means if at least one Objective-C object is present, the retain cycle of a whole graph of objects cannot be reclaimed, and it's impossible to break the cycle from the Kotlin side.

Unfortunately, no special instruments are currently available to automatically detect retain cycles in Kotlin/Native code. To avoid retain cycles, use weak or unowned references.

Support for background state and App Extensions

The current memory manager does not track application state by default and does not integrate with App Extensions out of the box.

It means that the memory manager doesn't adjust GC behavior accordingly, which might be harmful in some cases. To change this behavior, add the following Experimental binary option to your `gradle.properties`:

```
kotlin.native.binary.appStateTracking=enabled
```

It turns off a timer-based invocation of the garbage collector when the application is in the background, so GC is called only when memory consumption becomes too high.

What's next

Learn more about Swift/Objective-C interoperability.

Migrate to the new memory manager

> Support for the legacy memory manager has been completely removed in Kotlin 1.9.20. Migrate your projects to the current memory model, enabled by default since Kotlin 1.7.20.

This guide compares the new Kotlin/Native memory manager with the legacy one and describes how to migrate your projects.

The most noticeable change in the new memory manager is lifting restrictions on object sharing. You don't need to freeze objects to share them between threads, specifically:

- Top-level properties can be accessed and modified by any thread without using `@SharedImmutable`.
- Objects passing through interop can be accessed and modified by any thread without freezing them.
- `Worker.executeAfter` no longer requires operations to be frozen.
- `Worker.execute` no longer requires producers to return an isolated object subgraph.
- Reference cycles containing `AtomicReference` and `FreezableAtomicReference` do not cause memory leaks.

Apart from easy object sharing, the new memory manager also brings other major changes:

- Global properties are initialized lazily when the file they are defined in is accessed first. Previously, global properties were initialized at the program startup. As a workaround, you can mark properties that must be initialized at the program start with the `@EagerInitialization` annotation. Before using, check its documentation.
- `by lazy {}` properties support thread-safety modes and do not handle unbounded recursion.
- Exceptions that escape `operation` in `Worker.executeAfter` are processed like in other runtime parts, by trying to execute a user-defined unhandled exception hook or terminating the program if the hook was not found or failed with an exception itself.
- Freezing is deprecated and always disabled.

Follow these guidelines to migrate your projects from the legacy memory manager:

Update Kotlin

The new Kotlin/Native memory manager has been enabled by default since Kotlin 1.7.20. Check the Kotlin version and update to the latest one if necessary.

Update dependencies

kotlinx.coroutines

Update to version 1.6.0 or later. Do not use versions with the `native-mt` suffix.

There are also some specifics with the new memory manager you should keep in mind:

- Every common primitive (channels, flows, coroutines) works through the Worker boundaries, since freezing is not required.
- `Dispatchers.Default` is backed by a pool of Workers on Linux and Windows and by a global queue on Apple targets.
- Use `newSingleThreadContext` to create a coroutine dispatcher that is backed by a Worker.
- Use `newFixedThreadPoolContext` to create a coroutine dispatcher backed by a pool of `N` Workers.
- `Dispatchers.Main` is backed by the main queue on Darwin and by a standalone Worker on other platforms.

Ktor

Update to version 2.0 or later.

Other dependencies

The majority of libraries should work without any changes, however, there might be exceptions.

Make sure that you update dependencies to the latest versions, and there is no difference between library versions for the legacy and the new memory manager.

Update your code

To support the new memory manager, remove usages of the affected API:

Old API	What to do
`@SharedImmutable`	You can remove all usages, though there are no warnings for using this API in the new memory manager.
The `FreezableAtomicReference` class	Use `AtomicReference` instead.
The `FreezingException` class	Remove all usages.
The `InvalidMutabilityException` class	Remove all usages.
The `IncorrectDereferenceException` class	Remove all usages.

Old API	What to do
The `freeze()` function	Remove all usages.
The `isFrozen` property	You can remove all usages. Since freezing is deprecated, the property always returns `false`.
The `ensureNeverFrozen()` function	Remove all usages.
The `atomicLazy()` function	Use `lazy()` instead.
The `MutableData` class	Use any regular collection instead.
The `WorkerBoundReference<out T : Any>` class	Use `T` directly.
The `DetachedObjectGraph<T>` class	Use `T` directly. To pass the value through the C interop, use the StableRef class.

What's next

- Learn more about the new memory manager
- Check the specifics of integration with Swift/Objective-C ARC
- Learn how to reference objects safely from different coroutines

Kotlin/Native binary options

This page lists helpful Kotlin/Native binary options that you can use to configure Kotlin/Native final binaries and the ways to set up binary options in your project.

How to enable

You can enable binary options in the `gradle.properties` file, your build file, or pass them as compiler arguments.

In Gradle properties

You can set binary options in your project's `gradle.properties` file using the `kotlin.native.binary` property. For example:

```
kotlin.native.binary.latin1Strings=true
```

In your build file

You can set binary options for your project in your `build.gradle.kts` file:

-

For specific binaries using the `binaryOption` attribute. For example:

```
kotlin {
    iosArm64 {
        binaries {
            framework {
                binaryOption("smallBinary", "true")
            }
        }
    }
}
```

- As `-Xbinary=$option=$value` compiler options in the `freeCompilerArgs` attribute. For example:

```
kotlin {
    iosArm64 {
        compilations.configureEach {
            compilerOptions.configure {
                freeCompilerArgs.add("-Xbinary=smallBinary=true")
            }
        }
    }
}
```

In the command-line compiler

You can pass binary options as `-Xbinary=$option=$value` directly in the command line when executing the Kotlin/Native compiler. For example:

```
kotlinc-native main.kt -Xbinary=enableSafepointSignposts=true
```

Binary options

This table is not an exhaustive list of all the existing options, only the most notable ones.

Option	Values	Description	Status
`enableReleaseBinaryCache`	• `true` • `false` (default)	Enables the compiler to use caches in release mode, making builds faster at the expense of some compiler optimizations.	Experimental since 2.4.20
`objcExportBlockExplicitParameterNames`	• `true` • `false (default)`	Adds explicit parameter names to function types for exported Objective-C headers.	Experimental since 2.2.20

Option	Values	Description	Status
`smallBinary`	• `true` • `false` (default)	Decreases the binary size for release binaries.	Experimental since 2.2.20
`stackProtector`	• `yes` • `strong` • `all` • `no` (default)	Enables stack canaries: use `yes` for vulnerable functions, `all` for all functions, and `strong` to use a stronger heuristic.	Available since 2.2.20
`pagedAllocator`	• `true` (default) • `false`	Controls paging of allocations (buffering). When `false`, the memory allocator reserves memory on a per-object basis.	Experimental since 2.2.0
`latin1Strings`	• `true` • `false` (default)	Controls support for Latin-1-encoded strings to reduce application binary size and adjust memory consumption.	Experimental since 2.2.0
`mmapTag`	`UInt`	Controls memory tagging, necessary for memory consumption tracking on Apple platforms. Values `240` - `255` are available (default is `246`); `0` disables tagging.	Available since 2.2.0
`disableMmap`	• `true` • `false` (default)	Controls the default allocator. When `true`, uses the `malloc` memory allocator instead of `mmap`.	Available since 2.2.0
`gc`	• `cms` (default) • `pmcs` • `stwms`	Controls garbage collection behavior: •	`cms` is default since 2.4.0

Option	Values	Description	Status
	• `noop` • `cms` uses concurrent mark and sweep • `pmcs` uses parallel mark concurrent sweep • `stwms` uses simple stop-the-world mark and sweep • `noop` disables garbage collection		
`gcMarkSingleThreaded`	• `true` • `false` (default)	Disables parallelization of the mark phase in garbage collection. May increase GC pause time on large heaps.	Available since 1.7.20
`enableSafepointSignposts`	• `true` • `false` (default)	Enables tracking GC-related pauses in the project for debugging in Xcode Instruments.	Available since 2.0.20
`preCodegenInlineThreshold`	`UInt`	Configures inlining optimization pass in the Kotlin IR compiler, which comes before the actual code generation phase (disabled by default). The recommended number of tokens (code units parsed by the compiler) is 40.	Experimental since 2.1.20
`objcDisposeOnMain`	• `true` (default) • `false`	Controls deinitialization of Swift/Objective-C objects. When `false`, deinitialization happens on a special GC thread instead of the main one.	Available since 1.9.0
`appStateTracking`	• `enabled` •	Controls timer-based invocation of the garbage collector when	Experimental since 1.7.20

Option	Values	Description	Status
	`disabled` (default)	the application runs in the background. When `enabled`, GC is called only when memory consumption becomes too high.	
`bundleId`	• `String`	Sets bundle ID (`CFBundleIdentifier`) in the `Info.plst` file.	Available since 1.7.20
`bundleShortVersionString`	• `String`	Sets short bundle version (`CFBundleShortVersionString`) in the `Info.plst` file.	Available since 1.7.20
`bundleVersion`	• `String`	Sets bundle version (`CFBundleVersion`) in the `Info.plst` file.	Available since 1.7.20
`sourceInfoType`	• `libbacktrace` • `coresymbolication` (Apple targets) • `noop` (default)	Adds file locations and line numbers to exception stack traces. `coresymbolication` is only available for Apple targets and enabled by default for macOS and Apple simulators in debug mode.	Experimental since 1.6.20

For more information on stability levels, see the documentation.

What's next

Learn how to build final native binaries.

Debugging Kotlin/Native

The Kotlin/Native compiler can generate binaries with debug information, as well as create debug symbol files for symbolicating crash reports.

The debug information is compatible with the DWARF 2 specification, so modern debugger tools, like LLDB and GDB can:

- Set breakpoints
- Use stepping
- Inspect variable and type information

> Supporting the DWARF 2 specification means that the debugger tool recognizes Kotlin as C89, because before the DWARF 5 specification, there is no identifier for the Kotlin language type in the specification.

Generate binaries with debug information

When debugging in IntelliJ IDEA, Android Studio, or Xcode, binaries with debug information are generated automatically (unless the build is configured differently).

You can enable debugging manually and produce binaries that include debug information by:

- **Using Gradle tasks**. To get debug binaries, use `linkDebug*` Gradle tasks, for example:

  ```
  ./gradlew linkDebugFrameworkNative
  ```

  The tasks differ depending on the binary type (for example, `linkDebugSharedNative`) or your target (for example, `linkDebugExecutableMacosArm64`).

- **Using the command-line compiler**. In the command line, compile your Kotlin/Native binary with the `-g` option:

  ```
  kotlinc-native hello.kt -g -o terminator
  ```

Then launch your debugger tool. For example:

```
lldb terminator.kexe
```

The debugger outputs:

```
$ cat - > hello.kt
fun main(args: Array<String>) {
  println("Hello world")
  println("I need your clothes, your boots and your motorcycle")
}
$ dist/bin/konanc -g hello.kt -o terminator
KtFile: hello.kt
$ lldb terminator.kexe
(lldb) target create "terminator.kexe"
Current executable set to 'terminator.kexe' (x86_64).
(lldb) b kfun:main(kotlin.Array<kotlin.String>)
Breakpoint 1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at hello.kt:2,
address = 0x00000001000012e4
(lldb) r
Process 28473 launched: '/Users/minamoto/ws/.git-trees/debugger-fixes/terminator.kexe' (x86_64)
Process 28473 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00000001000012e4 terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) at hello.kt:2
   1    fun main(args: Array<String>) {
-> 2      println("Hello world")
   3      println("I need your clothes, your boots and your motorcycle")
   4    }
(lldb) n
Hello world
Process 28473 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x00000001000012f0 terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) at hello.kt:3
   1    fun main(args: Array<String>) {
   2      println("Hello world")
-> 3      println("I need your clothes, your boots and your motorcycle")
   4    }
(lldb)
```

Set breakpoints

Modern debuggers provide several ways to set a breakpoint. See below for a tool-by-tool breakdown:

LLDB

- By name:

```
(lldb) b -n kfun:main(kotlin.Array<kotlin.String>)
Breakpoint 4: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at hello.kt:2,
address = 0x00000001000012e4
```

  `-n` is optional, it's applied by default.

- By location (filename, line number):

```
(lldb) b -f hello.kt -l 1
Breakpoint 1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at hello.kt:2,
address = 0x00000001000012e4
```

- By address:

```
(lldb) b -a 0x00000001000012e4
Breakpoint 2: address = 0x00000001000012e4
```

- By regex. You might find it useful for debugging generated artifacts, like a lambda (with the `#` symbol in the name):

```
(lldb) b -r main\(
3: regex = 'main\(', locations = 1
  3.1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at hello.kt:2, address =
terminator.kexe[0x00000001000012e4], unresolved, hit count = 0
```

GDB

- By regex:

```
(gdb) rbreak main(
Breakpoint 1 at 0x1000109b4
struct ktype:kotlin.Unit &kfun:main(kotlin.Array<kotlin.String>);
```

- By name is *not* possible because `:` is a separator for the breakpoint by location:

```
(gdb) b kfun:main(kotlin.Array<kotlin.String>)
No source file named kfun.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (kfun:main(kotlin.Array<kotlin.String>)) pending
```

- By location:

```
(gdb) b hello.kt:1
Breakpoint 2 at 0x100001704: file /Users/minamoto/ws/.git-trees/hello.kt, line 1.
```

- By address:

```
(gdb) b *0x100001704
Note: breakpoint 2 also set at pc 0x100001704.
Breakpoint 3 at 0x100001704: file /Users/minamoto/ws/.git-trees/hello.kt, line 2.
```

Use stepping

Stepping through functions works mostly the same way as for C/C++ programs.

Inspect variables

Variable inspection for `var` variables works out of the box for both primitive and non-primitive types:

```
$ cat -n main.kt
     1  fun main(args: Array<String>) {
     2      var x = 1
     3      var y = 2
     4      var p = Point(x, y)
     5      println("p = $p")
     6  }
     7
     8  data class Point(val x: Int, val y: Int)

$ lldb ./program.kexe -o 'b main.kt:5' -o
(lldb) target create "./program.kexe"
Current executable set to './program.kexe' (x86_64).
(lldb) b main.kt:5
Breakpoint 1: where = program.kexe`kfun:main(kotlin.Array<kotlin.String>) + 289 at main.kt:5
(lldb) r
Process 4985 stopped
* thread #1, name = 'program.kexe', stop reason = breakpoint 1.1
    frame #0: program.kexe`kfun:main(kotlin.Array<kotlin.String>) at main.kt:5
   2        var x = 1
   3        var y = 2
   4        var p = Point(x, y)
-> 5        println("p = $p")
   6    }
   7
   8    data class Point(val x: Int, val y: Int)

Process 4985 launched: './program.kexe' (x86_64)
(lldb) fr var
(int) x = 1
(int) y = 2
(ObjHeader *) p = Point(x=1, y=2)

(lldb) v p->x
(int32_t) p->x = 1
```

Debug iOS applications

Debugging iOS applications sometimes involves analyzing crash reports in detail. Crash reports typically require symbolication, the process of translating memory addresses into readable source code locations.

To symbolicate addresses in Kotlin code (for example, for stack trace elements corresponding to Kotlin code), you need a special debug symbol (`.dSYM`) file. This file maps memory addresses in crash reports with actual locations in the source code, such as functions or line numbers.

The Kotlin/Native compiler generates `.dSYM` files for release (optimized) binaries on Apple platforms by default. When building in Xcode, the IDE looks for `.dSYM` files in standard locations and uses them automatically for symbolication. Xcode automatically detects `.dSYM` files in projects created from IntelliJ IDEA templates.

On other platforms, you can add debug information into the produced binaries (which increases their size) using the `-Xadd-light-debug` compiler option:

Kotlin

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.all {
            freeCompilerArgs += "-Xadd-light-debug=enable"
        }
    }
}
```

Groovy

```
kotlin {
    targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget) {
        binaries.all {
            freeCompilerArgs += "-Xadd-light-debug=enable"
        }
    }
}
```

For more information about crash reports, see the Apple documentation.

Known issues

- Performance of Python bindings.
- Expression evaluation in debugger tools is not supported, and currently there are no plans for implementing it.

What's next?

Learn how to improve crash reports for unhandled Kotlin exceptions in iOS code

Kotlin/Native supported targets and hosts

This document describes which targets and hosts are supported by the Kotlin/Native compiler.

We can adjust the list of supported targets and hosts, the number of tiers, and their features as we go.

Target tiers

The Kotlin/Native compiler supports a number of different targets, though the level of support for them varies. To clarify these levels, we've broken up the targets into several tiers depending on how well the compiler supports them.

Tier tables have the following columns:

- **Gradle target name** is a target name that is used in the Kotlin Multiplatform Gradle plugin to enable the target.
- **Target triple** is a target name according to the `<architecture>-<vendor>-<system>-<abi>` structure that compilers commonly use.

- **Running tests** indicates whether the target allows users to run tests out of the box in Gradle and IDE (not to be confused with CI tests that are run for targets themselves).

  This is only available on a native host for the specific target. For example, you can run `macosArm64` and `iosArm64` tests only on the macOS ARM64 host.

Tier 1

- The target is regularly tested on CI to be able to compile and run.
- We provide a source and binary compatibility between compiler releases.

Gradle target name	Target triple	Running tests	Description
Apple macOS hosts only:			
`macosArm64`	`aarch64-apple-macos`	✅	Apple macOS 12.0 and later on Apple Silicon platforms
`iosSimulatorArm64`	`aarch64-apple-ios-simulator`	✅	Apple iOS simulator 15.0 and later on Apple Silicon platforms
`iosArm64`	`aarch64-apple-ios`		Apple iOS and iPadOS 15.0 and later on ARM64 platforms

Tier 2

- The target is regularly tested on CI to be able to compile but may not be automatically tested to be able to run.
- We're doing our best to provide source and binary compatibility between compiler releases.

Gradle target name	Target triple	Running tests	Description
`linuxX64`	`x86_64-unknown-linux-gnu`	✅	Linux on x86_64 platforms
`linuxArm64`	`aarch64-unknown-linux-gnu`		Linux on ARM64 platforms
Apple macOS hosts only:			
`watchosSimulatorArm64`	`aarch64-apple-watchos-simulator`	✅	Apple watchOS simulator 8.0 and later on Apple Silicon platforms

Gradle target name	Target triple	Running tests	Description
`watchosArm64`	`arm64_32-apple-watchos`		Apple watchOS 8.0 and later on ARM64 platforms with ILP32
`tvosSimulatorArm64`	`aarch64-apple-tvos-simulator`	✅	Apple tvOS simulator 15.0 and later on Apple Silicon platforms
`tvosArm64`	`aarch64-apple-tvos`		Apple tvOS 15.0 and later on ARM64 platforms

Tier 3

- The target is not guaranteed to be tested on CI.
- We can't promise a source and binary compatibility between different compiler releases, though such changes for these targets are quite rare.

> The Tier 3 targets are not in active development and may come with breaking issues. Use them with caution.

Gradle target name	Target triple	Running tests	Description
`androidNativeArm32`	`arm-unknown-linux-androideabi`		Android NDK on ARM32 platforms
`androidNativeArm64`	`aarch64-unknown-linux-android`		Android NDK on ARM64 platforms
`androidNativeX86`	`i686-unknown-linux-android`		Android NDK on x86 platforms
`androidNativeX64`	`x86_64-unknown-linux-android`		Android NDK on x86_64 platforms
`mingwX64`	`x86_64-pc-windows-gnu`	✅	64-bit Windows 10 and later using MinGW compatibility layer
Apple macOS hosts only:			
`watchosDeviceArm64`	`aarch64-apple-watchos`		Apple watchOS 8.0 and later on ARM64 platforms

Gradle target name	Target triple	Running tests	Description
`iosX64`	`x86_64-apple-ios-simulator`		Apple iOS simulator 15.0 and later on x86-64 platforms

Deprecated targets

The following targets are deprecated and scheduled for removal:

Target	Deprecation start	Description
`watchosArm32`	Kotlin 2.4.20	Apple watchOS devices on ARM32 platforms
`macosX64`	Kotlin 2.3.20	Apple macOS on x86_64 platforms
`watchosX64`	Kotlin 2.3.20	Apple watchOS 64-bit simulator on x86_64 platforms
`tvosX64`	Kotlin 2.3.20	Apple tvOS simulator on x86_64 platforms
`linuxArm32Hfp`	Kotlin 1.8.20	Linux on ARM32 platforms

Supporting lower Apple target versions

Currently, the default minimum supported versions of Apple targets are:

- For iOS and tvOS, 15.0.
- For macOS, 12.0.
- For watchOS, 8.0.

If your project needs to support a version lower than the default one, use the `freeCompilerArgs` option in your build file:

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget>().configureEach {
        binaries.configureEach {
            freeCompilerArgs += "-Xoverride-konan-properties=minVersion.ios=14.0"
            freeCompilerArgs += "-Xoverride-konan-properties=minVersion.macos=11.0"
            freeCompilerArgs += "-Xoverride-konan-properties=minVersion.tvos=14.0"
            freeCompilerArgs += "-Xoverride-konan-properties=minVersion.watchos=7.0"
        }
    }
}
```

For library authors

We don't recommend library authors to test more targets or provide stricter guarantees than the Kotlin/Native compiler does. You can use the following approach when considering support for native targets:

-

Support all the targets from tiers 1, 2, and 3.

- Regularly test targets from tiers 1 and 2 that support running tests out of the box.

The Kotlin team uses this approach in the official Kotlin libraries, for example, kotlinx.coroutines and kotlinx.serialization.

Hosts

The Kotlin/Native compiler supports the following hosts:

Host OS	Building final binaries	Producing `.klib` artifacts
macOS on Apple silicon (ARM64)	Any supported target	Any supported target
macOS on Intel chips (x86_64)	Any supported target	Any supported target
Linux with x86_64 architecture	Any supported target, except for Apple targets	Any supported target, Apple targets only without cinterop dependencies
Windows (MinGW toolchain) with x86_64 architecture	Any supported target, except for Apple targets	Any supported target, Apple targets only without cinterop dependencies

Building final binaries

To produce final binaries, you can compile for supported targets only on *supported hosts*. For example, you cannot do it on FreeBSD or on a Linux machine running on the ARM64 architecture.

Building final binaries for Apple targets on Linux and Windows is also not possible.

Producing `.klib` artifacts

Generally, Kotlin/Native allows any *supported host* to produce `.klib` artifacts for supported targets.

However, artifact production for Apple targets still has some limitations on Linux and Windows. If your project uses cinterop dependencies (including CocoaPods), you must use a macOS host.

For example, you can produce a `.klib` for `macosArm64` target on a Windows machine running on the x86_64 architecture only if there are no cinterop dependencies.

What's next?

- Build final native binaries
- Compilation for Apple targets

Tips for improving compilation time

The Kotlin/Native compiler is constantly receiving updates that improve its performance. With the latest Kotlin/Native compiler and a properly configured build environment, you can significantly improve the compilation times of your projects with Kotlin/Native targets.

Read on for our tips on how to speed up the Kotlin/Native compilation process.

General recommendations

Use the latest version of Kotlin

This way, you always get the latest performance improvements. The most recent Kotlin version is 2.4.20.

Avoid creating huge classes

Try to avoid huge classes that take a long time to compile and load during execution.

Preserve downloaded and cached components between builds

When compiling projects, Kotlin/Native downloads the required components and caches some results of its work to the `$USER_HOME/.konan` directory. The compiler uses this directory for subsequent compilations, making them take less time to complete.

When building in containers (such as Docker) or with continuous integration systems, the compiler may have to create the `~/.konan` directory from scratch for each build. To avoid this step, configure your environment to preserve `~/.konan` between builds. For example, redefine its location using the `konan.data.dir` Gradle property.

Alternatively, you can use the `-Xkonan-data-dir` compiler option to configure your custom path to the directory via the `cinterop` and `konanc` tools.

Gradle configuration

The first compilation with Gradle usually takes more time than subsequent ones due to the need to download the dependencies, build caches, and perform additional steps. You should build your project at least twice to get an accurate reading of the actual compilation times.

Below are some recommendations for configuring Gradle for better compilation performance.

Increase Gradle heap size

To increase the Gradle heap size, add `org.gradle.jvmargs=-Xmx3g` to your `gradle.properties` file.

If you use parallel builds, you might need to choose the right number of workers with the `org.gradle.workers.max` property or the `--max-workers` command-line option. The default value is the number of CPU processors.

Build only necessary binaries

Don't run Gradle tasks that build the whole project, such as `build` or `assemble`, unless you really need to. These tasks build the same code more than once, increasing the compilation times. In typical cases, such as running tests from IntelliJ IDEA or starting the app from Xcode, the Kotlin tooling avoids executing unnecessary tasks.

If you have a non-typical case or build configuration, you might need to choose the task yourself:

- `linkDebug*`. To run your code during development, you usually need only one binary, so running the corresponding `linkDebug*` task should be enough.
- `embedAndSignAppleFrameworkForXcode`. Since iOS simulators and devices have different processor architectures, it's a common approach to distribute a Kotlin/Native binary as a universal (fat) framework.

  However, during local development, it's faster to build the `.framework` file only for the platform you're using. To build a platform-specific framework, use the embedAndSignAppleFrameworkForXcode task.

Build only for necessary targets

Similarly to the recommendation above, don't build a binary for all native platforms at once. For example, compiling an XCFramework (using an `*XCFramework` task) builds the same code for all targets, which takes proportionally more time than building for a single target.

If you do need XCFrameworks for your setup, you can reduce the number of targets. For example, you don't need `iosX64` if you don't run this project on iOS simulators on Intel-based Macs.

> Binaries for different targets are built with `linkDebug*$Target` and `linkRelease*$Target` Gradle tasks. You can look for the executed tasks in the build log or in the Gradle build scan by running a Gradle build with the `--scan` option.

Don't build unnecessary release binaries

Kotlin/Native supports two build modes, debug and release. Release is highly optimized, and this takes a lot of time: compilation of release binaries takes an order of magnitude more time than debug binaries.

Apart from an actual release, all these optimizations might be unnecessary in a typical development cycle. If you use a task with `Release` in its name during the development process, consider replacing it with `Debug`. Similarly, instead of running `assembleXCFramework`, you can run `assembleSharedDebugXCFramework`, for example.

> Release binaries are built with `linkRelease*` Gradle tasks. You can check for them in the build log or in the Gradle build scan by running a Gradle build with the `--scan` option.

Experimental

Enable caches for release binaries

By default, Kotlin/Native compiles release binaries in the link-time optimization (LTO) mode: all modules are compiled and optimized together. This makes release binaries faster at runtime but significantly increases compilation time.

If you prefer to prioritize faster compilation over some compiler optimizations, you can enable caching in release mode. To enable caching, add both following options to your `gradle.properties` file:

```
# Enables the compiler to use caches in release mode
kotlin.native.binary.enableReleaseBinaryCache=true
# Makes the Kotlin Gradle plugin invoke the compiler with `-Xauto-cache-from` and related options
kotlin.internal.native.enableReleaseBinaryCache=true
```

The feature is in active development, so runtime performance still has room for improvement. Performance improvements are planned in upcoming Kotlin releases.

Experimental

Reduce the size of release binaries

To reduce the size of release binaries and improve build time, try enabling the binary option `smallBinary`.

It effectively sets `-Oz` as the default optimization argument for the compiler during the LLVM compilation phase. The option is still Experimental and might affect runtime performance in some cases.

Don't use transitive export

Using `transitiveExport = true` disables dead code elimination in many cases, so the compiler has to process a lot of unused code. It increases the compilation time. Instead, use the `export` method explicitly for exporting the required projects and dependencies.

Don't export modules too much

Try to avoid unnecessary module export. Each exported module negatively affects compilation time and binary size.

Use Gradle build caching

Enable the Gradle build cache feature:

- **Local build cache**. For local caching, add `org.gradle.caching=true` to your `gradle.properties` file or run the build with the `--build-cache` option in the command line.
- **Remote build cache**. Learn how to configure the remote build cache for continuous integration environments.

Use Gradle configuration cache

The Gradle configuration cache improves build performance by caching the results of the configuration phase. It also enables parallel execution of independent tasks within a single project and implicitly enables the `org.gradle.parallel` property, allowing tasks across different projects to execute in parallel.

To use the Gradle configuration cache, add the `org.gradle.configuration-cache=true` property to your `gradle.properties` file.

The configuration cache also enables running `link*` tasks in parallel which could heavily load the machine, specifically with a lot of CPU cores. This issue will be fixed in KT-70915.

Enable previously disabled features

You might have disabled some Kotlin/Native features in the past to work around build issues. For example:

- `kotlin.native.disableCompilerDaemon=true` disables the Gradle daemon.
- `disableNativeCache` disables the compilation cache.

The issues that originally required these workarounds might have already been fixed. If your `gradle.properties` file or Gradle build files contain these lines, remove them and check whether the build completes successfully.

Beta

Try incremental compilation of klib artifacts

With incremental compilation, if only a part of the `klib` artifact produced by the project module changes, just a part of `klib` is further recompiled into a binary.

This feature is in Beta. To enable it, add the following option to your `gradle.properties` file:

```
kotlin.incremental.native=true
```

If you face any problems, create an issue in YouTrack.

Windows configuration

Windows Security may slow down the Kotlin/Native compiler. You can avoid this by adding the `.konan` directory, which is located in `%USERPROFILE%` by default, to Windows Security exclusions. Learn how to add exclusions to Windows Security.

LLVM configuration

If the above tips didn't help with improving compilation time, consider customizing the LLVM backend.

Tips for customizing LLVM backend

The Kotlin/Native compiler uses LLVM to optimize and generate binary executables for different target platforms. A noticeable part of the compilation time is also spent in LLVM, and for large apps, this can end up taking an unacceptably long time.

You can customize how Kotlin/Native uses LLVM and adjust the list of optimization passes.

Examine the build log

Let's take a look at the build log to understand how much compilation time is spent on LLVM optimization passes:

1. Run the `linkRelease*` Gradle task with `-Pkotlin.internal.compiler.arguments.log.level=warning` option to make Gradle output LLVM profiling details, for example:

   ```
   ./gradlew linkReleaseExecutableMacosArm64 -Pkotlin.internal.compiler.arguments.log.level=warning
   ```

   While executing, the task prints the necessary compiler arguments, for example:

```
> Task :linkReleaseExecutableMacosArm64
Run in-process tool "konanc"
Entry point method = org.jetbrains.kotlin.cli.utilities.MainKt.daemonMain
Classpath = [
        /Users/user/.konan/kotlin-native-prebuilt-macos-aarch64-2.2.0/konan/lib/kotlin-native-
compiler-embeddable.jar
        /Users/user/.konan/kotlin-native-prebuilt-macos-aarch64-2.2.0/konan/lib/trove4j.jar
]
Arguments = [
        -Xinclude=...
        -library
        /Users/user/.konan/kotlin-native-prebuilt-macos-aarch64-2.2.0/klib/common/stdlib
        -no-endorsed-libs
        -nostdlib
        ...
]
```

2. Run the command line compiler with the provided arguments plus the `-Xprofile-phases` argument, for example:

```
/Users/user/.konan/kotlin-native-prebuilt-macos-aarch64-2.2.0/bin/kotlinc-native \
-Xinclude=... \
-library /Users/user/.konan/kotlin-native-prebuilt-macos-aarch64-2.2.0/klib/common/stdlib \
... \
-Xprofile-phases
```

3. Examine the generated output in the build log. The log can contain tens of thousands of lines; sections with LLVM profiling are at the end.

Here is an excerpt from such a run of a simple Kotlin/Native program:

```
Frontend: 275 msec
PsiToIr: 1186 msec
...
... 30k lines
...
LinkBitcodeDependencies: 476 msec
StackProtectorPhase: 0 msec
MandatoryBitcodeLLVMPostprocessingPhase: 2 msec
===-------------------------------------------------------------------------===
                          Pass execution timing report
===-------------------------------------------------------------------------===
  Total Execution Time: 6.7726 seconds (6.7192 wall clock)

   ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Name ---
   0.9778 ( 22.4%)   0.5043 ( 21.0%)   1.4821 ( 21.9%)   1.4628 ( 21.8%)  InstCombinePass
   0.3827 (  8.8%)   0.2497 ( 10.4%)   0.6323 (  9.3%)   0.6283 (  9.4%)  InlinerPass
   0.2815 (  6.4%)   0.1792 (  7.5%)   0.4608 (  6.8%)   0.4555 (  6.8%)  SimplifyCFGPass
...
   0.6444 (100.0%)   0.5474 (100.0%)   1.1917 (100.0%)   1.1870 (100.0%)  Total

ModuleBitcodeOptimization: 8118 msec
...
LTOBitcodeOptimization: 1399 msec
...
```

The Kotlin/Native compiler runs two separate sequences of LLVM optimizations: the module passes and the link-time passes. For a typical compilation, the two pipelines are run back to back, and the only real distinction is in which LLVM optimization passes they run.

In the log above, the two LLVM optimizations are `ModuleBitcodeOptimization` and `LTOBitcodeOptimization`. The formatted tables are the optimizations' output with timing for each pass.

Customize LLVM optimization passes

If one of the passes above seems unreasonably long, you can skip it. However, this might hurt runtime performance, so you should check for changes in the benchmarks' performance afterward.

Currently, there is no direct way to disable a given pass. However, you can provide a new list of passes to run by using the following compiler options:

Option	Default value for release binary
`-Xllvm-module-passes`	`"default<O3>"`
`-Xllvm-lto-passes`	`"internalize,globaldce,lto<O3>"`

The default values are unfolded to a long list of actual passes, from which you need to exclude the undesired ones.

To get the list of actual passes, run the `opt` tool, which is automatically downloaded with the LLVM distribution to the `~/.konan/dependencies/llvm-{VERSION}-{ARCH}-{OS}-dev-{BUILD}/bin` directory.

For example, to get the list of the link-time passes, run:

```
opt -print-pipeline-passes -passes="internalize,globaldce,lto<O3>" < /dev/null
```

This outputs a warning and a long list of passes, which depends on the LLVM version.

There are two differences between the list of passes from the `opt` tool and the passes that Kotlin/Native compiler actually runs:

- Since `opt` is a debug tool, it includes one or more `verify` passes, which are not normally run.
- Kotlin/Native disables the `devirt` passes since the Kotlin compiler already does them itself.

After disabling any passes, always rerun performance tests to check if the runtime performance degradation is acceptable.

License files for the Kotlin/Native binaries

Like many other open-source projects, Kotlin relies on third-party code, meaning that the Kotlin project includes some code not developed by JetBrains or the Kotlin programming language contributors. Sometimes it is derived work, such as code rewritten from C++ to Kotlin.

You can find licenses for the third-party work used in Kotlin in our GitHub repository:

- Kotlin compiler
- Kotlin/Native

In particular, the Kotlin/Native compiler produces binaries that can include third-party code, data, or derived work. This means that the Kotlin/Native-compiled binaries are subject to the terms and conditions of the third-party licenses.

In practice, if you distribute a Kotlin/Native-compiled final binary, you should always include the necessary license files in your binary distribution. The files should be accessible to users of your distribution in a readable form.

Always include the following license files for the corresponding projects:

Project	Files to be included
Kotlin	• Apache License 2.0
Apache Harmony	• Apache Harmony copyright notice
GWT	
Guava	
libbacktrace	3-clause BSD license with copyright notice
mimalloc	MIT license Include in case you use the mimalloc memory allocator instead of the default one (the `-Xallocator=mimalloc` compiler option is set).
Unicode character database	Unicode license
Multi-producer/multi-consumer bounded queue	Copyright notice

The `mingwX64` target requires additional license files:

Project	Files to be included
MinGW-w64 headers and runtime libraries	• MinGW-w64 runtime license • Winpthreads license

None of these libraries require the distributed Kotlin/Native binaries to be open-sourced.

Kotlin/Native FAQ

How do I run my program?

Define a top-level function `fun main(args: Array<String>)` or just `fun main()` if you are not interested in passed arguments, please ensure it's not in a package. Also, compiler switch `-entry` could be used to make any function taking `Array<String>` or no arguments and return `Unit` as an entry point.

What is Kotlin/Native memory management model?

Kotlin/Native uses an automated memory management scheme that is similar to what Java or Swift provide.

Learn about the Kotlin/Native memory manager

How do I create a shared library?

Use the `-produce dynamic` compiler option or `binaries.sharedLib()` in your Gradle build file:

```
kotlin {
    iosArm64("mylib") {
        binaries.sharedLib()
    }
}
```

It produces a platform-specific shared object (`.so` on Linux, `.dylib` on macOS, and `.dll` on Windows targets) and a C language header, allowing the use of all public APIs available in your Kotlin/Native program from C/C++ code.

Complete the Kotlin/Native as a dynamic library tutorial

How do I create a static library or an object file?

Use the `-produce static` compiler option or `binaries.staticLib()` in your Gradle build file:

```
kotlin {
    iosArm64("mylib") {
        binaries.staticLib()
    }
}
```

It produces a platform-specific static object (`.a` library format) and a C language header, allowing you to use all the public APIs available in your Kotlin/Native program from C/C++ code.

How do I run Kotlin/Native behind a corporate proxy?

As Kotlin/Native needs to download a platform-specific toolchain, you need to specify `-Dhttp.proxyHost=xxx -Dhttp.proxyPort=xxx` as the compiler's or `gradlew` arguments, or set it via the `JAVA_OPTS` environment variable.

How do I specify a custom Objective-C prefix/name for my Kotlin framework?

Use the `-module-name` compiler option or matching Gradle DSL statement.

Kotlin

```
kotlin {
    iosArm64("myapp") {
        binaries.framework {
            freeCompilerArgs += listOf("-module-name", "TheName")
        }
    }
}
```

Groovy

```
kotlin {
    iosArm64("myapp") {
        binaries.framework {
            freeCompilerArgs += ["-module-name", "TheName"]
        }
    }
}
```

How do I rename the iOS framework?

The default name for an iOS framework is `<project name>.framework`. To set a custom name, use the `baseName` option. This will also set the module name.

```
kotlin {
    iosArm64("myapp") {
       binaries {
          framework {
              baseName = "TheName"
          }
       }
    }
}
```

How do I enable bitcode for my Kotlin framework?

Bitcode embedding was deprecated in Xcode 14 and removed in Xcode 15 for all Apple targets. The Kotlin/Native compiler does not support bitcode embedding since Kotlin 2.0.20.

If you're using earlier versions of Xcode but want to upgrade to Kotlin 2.0.20 or later versions, disable bitcode embedding in your Xcode projects.

How do I get better crash reports for iOS?

When an unhandled Kotlin exception reaches iOS code, crash reporting tools may show a vague report stating an internal Kotlin/Native issue instead of the actual line of code where the exception was thrown. This happens because the original Kotlin stack trace is lost by the time the crash is recorded.

You can work around this problem with third-party solutions like NSExceptionKt, which converts uncaught Kotlin exceptions into `NSException` instances so that your crash reporting tool captures the correct stack trace. It provides dedicated integrations for popular crash reporting tools:

- Bugsnag
- Firebase Crashlytics

How do I reference objects safely from different coroutines?

To safely access or update an object across multiple coroutines in Kotlin/Native, consider using concurrency-safe constructs, such as `@Volatile` and `AtomicReference`.

Use `@Volatile` to annotate a `var` property. This makes all reads and writes to the property's backing field atomic. In addition, writes become immediately visible to other threads. When another thread accesses this property, it observes not only the updated value but also the changes that happened before the update.

Alternatively, use AtomicReference, which supports atomic reads and updates. On Kotlin/Native, it wraps a volatile variable and performs atomic operations. Kotlin also provides a set of types for atomic operations tailored to specific data types. You can use `AtomicInt`, `AtomicLong`, `AtomicBoolean`, `AtomicArray`, as well as `AtomicIntArray` and `AtomicLongArray`.

For more information about access to shared mutable state, see the Coroutines documentation.

How can I compile my project with unreleased versions of Kotlin/Native?

First, please consider trying preview versions.

In case you need an even more recent development version, you can build Kotlin/Native from source code: clone the Kotlin repository and follow these steps.

Kotlin for Android

Android mobile development has been Kotlin-first since Google I/O in 2019.

Over 50% of professional Android developers use Kotlin as their primary language, while only 30% use Java as their main language. 70% of developers whose primary language is Kotlin say that Kotlin makes them more productive.

Using Kotlin for Android development, you can benefit from:

- **Less code combined with greater readability**. Spend less time writing your code and working to understand the code of others.
- **Fewer common errors**. Apps built with Kotlin are 20% less likely to crash based on Google's internal data.
- **Kotlin support in Jetpack libraries**. Jetpack Compose is Android's recommended modern toolkit for building native UI in Kotlin. KTX extensions add Kotlin language features, like coroutines, extension functions, lambdas, and named parameters to existing Android libraries.

- **Support for multiplatform development**. Kotlin Multiplatform allows development for not only Android but also iOS, backend, and web applications. Some Jetpack libraries are already multiplatform. Compose Multiplatform, JetBrains' declarative UI framework based on Kotlin and Jetpack Compose, makes it possible to share UIs across platforms – iOS, Android, desktop, and web.
- **Mature language and environment**. Since its creation in 2011, Kotlin has developed continuously, not only as a language but as a whole ecosystem with robust tooling. Now it's seamlessly integrated into Android Studio and is actively used by many companies for developing Android applications.
- **Interoperability with Java**. You can use Kotlin along with the Java programming language in your applications without needing to migrate all your code to Kotlin.
- **Easy learning**. Kotlin is very easy to learn, especially for Java developers.
- **Big community**. Kotlin has great support and many contributions from the community, which is growing all over the world. Over 95% of the top thousand Android apps use Kotlin.

Many startups and Fortune 500 companies have already developed Android applications using Kotlin, see the list on the Google website for Android developers.

To start using Kotlin for:

- Android development, read Google's documentation for developing Android apps with Kotlin.
- Developing cross-platform mobile applications, see Create an app with shared logic and native UI.

Introduction to library authors' guidelines

This guide contains a summary of best practices and ideas to consider when designing libraries.

To be effective, a library must achieve certain fundamental objectives. Specifically, it should:

- Define its problem domain and implement a set of related functional requirements that solve the problems it defined. For example, an HTTP client may aim to support all HTTP request types and understand the various headers, content types, and status codes.
- Meet the non-functional criteria appropriate to the problem domain. These typically involve performance, reliability, security, and usability. The relative importance of these criteria varies widely. For example, a library designed for batch processing may not require the same level of performance as one intended for day trading.

> The process of identifying and defining functional and non-functional requirements is a complex topic that has been extensively studied in software engineering. This guide does not cover these topics in depth, as they are beyond its scope.

The main focus of this guide is to explore the characteristics that a library must have to stay relevant and popular with its users. These characteristics include:

- **Minimize mental complexity:** All developers must consider the readability and maintainability of their code. It's crucial to reduce the mental effort required for others to read, understand, and use your APIs. Achieving this involves creating libraries that are clear, consistent, predictable, and easy to debug.
- **Backward compatibility:** When releasing a new version of an API, ensure the existing API remains operational. Clearly communicate and document any breaking changes well in advance. Provide a straightforward, clear, and gradual pathway for users to adopt the new API or design changes.
- **Informative documentation:** The documentation that accompanies the library needs to do more than repeat function and type declarations. It should be comprehensive and specifically tailored to the library's audience. It

should accurately reflect the needs and scenarios of various user roles, ensuring it provides essential information without being overly simplistic or complex. Always include clear examples, balancing explanatory text with practical code samples.

Additionally, **building your Kotlin library with multiplatform support** can broaden its applicability across projects targeting various environments. Designing APIs to work reliably in both shared and platform-specific code can improve the library's versatility and usability across all supported targets.

The following sections dig deeper into these characteristics, with practical advice on how you can provide the best possible experience to users of your libraries.

What's next

- Explore strategies for minimizing mental complexity in Minimizing mental complexity.
- Learn about maintaining backward compatibility in Backward compatibility.
- For an extensive overview on effective documentation practices, see Informative documentation.
- Discover best practices for building multiplatform libraries in Building Kotlin library for multiplatform.

Minimizing mental complexity overview

Users need to quickly and accurately build a mental model of your library's functions and abstractions before using it. The best way to achieve this is by minimizing the amount of complexity they encounter.

Strategies for minimizing mental complexity include:

- **Simplicity:** Strive for an API that provides the most functionality with the fewest components, reusing existing Kotlin types and structures to avoid redundancy. Where possible, create a small set of core abstractions and build additional functionality on top of them.
- **Readability:** Write the API in a declarative style to make the code's intent clear. Choose names for abstractions directly from the problem domain, unless it's absolutely necessary to invent new ones. Use basic data types for their intended purposes. Clearly distinguish between core and optional functionality.
- **Consistency:** Maintain a single, clear approach for every design aspect of your API. Use uniform naming conventions, error handling strategies, and patterns, whether they are object-oriented or functional.
- **Predictability:** Design your library to adhere to the 'principle of least surprise'. Ensure the default settings match the most common use cases, allowing users to accomplish their tasks with the simplest and shortest code. Allow extensions to your library only in clearly specified ways to maintain consistency and predictability.
- **Debuggability:** Ensure your library aids users in troubleshooting by facilitating the extraction of information and navigation through nested function calls. When exceptions are thrown, both the type and content of the exception should match the underlying issue, providing all necessary details to effectively diagnose and resolve problems. It should be possible to capture and output the state of domain objects, and to view any intermediate representations.
- **Testability:** Ensure that your library, as well as the code that uses it, can be easily tested.

The following sections give more detailed information on implementing these strategies in Kotlin.

Next step

To begin exploring these strategies in depth, you can start with learning about simplicity in the next section.

Proceed to the next part

Simplicity

The fewer concepts your users need to understand and the more explicitly these are communicated, the simpler their mental model is likely to be. This can be achieved by limiting the number of operations and abstractions in the API.

Ensure that the visibility of declarations in your library is set appropriately to keep internal implementation details out of the public API. Only APIs that are explicitly designed and documented for public use should be accessible to users.

In the next part of the guide, we'll discuss some guidelines for promoting simplicity.

Use explicit API mode

We recommend using the explicit API mode feature of the Kotlin compiler, which forces you to explicitly state your intentions when you're designing the API for your library.

With explicit API mode, you must:

- Add visibility modifiers to your declarations to make them public, instead of relying on the default public visibility. This ensures that you've considered what you're exposing as part of the public API.
- Define the types for all your public functions and properties to prevent unintended changes to your API from inferred types.

Reuse existing concepts

One way to limit the size of your API is to reuse existing types. For example, instead of creating a new type for durations, you can use `kotlin.time.Duration`. This approach not only streamlines development but also improves interoperability with other libraries.

Be careful when relying on types from third-party libraries or platform-specific types, as they can tie your library to these elements. In such cases, the costs may outweigh the benefits.

Reusing common types such as `String`, `Long`, `Pair`, and `Triple` can be effective, but this should not stop you from developing abstract data types if they better encapsulate domain-specific logic.

Define and build on top of core API

Another route to simplicity is to define a small conceptual model based around a limited set of core operations. Once the behavior of these operations is clearly documented, you can expand the API by developing new operations that build directly on or combine these core functions.

For example:

- In the Kotlin Flows API, common operations like `filter` and `map` are built on top of the `transform` operation.

- In the Kotlin Time API the `measureTime` function utilizes `TimeSource.Monotonic`.

While it's often beneficial to base additional operations on these core components, it's not always necessary. You may find opportunities to introduce optimized or platform-specific variations that expand the functionality or adapt more broadly to different inputs.

As long as users are able to solve non-trivial problems with the core operations and can refactor their solutions with additional operations without altering any behavior, the simplicity of the conceptual model is preserved.

Next step

In the next part of the guide, you'll learn about readability.

Proceed to the next part

Readability

Creating a readable API involves more than just writing clean code. It requires thoughtful design that simplifies integration and usage. This section explores how you can enhance API readability by structuring your library with composability in mind, utilizing domain-specific languages (DSLs) for concise and expressive setup, and using extension functions and properties for clear and maintainable code.

Prefer Explicit Composability

Libraries often provide advanced operators that allow for customization. For example, an operation might permit users to supply their own data structures, networking channels, timers, or lifecycle observers. However, introducing these customization options through additional function parameters can significantly increase the complexity of the API.

Instead of adding more parameters for customization, it's more effective to design an API where different behaviors can be composed together. For example, in the coroutine Flows API both buffering and conflation are implemented as separate functions. These can be chained together with more basic operations like `filter` and `map`, instead of each basic operation accepting parameters to control buffering and conflation.

Another example involves the Modifiers API in Jetpack Compose. This allows Composable components to accept a single `Modifier` parameter that handles common customization options, such as padding, sizing, and background color. This approach avoids the need for each Composable to accept separate parameters for these customizations, streamlining the API and reducing complexity.

```
Box(
    modifier = Modifier
        .padding(10.dp)
        .onClick { println("Box clicked!") }
        .fillMaxWidth()
        .fillMaxHeight()
        .verticalScroll(rememberScrollState())
        .horizontalScroll(rememberScrollState())
) {
    // Box content goes here
}
```

Use DSLs

A Kotlin library can significantly improve readability by providing a builder DSL. Using a DSL allows you to concisely repeat domain-specific data declarations. For example, consider the following sample from a Ktor-based server application:

```
fun Application.module() {
    install(ContentNegotiation) {
        json(Json {
            prettyPrint = true
            isLenient = true
        })
    }
    routing {
        post("/article") {
            call.respond<String>(HttpStatusCode.Created, ...)
        }
        get("/article/list") {
            call.respond<List<CreateArticle>>(...)
        }
        get("/article/{id}") {
            call.respond<Article>(...)
        }
    }
}
```

This sets up an application, installing the `ContentNegotiation` plugin configured to use Json serialization, and sets up routing so that the application responds to requests on various `/article` endpoints.

For a detailed description of creating DSLs, see Type-safe builders. The following points are worth noting in the context of creating libraries:

- The functions used in the DSL are builder functions, which take a lambda with receiver as the final parameter. This design allows these functions to be called without parentheses, making the syntax clearer. The lambda being passed can be used to configure the entity being created. In the example above the lambda passed to the `routing` function is used to configure the details of the routing.
- Factory functions that create instances of classes should have the same name as the return type and start with a capital letter. You can see this in the sample above with the creation of the `Json` instance. These functions may still take lambda parameters for configuration. For more information, see Coding conventions.
- As it's not possible to ensure that required properties have been set within the lambda supplied to a builder function at compile time, we recommend passing required values as function parameters.

Using DSLs to build objects not only improves readability but also improves backward compatibility, and simplifies the documentation process. For example, take the following function:

```
fun Json(prettyPrint: Boolean, isLenient: Boolean): Json
```

This function could replace the `Json{}` DSL builder. However, the DSL approach has noticeable benefits:

- Backward compatibility is easier to maintain with the DSL builder than with this function, as adding new configuration options simply means adding new properties (or in other examples, new functions), which is a backward-compatible change, unlike changing the parameter list of an existing function.
- It also makes creating and maintaining documentation easier. You can document each property separately at its point of declaration, instead of having to document many parameters of a function, all in one place.

Use extension functions and properties

We recommend using extension functions and properties to improve readability.

Classes and interfaces should define the core concept of a type. Additional functionality and information should be written as extension functions and properties. This makes it clear to the reader that the additional functionality can be implemented on top of the core concept, and additional information can be calculated from the data in the type.

For example, the `CharSequence` type (which `String` also implements) only contains the most basic information and operators to access its contents:

```
interface CharSequence {
    val length: Int
    operator fun get(index: Int): Char
    fun subSequence(startIndex: Int, endIndex: Int): CharSequence
}
```

Functionality commonly associated with strings is mostly defined as extension functions, which can all be implemented on top of the core concepts and basic APIs of the type:

```
inline fun CharSequence.isEmpty(): Boolean = length == 0
inline fun CharSequence.isNotEmpty(): Boolean = length > 0

inline fun CharSequence.trimStart(predicate: (Char) -> Boolean): CharSequence {
    for (index in this.indices)
        if (!predicate(this[index]))
            return subSequence(index, length)
    return ""
}
```

Consider declaring computed properties and normal methods as extensions. Only regular properties, overrides, and overloaded operators should be declared as members by default.

Avoid using the boolean type as an argument

Consider the following function:

```
fun doWork(optimizeForSpeed: Boolean) { ... }
```

If you were to provide this function in your API it could be invoked as:

```
doWork(true)
doWork(optimizeForSpeed=true)
```

In the first call it is impossible to infer what the boolean argument is for, unless you are reading the code in an IDE with Parameter Name Hints enabled. Using named arguments does clarify the intention, but there is no way to force your users to adopt this style. Consequently, to improve readability, your code should not use boolean types as arguments.

Alternatively, the API could create a separate function specifically for the task controlled by the boolean argument. This function should have a descriptive name that indicates what it does.

For example, the following extensions are available on the `Iterable` interface:

```
fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R>
fun <T, R : Any> Iterable<T>.mapNotNull(
    transform: (T) -> R?
): List<R>
```

Instead of the single method:

```
fun <T, R> Iterable<T>.map(
    includeNullResults: Boolean = true,
    transform: (T) -> R
): List<R>
```

Another good approach could be to use an `enum` class to define different operation modes. This approach is useful if there are several modes of operation, or if you expect these modes to change over time.

Use numeric types appropriately

Kotlin defines a set of numeric types that you may use as part of your API. Here's how to use them appropriately:

- Use the `Int`, `Long` and `Double` types as arithmetic types. They represent values with which calculations are performed.
- Avoid using arithmetic types for non-arithmetic entities. For example, if you represent an ID as a `Long`, your users might be tempted to compare IDs, on the assumption they are assigned in order. This could lead to unreliable or meaningless results, or create dependencies on implementations that could change without warning. A better strategy is to define a specialized class for the ID abstraction. You could use Inline value classes to build such abstractions without affecting performance. See the `Duration` class for an example.
- The `Byte`, `Float` and `Short` types are memory layout types. They are used to restrict the amount of memory available for storing a value, such as in caches or when transmitting data over a network. These types should only be used when the underlying data reliably fits within that type, and calculations are not required.
- The unsigned integer types `UByte`, `UShort`, `UInt` and `ULong` should be used to utilize the full range of positive values available in a given format. They are suitable for scenarios requiring values beyond the range of signed types or for interoperability with native libraries. However, avoid using them in situations where the domain only requires non-negative integers.

Next step

In the next part of the guide, you'll learn about consistency.

Proceed to the next part

Consistency

Consistency is crucial in API design to ensure ease of use. By maintaining consistent parameter order, naming conventions, and error handling mechanisms, your library will be more intuitive and reliable for users. Following these best practices helps avoid confusion and misuse, leading to a better developer experience and more robust applications.

Preserve parameter order, naming and usage

When designing a library, maintain consistency in the ordering of arguments, the naming scheme, and the use of overloading. For example, if one of your existing methods has `offset` and `length` parameters, you should not switch to alternatives like `startIndex` and `endIndex` for a new method unless there is a compelling reason.

Overloaded functions provided by the library should behave identically. Users expect the behavior to remain consistent when they change the type of a value they pass into your library. For example, these calls all create identical instances, as the input is semantically the same:

```
BigDecimal(200)
BigDecimal(200L)
BigDecimal("200")
```

Avoid mixing parameter names like `startIndex` and `stopIndex` with synonyms like `beginIndex` and `endIndex`. Similarly, choose one term for values in collections, such as `element`, `item`, `entry`, or `entity`, and stick with it.

Name related methods consistently and predictably. As an example, the Kotlin standard library contains pairs like `first` and `firstOrNull`, `single` or `singleOrNull`. These pairs clearly indicate that some might return `null` while others might throw an exception. Parameters should be declared from the general to the specific, so essential inputs appear first and optional inputs last. For example, in `CharSequence.findAnyOf` the `strings` collection goes first, followed by the `startIndex`, and finally the `ignoreCase` flag.

Consider a library managing employee records, and provides the following API to search for employees:

```
fun findStaffBySeniority(
    startIndex: Int,
    minYearsServiceExclusive: Int
): List<Employee>

fun findStaffByAge(
    minAgeInclusive: Int,
    startIndex: Int
): List<Employee>
```

This API would be extremely hard to use correctly. There are multiple parameters of the same type presented in an inconsistent order, and used in an inconsistent way. Users of your library will likely make incorrect assumptions about new functions based on their experience with existing ones.

Use Object-Oriented design for data and state

Kotlin supports both the Object-Oriented and Functional programming styles. Use classes to represent data and state in your API. When the data and state is hierarchical, consider using inheritance.

If all the state required can be passed as parameters, prefer using top-level functions. When calls to these functions will be chained, consider writing them as extension functions to improve readability.

Choose the appropriate error handling mechanism

Kotlin provides several mechanisms for error handling. Your API can throw an exception, return a `null` value, use a custom result type, or use the built-in `Result` type. Ensure that your library uses these options consistently and appropriately.

When data cannot be fetched or calculated, use a nullable return type and return `null` to indicate missing data. In other cases, throw an exception or return a `Result` type.

Consider providing overloads of functions, where one throws an exception, while the other wraps it in a result type instead. In these cases, use the `Catching` suffix to indicate that exceptions are caught in the function. For example, the standard library has the `run` and `runCatching` functions using this convention, and the coroutines library has `receive` and `receiveCatching` methods for channels.

Avoid using exceptions for normal control flow. Design your API to allow for condition checks before attempting operations, thus preventing unnecessary error handling. Command / Query Separation is a useful pattern that can be applied here.

Maintain conventions and quality

The final aspect of consistency relates, not to the design of the library itself, but to maintaining a high level of quality.

You should use automated tools (linters) for static analysis to ensure your code follows both general Kotlin conventions and project-specific conventions.

A Kotlin library should also provide a suite of unit and integration tests covering all documented behaviors of all the API entry points. Tests should include a wide range of inputs, especially known boundary and edge cases. Any untested behavior should be assumed to be (at best) unreliable.

Use this suite of tests during development to verify that changes do not break existing behavior. Run these tests on every release as part of a standardized build and release pipeline. Tools like Kover can be integrated into your build process to measure coverage and generate reports.

Next step

In the next part of the guide, you'll learn about predictability.

Proceed to the next part

Predictability

To design a robust and user-friendly Kotlin library, it's essential to anticipate common use cases, allow for extensibility, and enforce proper usage. Following best practices for default settings, error handling, and state management ensures a seamless experience for users while maintaining the integrity and quality of the library.

Do the right thing by default

Your library should anticipate the "happy path" for each use case, and provide default settings accordingly. Users should not need to supply default values for the library to function correctly.

For example, when using the Ktor `HttpClient` the most common use case is sending a GET request to the server. This can be accomplished using the code below, where only essential information needs to be specified:

```
val client = HttpClient(CIO)
val response: HttpResponse = client.get("https://ktor.io/")
```

It's not necessary to provide values for mandatory HTTP headers or custom event handlers for possible status codes in the response.

If there is no obvious "happy path" for a use case or if a parameter should have a default value but there is no non-contentious option, it likely indicates a flaw in the requirement analysis.

Allow opportunities for extension

When the correct choice cannot be anticipated, allow users to specify their preferred approach. Your library should also let the user supply their own approach or use a third-party extension.

For example, with the Ktor `HttpClient`, users are encouraged to install support for content negotiation when configuring the client, and to specify their preferred serialization formats:

```
val client = HttpClient(CIO) {
    install(ContentNegotiation) {
        json(Json {
            prettyPrint = true
            isLenient = true
        })
    }
}
```

Users can choose which plugins to install or create their own using the separate API for defining client plugins.

Additionally, users can define extension functions and properties for types in the library. As a library author, you can make this easier by designing with extensions in mind, and ensuring your library's types have clear core concepts.

Prevent unwanted and invalid extensions

Users should not be able to extend your library in ways that violate its original design or are impossible within the rules of the problem domain.

For example, when marshaling data to and from JSON, only six types are supported in the output format: `object`, `array`, `number`, `string`, `boolean`, and `null`.

If you create an open class or interface called `JsonElement`, users could create invalid derived types, like `JsonDate`. Instead, you can make the `JsonElement` interface sealed and provide an implementation for each type:

```
sealed interface JsonElement

class JsonNumber(val value: Number) : JsonElement
class JsonObject(val values: Map<String, JsonElement>) : JsonElement
class JsonArray(val values: List<JsonElement>) : JsonElement
class JsonBoolean(val value: Boolean) : JsonElement
class JsonString(val value: String) : JsonElement
object JsonNull : JsonElement
```

Sealed types also enable the compiler to ensure your `when` expressions are exhaustive, without requiring an `else` statement, improving readability and consistency.

Avoid exposing mutable state

When managing multiple values, your API should, whenever possible, accept and/or return read-only collections. Mutable collections are not thread-safe and introduce complexity and unpredictability into your library.

For example, if a user modifies a mutable collection returned from an API entry point, it will be unclear whether they are modifying the implementation's structure or a copy. Similarly, if users can modify the values within a collection after passing it to a library, it will be unclear whether this affects the implementation.

Since arrays are mutable collections, avoid using them in your API. If arrays must be used, make defensive copies before sharing data with users. This ensures your data structures remain unmodified.

This policy of making defensive copies is automatically performed by the compiler for `vararg` arguments. When using the spread operator to pass an existing array where a `vararg` argument is expected, a copy of your array is automatically created.

This behavior is demonstrated in the following example:

```
fun main() {
    fun demo(vararg input: String): Array<out String> = input

    val originalArray = arrayOf("one", "two", "three", "four")
    val newArray = demo(*originalArray)

    originalArray[1] = "ten"

    //prints "one, ten, three, four"
    println(originalArray.joinToString())

    //prints "one, two, three, four"
    println(newArray.joinToString())
}
```

Validate inputs and state

Ensure that your library is used correctly by validating inputs and existing state before implementation proceeds. Use the `require` function to verify inputs and the `check` function to validate existing state.

The `require` function throws an `IllegalArgumentException` if its condition is `false`, causing the function to fail immediately with an appropriate error message:

```
fun saveUser(username: String, password: String) {
    require(username.isNotBlank()) { "Username should not be blank" }
    require(username.all { it.isLetterOrDigit() }) {
        "Username can only contain letters and digits, was: $username"
    }
    require(password.isNotBlank()) { "Password should not be blank" }
    require(password.length >= 7) {
        "Password must contain at least 7 characters"
    }

    /* Implementation can proceed */
}
```

Error messages should include relevant inputs to help users determine the cause of the failure, as shown above by the error message for usernames that contain invalid characters, which includes the incorrect username. An exception to this practice is when including a value in the error message could reveal information that might be

used maliciously as part of a security exploit, which is why the error message for the password's length does not include the password input.

Similarly, the `check` function throws an `IllegalStateException` if its condition is `false`. Use this function to verify the state of an instance, as shown in the example below:

```
class ShoppingCart {
    private val contents = mutableListOf<Item>()

    fun addItem(item: Item) {
        contents.add(item)
    }

    fun purchase(): Amount {
        check(contents.isNotEmpty()) {
            "Cannot purchase an empty cart"
        }
        // Calculate and return amount
    }
}
```

Next step

In the next part of the guide, you'll learn about debuggability.

Proceed to the next part

Debuggability

Users of your library will build on its functionality, and the features they build will contain errors that need to be identified and resolved. This error resolution process might be conducted within a debugger during development or using logging and observability tools in production. Your library can follow these best practices to make debugging it easier.

Provide a toString method for stateful types

For every type that contains state, provide a meaningful `toString` implementation. This implementation should return an intelligible representation of the instance's current content, even for internal types.

Since `toString` representations of types are often written to logs, consider security when implementing this method and avoid returning sensitive user data.

Ensure the format used to describe the state is as consistent as possible across the different types in your library. This format should be explicitly described and thoroughly documented when it is part of a contract implemented by your API. The output from your `toString` methods may support parsing, for example in automated test suites.

For example, consider the following types from a library supporting service subscriptions:

```
enum class SubscriptionResultReason {
    Success, InsufficientFunds, IncompatibleAccount
}

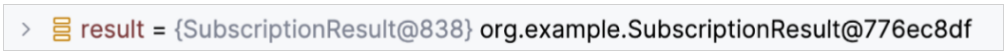
class SubscriptionResult(
    val result: Boolean,
    val reason: SubscriptionResultReason,
    val description: String
)
```

Without a `toString` method, printing a `SubscriptionResult` instance is not very useful:

```
fun main() {
    val result = SubscriptionResult(
       false,
       IncompatibleAccount,
       "Users account does not support this type of subscription"
    )

    //prints 'org.example.SubscriptionResult@13221655'
    println(result)
}
```

Nor is the information readily displayed in the debugger:



Adding a simple `toString` implementation improves the output significantly in both cases:

```
//prints 'Subscription failed (reason=IncompatibleAccount, description="Users
// account does not support this type of subscription")'
override fun toString(): String {
    val resultText = if(result) "succeeded" else "failed"
    return "Subscription $resultText (reason=$reason, description=\"$description\")"
}
```



While it might be tempting to use data classes to gain a `toString` method automatically, it's not recommended for backward compatibility reasons. Data classes are discussed in more detail in the Avoid using data classes in your API section.

Note that the state described in the `toString` method does not need to be information from the problem domain. It can relate to the status of ongoing requests (as in the example above), the health of connections to external services, or intermediate state within an ongoing operation.

For example, consider the following builder type:

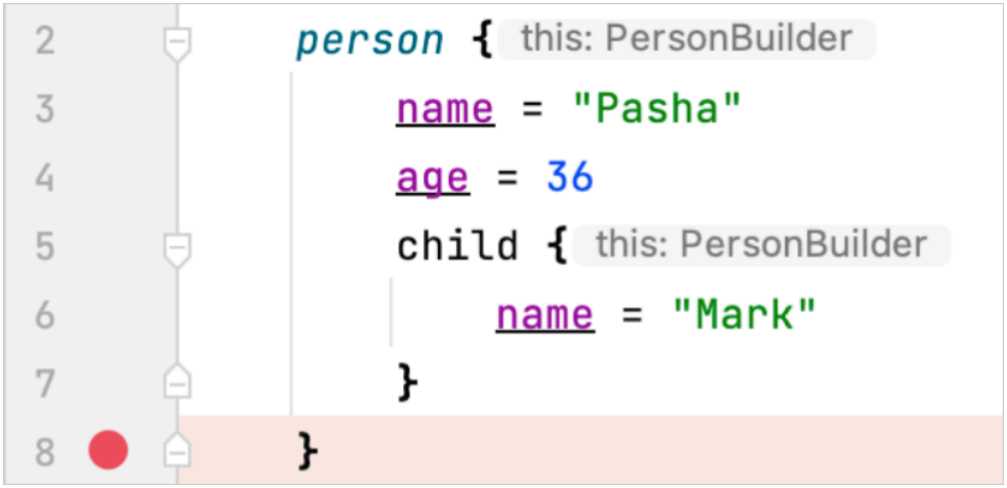
```
class Person(
    val name: String?,
    val age: Int?,
    val children: List<Person>
) {
    override fun toString(): String =
        "Person(name=$name, age=$age, children=$children)"
}

class PersonBuilder {
    var name: String? = null
    var age: Int? = null
    val children = arrayListOf<Person>()


    fun child(personBuilder: PersonBuilder.() -> Unit = {}) {
       children.add(person(personBuilder))
    }
    fun build(): Person = Person(name, age, children)
}

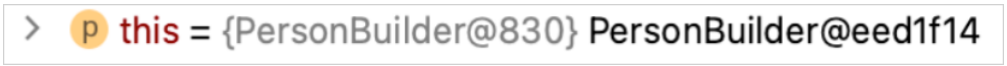
fun person(personBuilder: PersonBuilder.() -> Unit = {}): Person =
    PersonBuilder().apply(personBuilder).build()
```

This is how you would use this type:



Using the builder type example

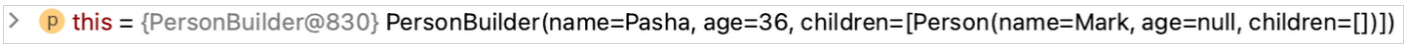
If you halt the code at the breakpoint displayed on the image above, the information displayed will not be helpful:



Adding a simple `toString` implementation results in a much more helpful output:

```
override fun toString(): String =
    "PersonBuilder(name=$name, age=$age, children=$children)"
```

With this addition, the debugger shows:



This way, you can immediately see which fields are set and which are not.

Adopt and document a policy for handling exceptions

As discussed in the Choose appropriate error handling mechanism section, there are occasions when it's appropriate for your library to throw an exception to signal an error. You may create your own exception types for this purpose.

Libraries that abstract and simplify low-level APIs will also need to handle exceptions thrown by their dependencies. A library might choose to suppress the exception, pass it on as it is, convert it to a different type of an exception, or signal the error to users in a different way.

Any of these options could be valid, depending on the context. For example:

- If a user adopts library A purely for the convenience of simplifying library B, it may be appropriate for library A to rethrow any exceptions generated by library B without modification.
- If library A adopts library B purely as an internal implementation detail, then library-specific exceptions thrown by library B should never be exposed to users of library A.

You must adopt and document a consistent approach to exception handling so users can make productive use of your library. This is especially important for debugging. Users of your library should be able to recognise, in the debugger and in logs, when an exception has originated from your library.

The type of the exception should indicate the type of the error, and the data in the exception should help the user locate the root cause of the issue. A common pattern is to wrap a low-level exception in a library-specific one, with the original exception accessible as the `cause`.

Experimental

Support coroutine stack trace recovery for custom exceptions

You can add support for coroutine stack trace recovery to custom exception types in your library to make them easier to debug. This improves your library's support for the `kotlinx.coroutines` library and other asynchronous runtimes in Kotlin.

When a coroutine receives an exception from another coroutine through a suspending function, stack trace recovery creates a copy of the exception with the stack frames that lead to that function call.

The `kotlinx.coroutines` library performs stack trace recovery automatically for exceptions with constructors that take only an exception message, a cause, both, or no arguments. If an exception type in your library requires additional constructor arguments, such as a line number or an error code, implement the `StackTraceRecoverable` interface.

To implement the interface, override the `copyForStackTraceRecovery()` function. In the override, return a new exception instance for stack trace recovery, or `null` if you don't want the `kotlinx.coroutines` library to copy the exception.

The `StackTraceRecoverable` interface is part of the Kotlin standard library, so implementing it doesn't add a dependency on the `kotlinx.coroutines` library.

Here's an example of a custom exception that preserves a `line` property when it creates a new instance for stack trace recovery:

```
import kotlin.coroutines.ExperimentalStdlibCoroutineSupportApi
import kotlin.coroutines.debug.StackTraceRecoverable

@OptIn(ExperimentalStdlibCoroutineSupportApi::class)
class FileEditException
// The implementation requires a private constructor
// to pass the cause to the IllegalStateException constructor
private constructor(
    val line: Int,
    private val detail: String,
    cause: Throwable?,
) : IllegalStateException("When editing line $line: $detail", cause),
    // Implements StackTraceRecoverable for stack trace recovery
    StackTraceRecoverable<FileEditException> {

    constructor(line: Int, detail: String) : this(line, detail, null)

    // Copies the line number and message details
    override fun copyForStackTraceRecovery(): FileEditException =
        FileEditException(line, detail, this)
    }

fun main() {
    val original = FileEditException(15, "Unexpected token")

    // Normally, you don't need to call this function directly unless you're testing its behavior
    // The kotlinx.coroutines library invokes it automatically during stack trace recovery
    val copy = original.copyForStackTraceRecovery()

    println(copy.message)
    // When editing line 15: Unexpected token

    println(copy.cause == original)
    // true
}
```

Next step

In the next part of the guide, you'll learn about testability.

Proceed to the next part

Testability

In addition to testing your library, ensure that code using your library is also testable.

Avoid global state and stateful top-level functions

Your library should not rely on state in global variables or provide stateful top-level functions as part of its public API. Such variables and functions make testing code that uses the library difficult, as tests need to find ways to control these global values.

For example, a library might define a globally accessible function that provides access to the current time:

```
val instant: Instant = Clock.now()
println(instant)
```

Any code using this API will be difficult to test, as the call to the `now()` function will always return the real current time, while in tests it's often desirable to return fake values instead.

To enable testability, the `kotlinx-datetime` library has an API that lets users get a `Clock` instance, and then use that to get the current time:

```
val clock: Clock = Clock.System
val instant: Instant = clock.now()
println(instant)
```

This allows users of a library to inject a `Clock` instance into their own classes, and replace the real implementation with a fake one during tests.

What's next

If you haven't already, consider checking out these pages:

- Learn about maintaining backward compatibility in the Backward compatibility page.
- For an extensive overview on effective documentation practices, see Informative Documentation.

Backward compatibility guidelines for library authors

The most common motivation for creating a library is to expose functionality to a wider community. This community might be a single team, a company, a particular industry, or a technology platform. In every case backward compatibility will be an important consideration. The wider the community the more important backward compatibility becomes, since you will be less aware of who your users are and what constraints they work within.

Backward compatibility is not a single term, but can be defined at the binary, source and behavioral levels. More information about these types is provided in this section.

Note that:

- It is possible to break binary compatibility without breaking source compatibility, as well as the other way around.
- It is desirable but very difficult to guarantee source compatibility. As a library author, you must consider every possible way a function or type could be invoked or instantiated by a user of the library. Source compatibility is typically an aspiration, not a promise.

The rest of this section describes actions you can take, and tools you can use to help ensure the different kinds of compatibility.

Compatibility types

Binary compatibility means that a new version of a library can replace a previously compiled version of the library. Any software that was compiled against the previous version of the library should continue to work correctly.

> Learn more about binary compatibility in the Binary compatibility validator's README or from the Evolving Java-based APIs document.

Source compatibility means that a new version of a library can replace a previous one without modifying any of the source code that uses the library. However, the outputs from compiling this client code may no longer be compatible with the outputs from compiling the library, so the client code must be rebuilt against the new version of the library to guarantee compatibility.

Behavioral compatibility means that a new version of the library does not modify the existing functionality, except to fix bugs. The same features are involved and they have the same semantics.

Choose compatible language and API versions

When publishing a library, consider both its compile-time and runtime compatibility:

- The language version determines which Kotlin compiler versions can compile code that directly uses your library.
- The API version determines the minimum Kotlin standard library version required at runtime.

In most cases, use the same language and API version.

Setting a newer language version for your library requires your consumers to use a newer Kotlin compiler version:

- On the JVM, consumers can use any compiler version from the previous language version onwards. For example, if your library uses language version 2.2, consumers can use compiler version 2.1.x, 2.2.x or later.
- On other platforms, consumers can use a compiler version the same as the library's configured language version or any later version. This requirement can delay consumers from upgrading to newer versions of your library if they can't upgrade their compiler immediately.

Your consumers also need to provide a Kotlin standard library version that is at least as new as your library's configured API version. This can make upgrades more difficult when the runtime environment is outside their control and it's hard to provide a newer standard library version, such as in Gradle or IDE plugins.

Choose the language and API versions that best fit your library. Newer versions let you adopt the latest Kotlin features, while older versions help more consumers use your library. The best choice depends on your library's use case and the number of consumers who rely on it.

Use the Binary compatibility validator

JetBrains provides a Binary compatibility validator tool, which can be used to ensure binary compatibility across different versions of your API.

This tool is implemented as a Gradle plugin, and it adds two tasks to your build:

- The `apiDump` task creates a human-readable `.api` file that describes your API.
- The `apiCheck` task compares the saved description of the API to classes compiled in the current build.

The `apiCheck` task is invoked at build time by the standard Gradle `check` task. When compatibility is broken, the build fails. At that point, you should run the `apiDump` task manually and compare the differences between the old and new versions. If you are satisfied with the changes, you can update the existing `.api` file, which resides within your VCS.

The validator has experimental support for validating KLibs produced by multiplatform libraries.

Experimental

Binary compatibility validation in the Kotlin Gradle plugin

Starting with version 2.2.0, the Kotlin Gradle plugin supports binary compatibility validation. For more information, see Binary compatibility validation in the Kotlin Gradle plugin.

Specify return types explicitly

As discussed in the Kotlin coding guidelines, you should always explicitly specify function return types and property types within the API. See also the section about Explicit API mode.

Consider the following example, where the library author creates a `JsonDeserializer` and, for convenience, uses an extension function to associate it with the `Int` type:

```
class JsonDeserializer<T>(private val fromJson: (String) -> T) {
    fun deserialize(input: String): T {
        ...
    }
}

fun Int.defaultDeserializer() = JsonDeserializer { ... }
```

Let's say the author replaces this implementation with a `JsonOrXmlDeserializer`:

```
class JsonOrXmlDeserializer<T>(
    private val fromJson: (String) -> T,
    private val fromXML: (String) -> T
) {
    fun deserialize(input: String): T {
        ...
    }
}

fun Int.defaultDeserializer() = JsonOrXmlDeserializer({ ... }, { ... })
```

Existing functionality will continue to work, with the added ability to deserialize XML. However, this breaks binary compatibility.

Avoid adding arguments to existing API functions

Adding non-default arguments to a public API breaks both binary and source compatibility, as users are required to provide more information on an invocation than before. However, even adding default arguments can break compatibility.

For example, imagine you have the following function in `lib.kt`:

```
fun fib() = … // Returns zero
```

And the following function in `client.kt`:

```
fun main() {
    println(fib()) // Prints zero
}
```

Compiling these two files on the JVM would produce the outputs `LibKt.class` and `ClientKt.class`.

Let's say you reimplement and compile the `fib` function to represent the Fibonacci sequence, such that `fib(3)` returns 2, `fib(4)` returns 3, and so on. You add a parameter but give it a default value of zero to preserve the existing behavior:

```
fun fib(input: Int = 0) = … // Returns Fibonacci member
```

You now need to recompile the file `lib.kt`. You might expect that the `client.kt` file does not need to be recompiled, and the associated class file can be invoked as follows:

```
$ kotlin ClientKt.class
```

But if you try this, a `NoSuchMethodError` occurs:

```
Exception in thread "main" java.lang.NoSuchMethodError: 'int LibKt.fib()'
       at LibKt.main(fib.kt:2)
       at LibKt.main(fib.kt)
       …
```

This is because the signature of the method changed in the bytecode generated by the Kotlin/JVM compiler, breaking binary compatibility.

Source compatibility, however, is preserved. If you recompile both files, the program will run as before.

Use overloads to preserve binary compatibility

When adding optional parameters to published APIs, you can use the Experimental `@IntroducedAt` annotation to preserve binary compatibility.

Add the annotation to each new optional parameter with the version in which it was introduced. For example:

```
@OptIn(ExperimentalVersionOverloading::class)
fun fib(@IntroducedAt("1.1") input: Int = 0) = …
```

The compiler uses this information to generate the corresponding hidden overloads.

When writing Kotlin code for the JVM, you can also use the `@JvmOverloads` annotation on functions with default arguments to generate overloads.

> The `@JvmOverloads` annotation doesn't preserve binary compatibility for Kotlin callers. Instead, use the `@IntroducedAt` annotation or add manual overloads when changing published APIs.

You can also create overloads manually instead of using a single function with default arguments. For example, if you want the `fib()` function to accept an `Int` parameter, create a separate overload:

```
fun fib() = …
fun fib(input: Int) = …
```

Avoid widening or narrowing return types

When evolving an API, it is common to want to widen or narrow the return type of a function. For example, in an upcoming version of your API, you might want to switch a return type from `List` to `Collection` or from `Collection` to `List`.

You might want to narrow the type to `List` to meet user requests for indexing support. Conversely, you might want to widen the type to `Collection` because you realize the data you are working with has no natural order.

It is easy to see why widening a return type breaks compatibility. For example, converting from `List` to `Collection` breaks all the code that uses indexing.

You might think that narrowing a return type, for example from `Collection` to `List` would preserve compatibility. Unfortunately, while source compatibility is preserved, binary compatibility is broken.

Let's say you have a demo function in the file `Library.kt`:

```
public fun demo(): Number = 3
```

And a client for the function in `Client.kt`:

```
fun main() {
    println(demo()) // Prints 3
}
```

Let's imagine a scenario where you change the return type of demo and only recompile `Library.kt`:

```
fun demo(): Int = 3
```

When you rerun the client, the following error will occur (on the JVM):

```
Exception in thread "main" java.lang.NoSuchMethodError: 'java.lang.Number Library.demo()'
        at ClientKt.main(call.kt:2)
        at ClientKt.main(call.kt)
        …
```

This happens because of the following instruction in the bytecode generated from the `main` method:

```
0: invokestatic  #12 // Method Library.demo:()Ljava/lang/Number;
```

The JVM is trying to invoke a static method called demo which returns a `Number`. However, as this method no longer exists, you have broken binary compatibility.

Avoid using data classes in your API

In regular development, the strength of data classes is the extra functions that are generated for you. In API design, this strength becomes a weakness.

For example, let's say you use the following data class in your API:

```
data class User(
    val name: String,
    val email: String
)
```

Later, you might want to add a property called `active`:

```
data class User(
    val name: String,
    val email: String,
    val active: Boolean = true
)
```

This would break binary compatibility in two ways. Firstly, the generated constructor will have a different signature. Additionally, the signature of the generated `copy` method changes.

The original signature (on Kotlin/JVM) would be:

```
public final User copy(java.lang.String, java.lang.String)
```

After adding the `active` property, the signature becomes:

```
public final User copy(java.lang.String, java.lang.String, boolean)
```

As with the constructor, this breaks binary compatibility.

It's possible to work around these issues by manually writing a secondary constructor and overriding the `copy` method. However, the effort involved negates the convenience of using a data class.

Another issue with data classes is that changing the order of constructor arguments affects the generated `componentX` methods, which are used for destructuring. Even if it does not break binary compatibility, changing the order will definitely break behavioral compatibility.

Avoid changing annotation targets

When you expose an annotation, avoid changing its allowed targets after publishing your library. Changing them can affect how the same annotation is applied when users recompile existing code.

For example, if an annotation declares only `AnnotationTarget.FIELD`, an unqualified annotation on a property applies to its backing field:

```
@Target(AnnotationTarget.FIELD)
annotation class Example

class User {
    @Example
    val name: String = ""
}
```

If you later add `AnnotationTarget.PROPERTY`, the same unqualified annotation is applied to the property instead:

```
@Target(AnnotationTarget.PROPERTY, AnnotationTarget.FIELD)
annotation class Example

class User {
    @Example
    val name: String = ""
}
```

This happens because the Kotlin compiler chooses the `property` target before the `field` target. The `field` target is used only when `property` isn't applicable.

This can break compatibility for tools and frameworks that expect the annotation on a specific generated element. In particular, the `property` target isn't visible to Java. If Java reflection or Java annotation processors need to find the annotation on the backing field, users must specify the `field` use-site target explicitly:

```
class User {
    @field:Example
    val name: String = ""
}
```

Considerations for using the PublishedApi annotation

Kotlin allows inline functions to be a part of your library's API. Calls to these functions will be inlined into the client code written by your users. This can introduce compatibility issues, so these functions are not allowed to call non-public-API declarations.

If you need to call an internal API of your library from an inlined public function, you can do so by annotating it with `@PublishedApi`. This makes the internal declaration effectively public, as references to it will end up in compiled client code. Therefore, it must be treated the same as public declarations when making changes to it, as these changes might affect binary compatibility.

Evolve APIs pragmatically

There are cases where you need to make breaking changes to your library's API over time by removing or changing an existing declaration. In this section, we'll discuss how to handle such cases pragmatically.

When users upgrade to a newer version of your library, they should not end up with unresolved references to your library's APIs in their project's source code. Instead of immediately removing something from your library's public API, you should follow a deprecation cycle. This way, you give your users time to migrate to an alternative.

Use the `@Deprecated` annotation on the old declaration to indicate that it's being replaced. The parameters of this annotation provide important details about the deprecation:

- The `message` should explain what's being changed and why.
- The `replaceWith` parameter should be used where possible to provide automatic migration to a new API.
- The deprecation's level should be used to deprecate the API gradually. For more information, see the Deprecated page of the Kotlin documentation.

Generally, a deprecation should first produce a warning, then an error, and then hide the declaration. This process should occur across several minor releases, giving users time to make any required changes in their projects. Breaking changes, such as removing an API, should happen only in major releases. A library may adopt different versioning and deprecation strategies, but this must be communicated to its users to set the correct expectations.

You can learn more in the Kotlin Evolution principles document or in the Evolving your Kotlin API painlessly for clients talk by Leonid Startsev from KotlinConf 2023.

Use the RequiresOptIn mechanism

The Kotlin standard library provides the opt-in mechanism to require explicit consent from users before they use a part of your API. This is based on creating marker annotations, which are themselves annotated with

`@RequiresOptIn`. You should use this mechanism to manage expectations concerning source and behavioral compatibility, especially when introducing new APIs to your library.

If you choose to use this mechanism, we recommend following these best practices:

- Use the opt-in mechanism to provide different guarantees to different parts of the API. For example, you could mark features as *Preview*, *Experimental*, and *Delicate*. Each category should be clearly explained in your documentation and in KDoc comments, with appropriate warning messages.
- If your library uses an experimental API, propagate the annotation to your own users. This ensures your users are aware that you have dependencies which are still evolving.
- Avoid using the opt-in mechanism to deprecate already existing declarations in your library. Use `@Deprecated` instead, as described in the Evolve APIs pragmatically section.

What's next

If you haven't already, consider checking out these pages:

- Explore strategies for minimizing mental complexity in the Minimizing mental complexity page.
- For an extensive overview on effective documentation practices, see Informative documentation.

Best practices for library authors to create informative documentation

The documentation you provide for your library is crucial. It can determine whether users investigate your library, adopt it within their projects, and persevere when they encounter difficulties. Developers today have unprecedented choice between languages, libraries, frameworks and platforms. Therefore, it is essential to engage and inform your users; otherwise they may pursue other options.

In the earliest versions of your library, feedback from users will be scarce. Fortunately, creating and refining documentation can serve as a feedback loop to greatly increase the quality of your project. As such, creating documentation should never be seen as a burden, and should not be pushed down the list of priorities when creating a library.

Effective documentation not only informs users but also drives the development and refinement of your library. Here are several key ways documentation can guide your development process:

- You should be able to explain, in a couple of paragraphs, what your library does, who will benefit from using it, and what the advantages are over alternative approaches. If you cannot do this, reconsider the scope and objectives of your project.
- You should be able to create a "Getting Started" guide that can get a potential user up and running as quickly as possible. What counts as *quickly* will depend on the problem domain, but you can compare against similar libraries on other platforms. The guide should hook the user into a feedback loop that keeps getting easier and faster while always producing reliable results. Creating this guide will help you identify sudden increases in complexity (cliff edges) that could hinder the user's progress.
- The act of documenting a function forces you to consider all the edge cases, such as valid ranges of inputs, exceptions that might be thrown, and how performance degrades as the work increases. This can often lead to improvements in the function signatures and the underlying implementation.
- If the code required to initialize your library always eclipses the code required to accomplish a task, rethink your configuration options.

- If you cannot create clear examples of performing basic tasks with standard options, consider optimizing your API for day-to-day use.
- If you cannot demonstrate how to test your library without using real data sources and online services, consider providing test doubles for components that access the network (and the outside world in general).

The sooner you provide documentation for your library, the sooner it can be tested by real-world users. Feedback from these tests can then be used to improve the design.

Provide comprehensive documentation

Your library must provide sufficient documentation to let users adopt it with minimum effort. This documentation should include:

- A Getting Started guide
- An in-depth description of the API
- Longer examples for common use cases (also known as recipes)
- Links to resources such as blogs, articles, webinars, and conference talks

The Getting Started guide should cover how your library integrates with the supported build systems. It should include a brief description of the most commonly used entities, with small examples of how they are used. At each point where the library interacts with the outside world, specify the necessary steps to configure the environment and how to verify that these steps have been completed successfully. State explicitly if no steps are necessary.

If possible, provide separate versions of the documentation for each version of the library you support. This prevents users from viewing information that is either outdated or too recent. When this is not possible, clearly mark sections of the documentation that relate to parts of the API that have been redesigned.

Create personas for your users

Creating and evaluating documentation without a clear understanding of the intended audience is challenging. Defining multiple personas for the types of users who will read your documentation can be helpful.

Consider the user's constraints, such as the pre-existing software stack within which they need to operate. Reviewers of the documentation can adopt these personas to make their conclusions more meaningful.

When you lack specific information about your users, it's best to be pessimistic. For example, do not assume expertise in the latest or most advanced features of Kotlin. Keep your code examples as simple as possible.

Personas are especially useful when real-world users cannot be consulted due to time, budgetary constraints, or confidentiality agreements. Over time, as you gain a better understanding of your users, refine the personas to match their needs more accurately.

Document by example whenever possible

Documentation by example is one of the most cost-effective ways to explain basic concepts to users. Whenever possible, provide simple and clear code examples that help to explain or demonstrate the current topic or concept being discussed.

The KDoc documentation format lets you use inline markup using Markdown in your documentation comments. Use inline code snippets in comments to showcase the usage of an API. For an example, see the source code and rendered documentation of the test dispatchers of the coroutines library.

Providing examples like this can avoid the need to write lengthy descriptions of expected inputs, possible outputs, and failure modes. However, the context for each example needs to be clear, as do the circumstances in which it is relevant. Simply providing a folder of uncommented sample programs does not qualify as documentation.

Thoroughly document your API

Every supported API entry point should be documented using KDoc.

Kotlin's documentation engine, Dokka, includes only public declarations in its outputs by default. As discussed in the Simplicity section, you should minimize your public API and remove public entry points that you don't want users to access. If there are APIs you cannot hide from users by controlling their visibility, omit them from the documentation using the suppress directive.

Begin the description of entry points with a clear, high-level description of what the function does. Avoid simply restating the signature in natural language.

For example, don't say “takes a `String` and returns a `Connection` ” but instead perhaps “Attempts to connect to the database specified by the input string, returning a `Connection` if successful, and throwing a `ConnectionTimeoutException` otherwise”.

Specify each input's expected values and behavior with different inputs. Explain the range of valid values and what happens when invalid values are provided. For example, if a string input is supposed to be a URL, describe what happens when the string is empty, invalid, uses an unsupported protocol, or refers to a location that does not exist.

Document every exception an API entry point may throw. Discuss failure conditions in the general description and reserve the exception section for detailed information. This enhances readability and helps the reader focus. Instead, include this information organically into the general description. Providing usage examples whenever possible also helps users understand how to use the API correctly.

> We recommend learning about technical writing to improve the clarity and effectiveness of your documentation. Consider exploring resources, such as this course from Google (part one and part two).

Document lambda parameters

When an API entry point takes a lambda, the user is providing some functionality that your library will execute on their behalf. This requires extra documentation in at least two areas.

First, document what happens if the lambda throws an exception. Consider addressing the following questions:

- Will this cause an immediate failure, will the lambda be invoked repeatedly, or is there a fallback behavior?
- If the calling function needs to exit, will it rethrow the exception thrown from the lambda or a different one?
- If the exception is different, will it include the original?

Additionally, unless the function is declared as inline,document any special behavior relating to concurrency. Ensure the following are covered:

- Will the lambda be invoked in the same thread as the caller?
- If the lambda is not invoked in the same thread as the caller, which thread (or thread pool) will it be invoked in?
- Could multiple copies of the lambda be run in parallel?
- Which other jobs of work may be using that thread?

- Can the user specify a thread for the library to use?
- Where multiple lambdas are being invoked, what guarantees are provided regarding sequencing?

Use explicit links in documentation

It is very rare for an API entry point to be completely independent of other functionality in your library. Typically, calls must be made in a specific sequence, there are multiple options for performing a particular task, and entry points that perform related tasks are used in similar ways. For example, functions like `format` and `parse` mirror each other.

Use the `@see` tag or internal links to make these relationships explicit in your documentation. This helps the reader by enabling them to chunk the information together, building a better-integrated mental map of the library.

Be self-contained where possible

When describing what makes an input valid, it is tempting to simply refer to the relevant standard, such as those produced by the W3C, IEEE, or Unicode Consortium. While providing these links can be helpful, the reader should not be forced to refer to an external specification to discover basic information, like the set of whitespace characters.

Wherever possible, the documentation should be self-contained. It should provide enough information for the user to understand the typical use of each API entry point. You can refer the user to external documents for edge cases that will not happen in common use.

Use simple English

It's important to use simple and clear English when creating documentation. This ensures that your content is accessible to a global audience, including those whose first language is not English. Avoid using complex words, jargon, Latin phrases, or idiomatic expressions that might confuse readers. Instead, use straightforward language and concise sentences.

Simple English also makes the documentation easier to translate if needed. Clear, unambiguous text reduces the risk of misinterpretation and improves the overall readability.

What's next

If you haven't already, consider checking out these pages:

- Explore strategies for minimizing mental complexity in the Minimizing mental complexity page.
- Learn about maintaining backward compatibility in the Backward compatibility page.

Building a Kotlin library for multiplatform

When creating a Kotlin library, consider building and publishing it with support for Kotlin Multiplatform. This broadens the target audience of your library, making it compatible with projects targeting multiple platforms.

> You can find already available Kotlin Multiplatform libraries for different use cases and target platforms on klibs.io.

The following sections provide guidelines to help you build a Kotlin Multiplatform library effectively.

Maximize your reach

To make your library available to the largest number of projects as a dependency, aim to support as many target platforms of Kotlin Multiplatform as possible.

If your library doesn't support the platforms used by a multiplatform project, whether it's a library or an application, it becomes difficult for that project to depend on your library. In that case, projects can use your library for some platforms and need to implement separate solutions for others, or they will choose an alternative library altogether that supports all their platforms.

To streamline artifact production, use cross-compilation to publish Kotlin Multiplatform libraries from any host. This allows you to generate `.klib` artifacts for Apple targets without an Apple machine.

For Kotlin/Native targets, consider using a tiered approach to support all possible targets.

Design APIs for use from common code

When creating a library, design the APIs to be usable from common Kotlin code, instead of writing platform-specific implementations.

Provide reasonable default configurations when possible and include platform-specific configuration options. Good defaults allow users to use the library's APIs from common Kotlin code, without the need to write platform-specific implementations to configure the library.

Place APIs in the broadest relevant source set using the following priority:

- **The `commonMain` source set:** APIs in the `commonMain` source set are available to all platforms the library supports. Aim to place most of your library's API here.
- **Intermediate source sets:** If some platforms don't support certain APIs, use intermediate source sets to target specific platforms. For example, you can create a `concurrent` source set for targets that support multi-threading or a `nonJvm` source set for all non-JVM targets.
- **Platform-specific source sets:** For platform-specific APIs, use source sets like `androidMain`.

To learn more about the source sets of Kotlin Multiplatform projects, see Hierarchical project structure.

Ensure consistent behavior across platforms

To ensure your library behaves consistently on all supported platforms, APIs in a multiplatform library should accept the same range of valid inputs, perform the same actions, and return the same results across all platforms. Similarly, the library should treat invalid inputs uniformly and report errors or throw exceptions consistently on all platforms.

Inconsistent behavior makes the library difficult to use and forces users to add conditional logic in common code to manage platform-specific differences.

You can use `expect` and `actual` declarations to declare functions in common code with platform-specific implementations that have full access to the native APIs of each platform. These implementations must also have the same behavior to ensure that they can be used reliably from common code.

When APIs behave consistently across platforms, they only need to be documented once in the `commonMain` source set.

> If platform differences are unavoidable, such as when one platform supports a broader set of inputs, minimize them where possible. For example, you might not want to limit one platform's functionality to match the others. In such cases, clearly document the specific differences.

Test on all platforms

Multiplatform libraries can have multiplatform tests written in common code that runs on all platforms. Executing this common test suite regularly on your supported platforms can ensure that the library behaves correctly and consistently.

Regularly testing Kotlin/Native targets across all published platforms can be challenging. However, to ensure broader compatibility, consider publishing the library for all the targets it can support, using a tiered method when testing compatibility.

Use the `kotlin-test` library to write tests in common code and execute them with platform-specific test runners.

Consider non-Kotlin users

Kotlin Multiplatform offers interoperability with native APIs and languages across its supported target platforms. When creating a Kotlin Multiplatform library, consider whether users might need to use your library's types and declarations from languages other than Kotlin.

For example, if some types from your library will be exposed to Swift code through interoperability, design those types to be easily accessible from Swift. The Kotlin-Swift interopedia provides useful insights into how Kotlin APIs appear when called from Swift.

Promote your library

You can feature your library on klibs.io, the search platform where developers discover and evaluate Kotlin Multiplatform libraries.

klibs.io automatically lists libraries that meet its listing criteria. To check whether your library qualifies, see the FAQ.

Get started with Kotlin custom scripting – tutorial

> Kotlin custom scripting is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin scripting is the technology that enables executing Kotlin code as scripts without prior compilation or packaging into executables.

For an overview of Kotlin scripting with examples, check out the talk Implementing the Gradle Kotlin DSL by Rodrigo Oliveira from KotlinConf'19.

In this tutorial, you'll create a Kotlin scripting project that executes arbitrary Kotlin code with Maven dependencies. You'll be able to execute scripts like this:

```
@file:Repository("https://maven.pkg.jetbrains.space/public/p/kotlinx-html/maven")
@file:DependsOn("org.jetbrains.kotlinx:kotlinx-html-jvm:0.7.3")

import kotlinx.html.*
import kotlinx.html.stream.*
import kotlinx.html.attributes.*

val addressee = "World"

print(
    createHTML().html {
        body {
            h1 { +"Hello, $addressee!" }
        }
    }
)
```

The specified Maven dependency (`kotlinx-html-jvm` for this example) will be resolved from the specified Maven repository or local cache during execution and used for the rest of the script.

Project structure

A minimal Kotlin custom scripting project contains two parts:

- *Script definition* – a set of parameters and configurations that define how this script type should be recognized, handled, compiled, and executed.
- *Scripting host* – an application or component that handles script compilation and execution – actually running scripts of this type.

With all of this in mind, it's best to split the project into two modules.

Before you start

Download and install the latest version of IntelliJ IDEA.

Create a project

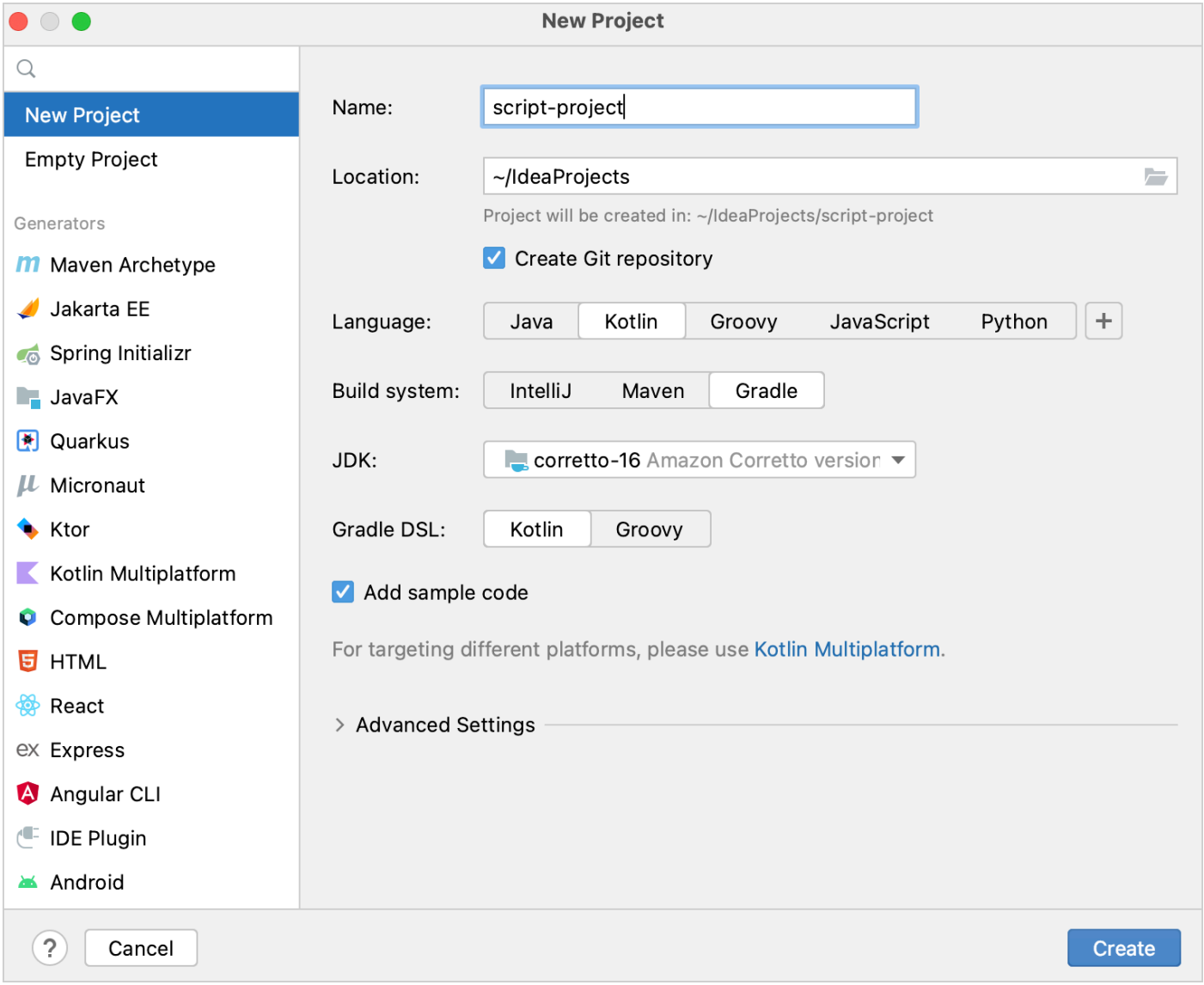
1. In IntelliJ IDEA, select **File** | **New** | **Project**.
2. In the panel on the left, select **New Project**.
3. Name the new project and change its location if necessary.

   > Select the **Create Git repository** checkbox to place the new project under version control. You will be able to do it later at any time.

4. From the **Language** list, select **Kotlin**.
5. Select the **Gradle** build system.
6. From the **JDK** list, select the JDK that you want to use in your project.
   - If the JDK is installed on your computer, but not defined in the IDE, select **Add JDK** and specify the path to the JDK home directory.

- If you don't have the necessary JDK on your computer, select **Download JDK**.

7. Select the Kotlin or Gradle language for the **Gradle DSL**.
8. Click **Create**.

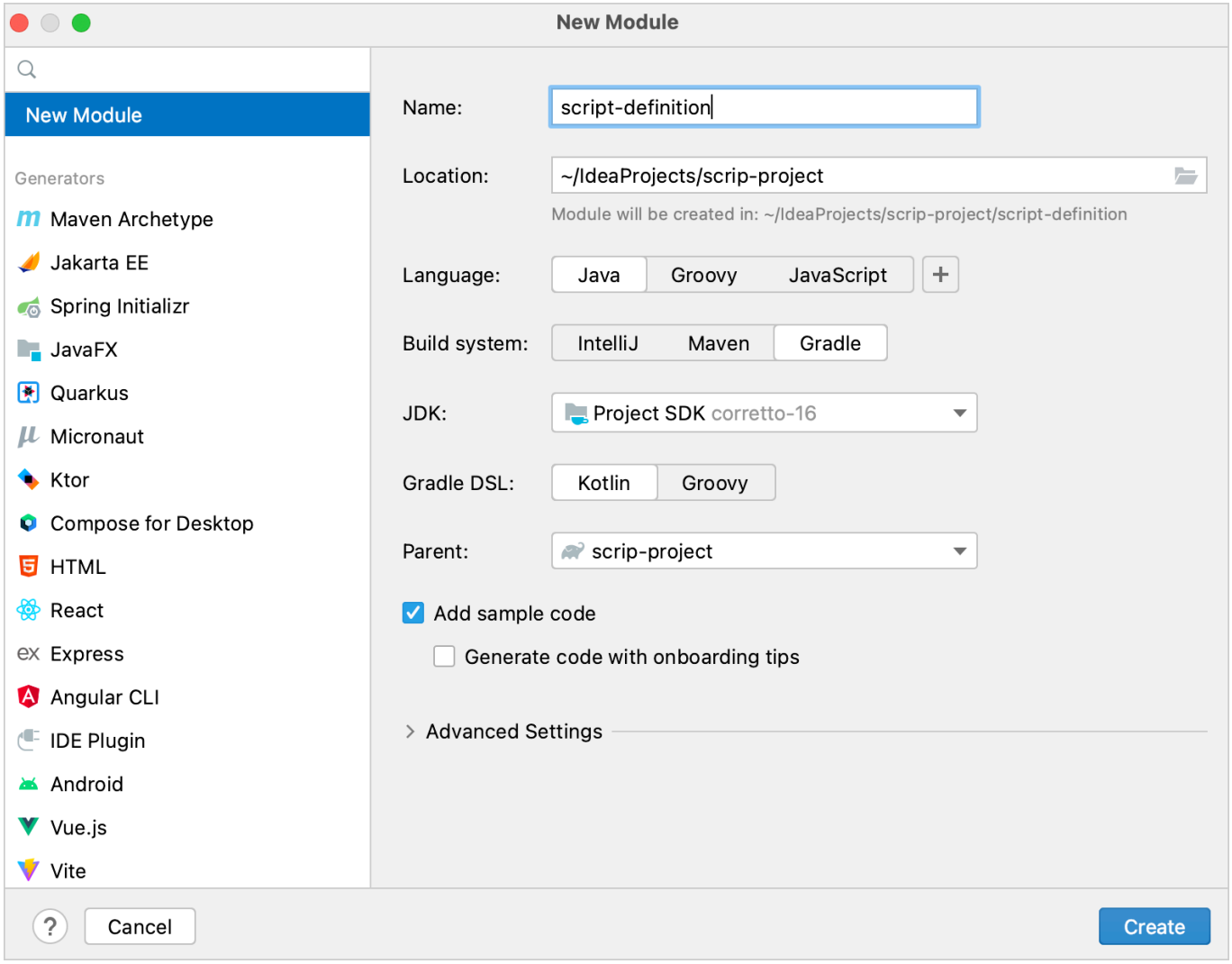


Create a root project for custom Kotlin scripting

Add scripting modules

Now you have an empty Kotlin/JVM Gradle project. Add the required modules, script definition and scripting host:

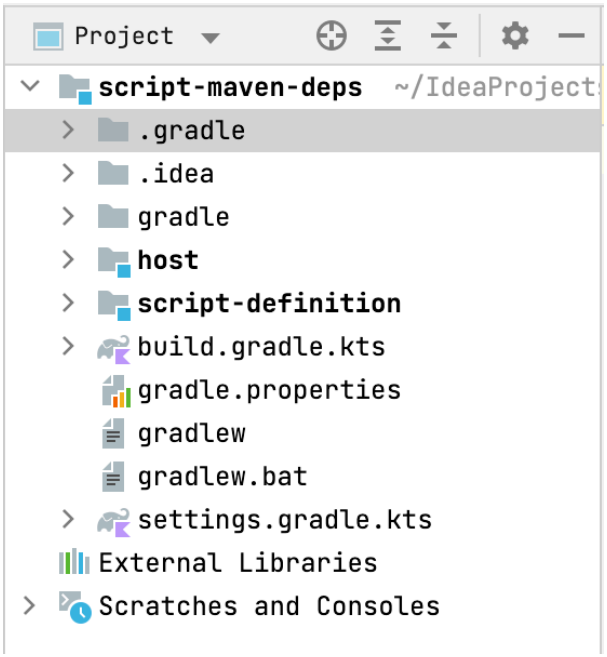
1. In IntelliJ IDEA, select **File | New | Module**.
2. In the panel on the left, select **New Module**. This module will be the script definition.
3. Name the new module and change its location if necessary.
4. From the **Language** list, select **Java**.
5. Select the **Gradle** build system and Kotlin for the **Gradle DSL** if you want to write the build script in Kotlin.
6. As a module's parent, select the root module.
7. Click **Create**.



Create script definition module

8. In the module's `build.gradle(.kts)` file, remove the `version` of the Kotlin Gradle plugin. It is already in the root project's build script.
9. Repeat previous steps one more time to create a module for the scripting host.

The project should have the following structure:



Custom scripting project structure

You can find an example of such a project and more Kotlin scripting examples in the kotlin-script-examples GitHub repository.

Create a script definition

First, define the script type: what developers can write in scripts of this type and how it will be handled. In this tutorial, this includes support for the `@Repository` and `@DependsOn` annotations in the scripts.

1. In the script definition module, add the dependencies on the Kotlin scripting components in the `dependencies` block of `build.gradle(.kts)`. These dependencies provide the APIs you will need for the script definition:

Kotlin

```
dependencies {
    implementation("org.jetbrains.kotlin:kotlin-scripting-common")
    implementation("org.jetbrains.kotlin:kotlin-scripting-jvm")
    implementation("org.jetbrains.kotlin:kotlin-scripting-dependencies")
    implementation("org.jetbrains.kotlin:kotlin-scripting-dependencies-maven")
    // coroutines dependency is required for this particular definition
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0")
}
```

Groovy

```
dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-scripting-common'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-jvm'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-dependencies'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-dependencies-maven'
    // coroutines dependency is required for this particular definition
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0'
}
```

2. Create the `src/main/kotlin/` directory in the module and add a Kotlin source file, for example, `scriptDef.kt`.
3. In `scriptDef.kt`, create a class. It will be a superclass for scripts of this type, so declare it `abstract` or `open`.

```
// abstract (or open) superclass for scripts of this type
abstract class ScriptWithMavenDeps
```

   This class will also serve as a reference to the script definition later.

4. To make the class a script definition, mark it with the `@KotlinScript` annotation. Pass two parameters to the annotation:
   - `fileExtension` – a string ending with `.kts` that defines a file extension for scripts of this type.
   - `compilationConfiguration` – a Kotlin class that extends `ScriptCompilationConfiguration` and defines the compilation specifics for this script definition. You'll create it in the next step.

```
// @KotlinScript annotation marks a script definition class
@KotlinScript(
    // File extension for the script type
    fileExtension = "scriptwithdeps.kts",
    // Compilation configuration for the script type
    compilationConfiguration = ScriptWithMavenDepsConfiguration::class
)
abstract class ScriptWithMavenDeps

object ScriptWithMavenDepsConfiguration: ScriptCompilationConfiguration()
```

> In this tutorial, we provide only the working code without explaining Kotlin scripting API. You can find the same code with a detailed explanation on GitHub.

5. Define the script compilation configuration as shown below.

```
object ScriptWithMavenDepsConfiguration : ScriptCompilationConfiguration(
    {
        // Implicit imports for all scripts of this type
        defaultImports(DependsOn::class, Repository::class)
        jvm {
            // Extract the whole classpath from context classloader and use it as dependencies
            dependenciesFromCurrentContext(wholeClasspath = true)
        }
        // Callbacks
        refineConfiguration {
            // Process specified annotations with the provided handler
            onAnnotations(DependsOn::class, Repository::class, handler =
::configureMavenDepsOnAnnotations)
        }
    }
)
```

The `configureMavenDepsOnAnnotations` function is as follows:

```
// Handler that reconfigures the compilation on the fly
fun configureMavenDepsOnAnnotations(context: ScriptConfigurationRefinementContext):
ResultWithDiagnostics<ScriptCompilationConfiguration> {
    val annotations = context.collectedData?.get(ScriptCollectedData.collectedAnnotations)?.takeIf
{ it.isNotEmpty() }
        ?: return context.compilationConfiguration.asSuccess()
    return runBlocking {
        resolver.resolveFromScriptSourceAnnotations(annotations)
    }.onSuccess {
        context.compilationConfiguration.with {
            dependencies.append(JvmDependency(it))
        }.asSuccess()
    }
}

private val resolver = CompoundDependenciesResolver(FileSystemDependenciesResolver(),
MavenDependenciesResolver())
```

You can find the full code here.

Create a scripting host

The next step is creating the scripting host – the component that handles the script execution.

1. In the scripting host module, add the dependencies in the `dependencies` block of `build.gradle(.kts)`:
   - Kotlin scripting components that provide the APIs you need for the scripting host
   - The script definition module you created previously

Kotlin

```
dependencies {
    implementation("org.jetbrains.kotlin:kotlin-scripting-common")
    implementation("org.jetbrains.kotlin:kotlin-scripting-jvm")
    implementation("org.jetbrains.kotlin:kotlin-scripting-jvm-host")
    implementation(project(":script-definition")) // the script definition module
}
```

Groovy

```
dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-scripting-common'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-jvm'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-jvm-host'
    implementation project(':script-definition') // the script definition module
}
```

2. Create the `src/main/kotlin/` directory in the module and add a Kotlin source file, for example, `host.kt`.

3. Define the `main` function for the application. In its body, check that it has one argument – the path to the script file – and execute the script. You'll define the script execution in a separate function `evalFile` in the next step. Declare it empty for now.

   `main` can look like this:

```
fun main(vararg args: String) {
     if (args.size != 1) {
         println("usage: <app> <script file>")
     } else {
         val scriptFile = File(args[0])
         println("Executing script $scriptFile")
         evalFile(scriptFile)
     }
 }
```

4. Define the script evaluation function. This is where you'll use the script definition. Obtain it by calling `createJvmCompilationConfigurationFromTemplate` with the script definition class as a type parameter. Then call `BasicJvmScriptingHost().eval`, passing it the script code and its compilation configuration. `eval` returns an instance of `ResultWithDiagnostics`, so set it as your function's return type.

```
fun evalFile(scriptFile: File): ResultWithDiagnostics<EvaluationResult> {
     val compilationConfiguration =
createJvmCompilationConfigurationFromTemplate<ScriptWithMavenDeps>()
     return BasicJvmScriptingHost().eval(scriptFile.toScriptSource(), compilationConfiguration,
null)
 }
```

5. Adjust the `main` function to print information about the script execution:

```
fun main(vararg args: String) {
     if (args.size != 1) {
         println("usage: <app> <script file>")
     } else {
         val scriptFile = File(args[0])
         println("Executing script $scriptFile")
         val res = evalFile(scriptFile)
         res.reports.forEach {
             if (it.severity > ScriptDiagnostic.Severity.DEBUG) {
                 println(" : ${it.message}" + if (it.exception == null) "" else ":
${it.exception}")
             }
         }
     }
 }
```

You can find the full code here

Run scripts

To check how your scripting host works, prepare a script to execute and a run configuration.

1. Create the file `html.scriptwithdeps.kts` with the following content in the project root directory:

```
@file:Repository("https://maven.pkg.jetbrains.space/public/p/kotlinx-html/maven")
@file:DependsOn("org.jetbrains.kotlinx:kotlinx-html-jvm:0.7.3")

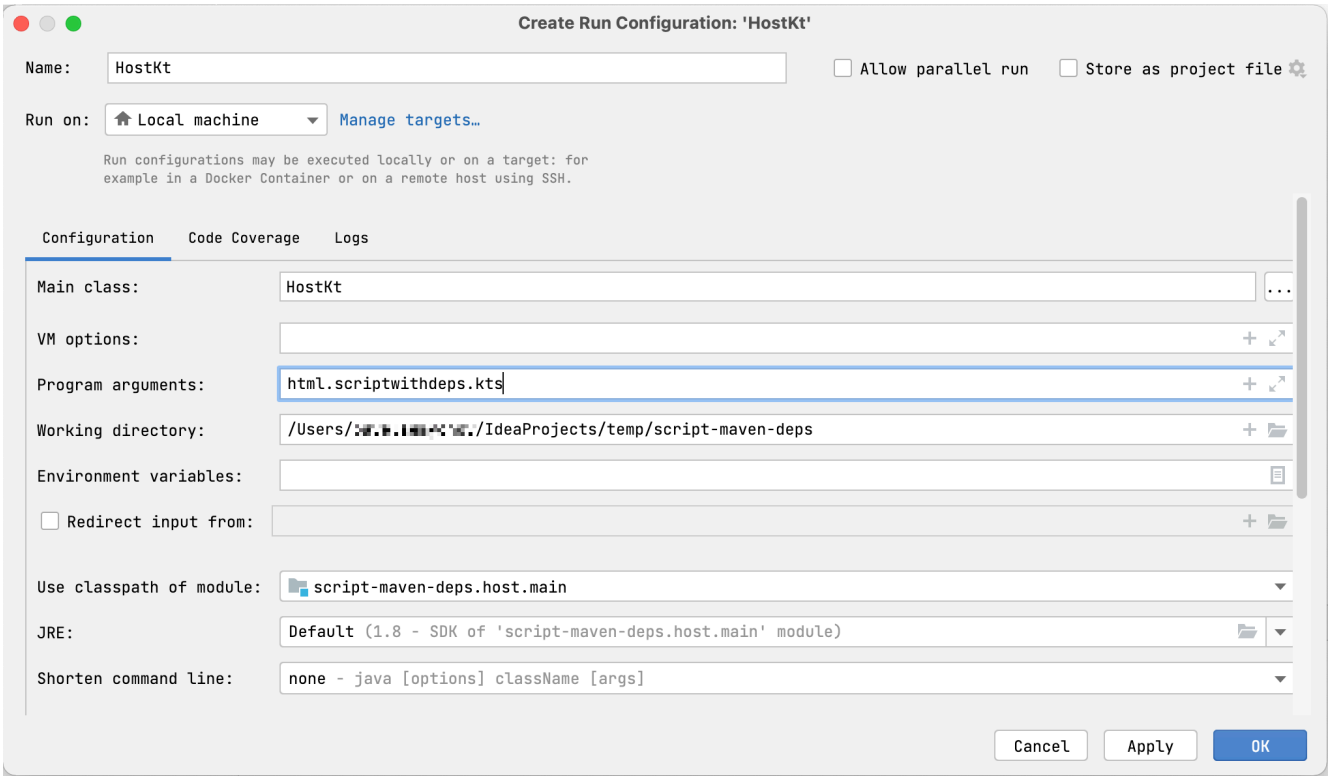
import kotlinx.html.*; import kotlinx.html.stream.*; import kotlinx.html.attributes.*

val addressee = "World"

print(
    createHTML().html {
        body {
            h1 { +"Hello, $addressee!" }
        }
    }
)
```

   It uses functions from the `kotlinx-html-jvm` library which is referenced in the `@DependsOn` annotation argument.

2. Create a run configuration that starts the scripting host and executes this file:
   1. Open `host.kt` and navigate to the `main` function. It has a **Run** gutter icon on the left.
   2. Right-click the gutter icon and select **Modify Run Configuration**.
   3. In the **Create Run Configuration** dialog, add the script file name to **Program arguments** and click **OK**.



Scripting host run configuration

3. Run the created configuration.

You'll see how the script is executed, resolving the dependency on `kotlinx-html-jvm` in the specified repository and printing the results of calling its functions:

```
<html>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

Resolving dependencies may take some time on the first run. Subsequent runs will complete much faster because they use downloaded dependencies from the local Maven repository.

What's next?

Once you've created a simple Kotlin scripting project, find more information on this topic:

- Read the Kotlin scripting KEEP
- Browse more Kotlin scripting examples
- Watch the talk Implementing the Gradle Kotlin DSL by Rodrigo Oliveira

IDEs for Kotlin development

JetBrains provides official Kotlin support for the following IDEs and code editors: IntelliJ IDEA and Android Studio. You can also install the official Kotlin by JetBrains extension for Visual Studio Code, which is currently in Alpha.

Other IDEs and code editors only have Kotlin community-supported plugins.

IntelliJ IDEA

IntelliJ IDEA is an IDE designed for JVM languages, such as Kotlin and Java, to maximize developer productivity. It does the routine and repetitive tasks for you by providing clever code completion, static code analysis, and refactorings. It lets you focus on the bright side of software development, making it not only productive but also an enjoyable experience.

The Kotlin plugin is bundled with each IntelliJ IDEA release. Each IDEA release introduces new features and upgrades that improve the experience for Kotlin developers in the IDE. See What's new in IntelliJ IDEA for the latest updates and improvements for Kotlin.

Read more about IntelliJ IDEA in the official documentation.

Android Studio

Android Studio is the official IDE for Android app development, based on IntelliJ IDEA. On top of IntelliJ's powerful code editor and developer tools, Android Studio offers even more features that enhance your productivity when building Android apps.

Kotlin plugin is bundled with each Android Studio release.

Read more about Android Studio in the official documentation.

Visual Studio Code

Visual Studio Code is a code editor with a wide range of extensions, including the official Kotlin by JetBrains extension.

The Kotlin extension provides code completion, navigation, debugging, and other Kotlin development features through the Kotlin Language Server.

For more information, see Kotlin Language Server and Visual Studio Code.

Other IDEs support

JetBrains doesn't provide official Kotlin plugins for other IDEs. You can use the Kotlin Language Server with other code editors.

To use Kotlin in text editors without IDE-related features (such as code formatting, debugging tools, refactoring), you can download the latest Kotlin command-line compiler (`kotlin-compiler-2.4.20.zip`) from Kotlin GitHub Releases and install it manually. Also, you could use package managers, such as Homebrew, SDKMAN!, and Snap package.

Compatibility with the Kotlin language versions

For IntelliJ IDEA and Android Studio, the Kotlin plugin is bundled with each release. When the new Kotlin version is released, these tools will suggest updating Kotlin to the latest version automatically. See the latest supported language version in Kotlin releases.

What's next?

- Create a console application in IntelliJ IDEA
- Create your first cross-platform mobile app using IntelliJ IDEA or Android Studio

Kotlin Language Server

The Kotlin Language Server is JetBrains' official implementation of the Language Server Protocol (LSP) for Kotlin.

The server is based on IntelliJ IDEA, the IntelliJ IDEA Kotlin plugin, JetBrains AIR, and Fleet. It is designed to work with any code editor that supports LSP.

IntelliJ IDEA and Android Studio provide the best Kotlin development experience.

Kotlin in Visual Studio Code

Kotlin Language Server provides official Kotlin language support for Visual Studio Code.

If you use Visual Studio Code for Kotlin development, install the official Kotlin by JetBrains extension from the Visual Studio Marketplace.

For VS Code-compatible editors, such as Cursor or VSCodium, install the extension from the Open VSX Registry.

To activate the **Kotlin by JetBrains** extension, open a Kotlin project in Visual Studio Code, and then open any Kotlin file.

Supported features

The Kotlin Language Server includes core language features such as:

- Support for up-to-date Kotlin language versions
- IntelliJ-powered code completion
- IntelliJ-powered diagnostics and quick fixes for Kotlin and `kotlinx.*` libraries
- Build system support for JVM projects: Gradle, Maven, and experimental Android Gradle Plugin support

  > Support for Kotlin Multiplatform projects is under development.

- Semantic highlighting
- Organize imports
- Rename refactoring
- Code formatting
- Documentation navigation and hover support
- Call hierarchy
- Code folding

Feedback

The Kotlin Language Server is under active development, and feedback is especially valuable during the Alpha stage.

If you run into problems or want to suggest improvements, report them in the Kotlin LSP repository.

What's next?

- Explore the Kotlin Language Server repository on GitHub

Introduction

Dokka is an API documentation engine for Kotlin.

Just like Kotlin itself, Dokka supports mixed-language projects. It understands Kotlin's KDoc comments and Java's Javadoc comments.

Dokka can generate documentation in multiple formats, including its own modern HTML format, Java's Javadoc HTML, as well as the GFM and Jekyll flavors of Markdown.

Here are some libraries that use Dokka for their API reference documentation:

- kotlinx.coroutines

- Bitmovin
- Hexagon
- Ktor
- OkHttp

You can run Dokka using Gradle, Maven or from the command line. It is also highly pluggable.

See Get started with Dokka to take your first steps in using Dokka.

Community

Dokka has a dedicated `#dokka` channel in Kotlin Community Slack where you can chat about Dokka, its plugins and how to develop them, as well as get in touch with maintainers.

Get started with Dokka

Below you can find simple instructions to help you get started with Dokka.

> This guide applies to Dokka Gradle plugin (DGP) v2 mode. The DGP v1 mode is no longer supported. To upgrade from v1 to v2 mode, follow the Migration guide.

Apply the Gradle Dokka plugin

Apply the Dokka Gradle plugin (DGP) in the root build script of your project:

Gradle Kotlin DSL

```
plugins {
    id("org.jetbrains.dokka") version "2.2.0"
}
```

Document multi-project builds

When documenting multi-project builds, apply the plugin to every subproject you want to document. Share Dokka configuration across subprojects by using one of the following approaches:

- Convention plugin
- Direct configuration in each subproject if you're not using convention plugins

For more information about sharing Dokka configuration in multi-project builds, see Multi-project configuration.

Generate documentation

To generate documentation, run the following Gradle task:

```
./gradlew :dokkaGenerate
```

This task works for both single and multi-project builds.

Run the `dokkaGenerate` task from the aggregating project by prefixing the task with its project path (`:`). For example:

```
./gradlew :dokkaGenerate

// OR

./gradlew :aggregatingProject:dokkaGenerate
```

Avoid running `./gradlew dokkaGenerate` instead of `./gradlew :dokkaGenerate` or `./gradlew :aggregatingProject:dokkaGenerate`. Without a project path (`:`) prefixing the task, Gradle tries to run all `dokkaGenerate` tasks across the entire build, which may trigger unnecessary work.

You can use different tasks to generate output in HTML, Javadoc or both HTML and Javadoc.

To learn more about using Dokka with Gradle, see Gradle.

This guide applies to Dokka Gradle plugin (DGP) v2 mode. The DGP v1 mode is no longer supported. To upgrade from v1 to v2 mode, follow the Migration guide.

Apply the Gradle Dokka plugin

Apply the Gradle plugin for Dokka in the root build script of your project:

Gradle Groovy DSL

```
plugins {
    id 'org.jetbrains.dokka' version '2.2.0'
}
```

Document multi-project builds

When documenting multi-project builds, you need to apply the plugin to every subproject you want to document. Share Dokka configuration across subprojects by using one of the following approaches:

- Convention plugin
- Direct configuration in each subproject if you're not using convention plugins

For more information about sharing Dokka configuration in multi-project builds, see Multi-project configuration.

Generate documentation

To generate documentation, run the following Gradle task:

```
./gradlew :dokkaGenerate
```

This task works for both single and multi-project builds.

Run the `dokkaGenerate` task from the aggregating project by prefixing the task with its project path. For example:

```
./gradlew :dokkaGenerate

// OR

./gradlew :aggregatingProject:dokkaGenerate
```

Avoid running `./gradlew dokkaGenerate` instead of `./gradlew :dokkaGenerate` or `./gradlew :aggregatingProject:dokkaGenerate`. Without a project path (`:`) prefixing the task, Gradle tries to run all `dokkaGenerate` tasks across the entire build, which may trigger unnecessary work.

You can use different tasks to generate output in HTML, Javadoc or both HTML and Javadoc.

To learn more about using Dokka with Gradle, see Gradle.

Add the Maven plugin for Dokka to the `plugins` section of your POM file:

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.dokka</groupId>
            <artifactId>dokka-maven-plugin</artifactId>
            <version>2.2.0</version>
            <executions>
                <execution>
                    <phase>pre-site</phase>
                    <goals>
                        <goal>dokka</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

To generate documentation, run the `dokka:dokka` goal.

By default, the output directory is set to `target/dokka`.

To learn more about using Dokka with Maven, see Maven.

Gradle

This guide applies to Dokka Gradle plugin (DGP) v2 mode. The previous DGP v1 mode is no longer supported. If you're upgrading from v1 to v2 mode, see the Migration guide.

To generate documentation for a Gradle-based project, you can use the Gradle plugin for Dokka.

The Dokka Gradle plugin (DGP) comes with basic autoconfiguration for your project, includes Gradle tasks for generating documentation, and provides configuration options to customize the output.

You can play around with Dokka and explore how to configure it for various projects in our Gradle example projects.

Supported versions

Ensure that your project meets the minimum version requirements:

Tool	Version
Gradle	7.6 or higher
Android Gradle plugin	7.0 or higher

Tool	Version
Kotlin Gradle plugin	1.9 or higher

Apply Dokka

The recommended way of applying the Gradle plugin for Dokka is with the plugins block. Add it in the `plugins {}` block of your project's `build.gradle.kts` file:

Gradle Kotlin DSL

```
plugins {
    id("org.jetbrains.dokka") version "2.2.0"
}
```

Gradle Groovy DSL

```
plugins {
    id 'org.jetbrains.dokka' version '2.2.0'
}
```

When documenting multi-project builds, you need to apply the plugin explicitly to every subproject you want to document. You can configure Dokka directly in each subproject or share Dokka configuration across subprojects using a convention plugin. For more information, see how to configure single-project and multi-project builds.

- Under the hood, Dokka uses the Kotlin Gradle plugin to automatically configure source sets for which documentation is generated. Make sure to apply the Kotlin Gradle Plugin or configure source sets manually.
- If you are using Dokka in a precompiled script plugin, add the Kotlin Gradle plugin as a dependency to ensure it works properly.

Enable build cache and configuration cache

DGP supports Gradle build cache and configuration cache, improving build performance.

- To enable build cache, follow instructions in the Gradle build cache documentation.
- To enable configuration cache, follow instructions in the Gradle configuration cache documentation.

Generate documentation

The Dokka Gradle plugin comes with HTML and Javadoc output formats built in.

Use the following Gradle task to generate documentation:

```
./gradlew :dokkaGenerate
```

The key behavior of the `dokkaGenerate` Gradle task is:

- This task generates documentation for both single and multi-project builds.
- By default, the documentation output format is HTML. You can also generate Javadoc or both HTML and Javadoc formats by adding the appropriate plugins.
- The generated documentation is automatically placed in the `build/dokka/html` directory for both single and multi-project builds. You can change the location (`outputDirectory`).

Configure documentation output format

> The Javadoc output format is in Alpha. You may find bugs and experience migration issues when using it. Successful integration with tools that accept Javadoc as input is not guaranteed. Use it at your own risk.

You can choose to generate the API documentation in HTML, Javadoc, or both formats at the same time:

1. Place the corresponding plugin `id` in the `plugins {}` block of your project's `build.gradle.kts` file:

```
plugins {
    // Generates HTML documentation
    id("org.jetbrains.dokka") version "2.2.0"

    // Generates Javadoc documentation
    id("org.jetbrains.dokka-javadoc") version "2.2.0"

    // Keeping both plugin IDs generates both formats
}
```

2. Run the corresponding Gradle task.

   Here is a list of the plugin `id` and Gradle task that correspond to each format:

	HTML	Javadoc	Both
Plugin `id`	`id("org.jetbrains.dokka")`	`id("org.jetbrains.dokka-javadoc")`	Use both HTML and Javadoc plugins
Gradle task	`./gradlew :dokkaGeneratePublicationHtml`	`./gradlew :dokkaGeneratePublicationJavadoc`	`./gradlew :dokkaGenerate`

> - The `dokkaGenerate` task generates documentation in all available formats based on the applied plugins. If both the HTML and Javadoc plugins are applied, you can choose to generate only HTML by running the `dokkaGeneratePublicationHtml` task, or only Javadoc by running the `dokkaGeneratePublicationJavadoc` task.

If you're using IntelliJ IDEA, you may see the `dokkaGenerateHtml` Gradle task. This task is simply an alias of `dokkaGeneratePublicationHtml` . Both tasks perform exactly the same operation.

Aggregate documentation output in multi-project builds

Dokka can aggregate documentation from multiple subprojects into a single output or publication.

You have to apply the Dokka plugin to all documentable subprojects before aggregating the documentation.

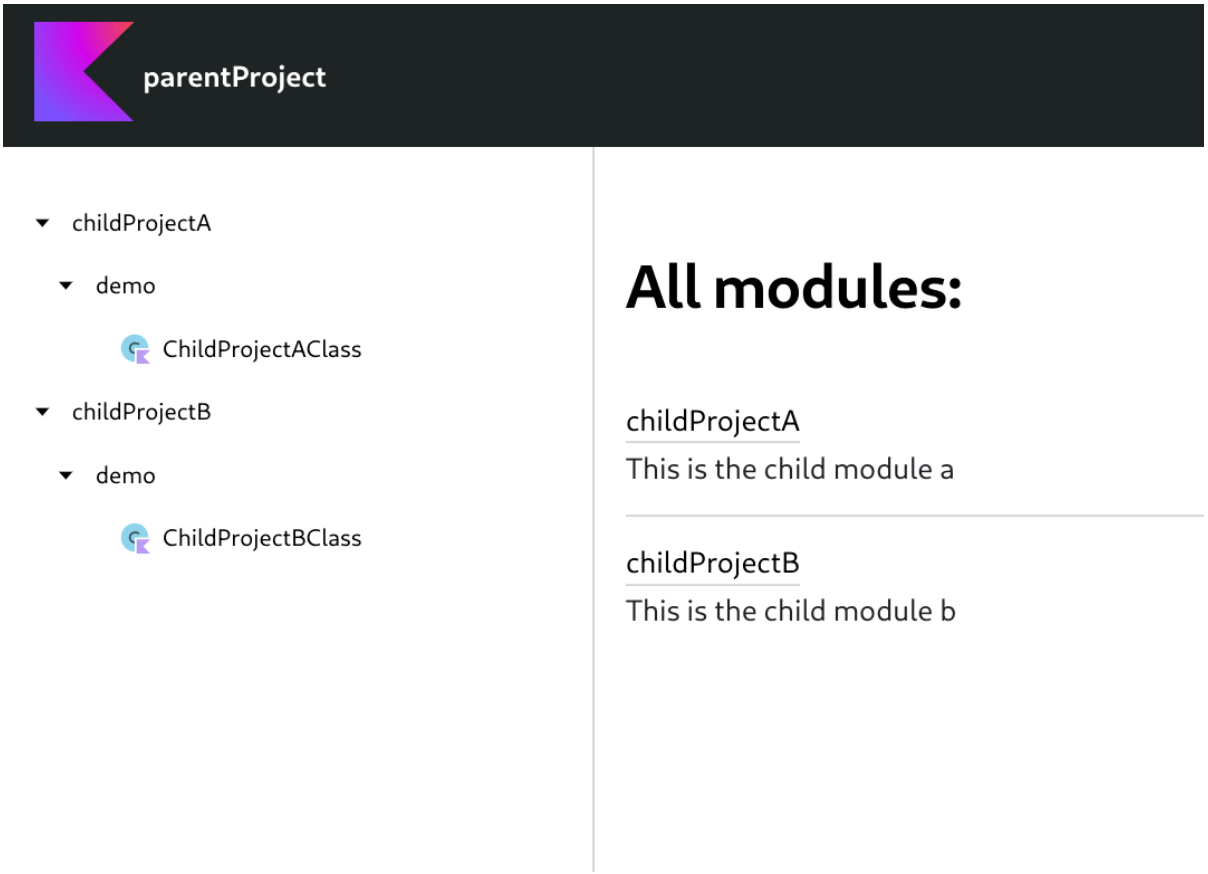
To aggregate documentation from multiple subprojects, add the `dependencies {}` block in the `build.gradle.kts` file of the root project:

```
dependencies {
    dokka(project(":childProjectA:"))
    dokka(project(":childProjectB:"))
}
```

Given a project with the following structure:

```
.
└── parentProject/
    ├── childProjectA/
    │   └── demo/
    │       └── ChildProjectAClass.kt
    └── childProjectB/
        └── demo/
            └── ChildProjectBClass.kt
```

The generated documentation is aggregated as follows:



Screenshot for output of dokkaHtmlMultiModule task

See our multi-project example for more details.

Directory of aggregated documentation

When DGP aggregates subprojects, each subproject has its own subdirectory within the aggregated documentation. DGP ensures each subproject has a unique directory by retaining the full project structure.

For example, given a project with an aggregation in `:turbo-lib` and a nested subproject `:turbo-lib:maths`, the generated documentation is placed under:

```
turbo-lib/build/dokka/html/turbo-lib/maths/
```

You can revert this behavior by manually specifying the subproject directory. Add the following configuration to the `build.gradle.kts` file of each subproject:

```
// /turbo-lib/maths/build.gradle.kts

plugins {
    id("org.jetbrains.dokka")
}

dokka {
    // Overrides the subproject directory
    modulePath.set("maths")
}
```

This configuration changes the generated documentation for the `:turbo-lib:maths` module to be generated into `turbo-lib/build/dokka/html/maths/`.

Build javadoc.jar

If you want to publish your library to a repository, you may need to provide a `javadoc.jar` file that contains API reference documentation of your library.

For example, if you want to publish to Maven Central, you must supply a `javadoc.jar` alongside your project. However, not all repositories have that rule.

The Gradle plugin for Dokka does not provide any way to do this out of the box, but it can be achieved with custom Gradle tasks. One for generating documentation in HTML format and another one for Javadoc format:

Gradle Kotlin DSL

```
// To generate documentation in HTML
val dokkaHtmlJar by tasks.registering(Jar::class) {
    description = "A HTML Documentation JAR containing Dokka HTML"
    from(tasks.dokkaGeneratePublicationHtml.flatMap { it.outputDirectory })
    archiveClassifier.set("html-doc")
}

// To generate documentation in Javadoc
val dokkaJavadocJar by tasks.registering(Jar::class) {
    description = "A Javadoc JAR containing Dokka Javadoc"
    from(tasks.dokkaGeneratePublicationJavadoc.flatMap { it.outputDirectory })
    archiveClassifier.set("javadoc")
}
```

Gradle Groovy DSL

```
// To generate documentation in HTML
tasks.register('dokkaHtmlJar', Jar) {
    description = 'A HTML Documentation JAR containing Dokka HTML'
    from(tasks.named('dokkaGeneratePublicationHtml').flatMap { it.outputDirectory })
    archiveClassifier.set('html-doc')
}

// To generate documentation in Javadoc
tasks.register('dokkaJavadocJar', Jar) {
    description = 'A Javadoc JAR containing Dokka Javadoc'
    from(tasks.named('dokkaGeneratePublicationJavadoc').flatMap { it.outputDirectory })
    archiveClassifier.set('javadoc')
}
```

If you publish your library to Maven Central, you can use services like javadoc.io to host your library's API documentation for free and without any setup. It takes documentation pages straight from the `javadoc.jar`. It works well with the HTML format as demonstrated in this example.

Configuration examples

Depending on the type of project that you have, the way you apply and configure Dokka differs slightly. However, configuration options themselves are the same, regardless of the type of your project.

For simple and flat projects with a single `build.gradle.kts` or `build.gradle` file found in the root of your project, see Single-project configuration.

For a more complex build with subprojects and multiple nested `build.gradle.kts` or `build.gradle` files, see Multi-project configuration.

Single-project configuration

Single-project builds usually have only one `build.gradle.kts` or `build.gradle` file in the root of the project. They can be either single-platform or multiplatform and typically have the following structure:

Single platform:

Kotlin

```
.
├── build.gradle.kts
└── src/
    └── main/
        └── kotlin/
            └── HelloWorld.kt
```

Multiplatform:

```
.
├── build.gradle.kts
└── src/
    ├── commonMain/
    │   └── kotlin/
    │       └── Common.kt
    ├── jvmMain/
    │   └── kotlin/
    │       └── JvmUtils.kt
    └── nativeMain/
        └── kotlin/
            └── NativeUtils.kt
```

Single platform:

Groovy

```
.
├── build.gradle
└── src/
    └── main/
        └── kotlin/
            └── HelloWorld.kt
```

Multiplatform:

```
.
├── build.gradle
└── src/
    ├── commonMain/
    │   └── kotlin/
    │       └── Common.kt
    ├── jvmMain/
    │   └── kotlin/
    │       └── JvmUtils.kt
    └── nativeMain/
        └── kotlin/
            └── NativeUtils.kt
```

Apply the Dokka Gradle plugin in your root `build.gradle.kts` file and configure it using the top-level `dokka {}` DSL:

Gradle Kotlin DSL

```
plugins {
    id("org.jetbrains.dokka") version "2.2.0"
}

dokka {
    dokkaPublications.html {
        moduleName.set("MyProject")
        outputDirectory.set(layout.buildDirectory.dir("documentation/html"))
        includes.from("README.md")
    }

    dokkaSourceSets.main {
        sourceLink {
            localDirectory.set(file("src/main/kotlin"))
            remoteUrl.set(URI("https://github.com/your-repo"))
            remoteLineSuffix.set("#L")
        }
    }
}
```

Inside `./build.gradle`:

Gradle Groovy DSL

```
plugins {
    id 'org.jetbrains.dokka' version '2.2.0'
}

dokka {
    dokkaPublications {
        html {
            moduleName.set("MyProject")
            outputDirectory.set(layout.buildDirectory.dir("documentation/html"))
            includes.from("README.md")
        }
    }

    dokkaSourceSets {
        named("main") {
            sourceLink {
                localDirectory.set(file("src/main/kotlin"))
                remoteUrl.set(new URI("https://github.com/your-repo"))
                remoteLineSuffix.set("#L")
            }
        }
    }
}
```

This configuration applies Dokka to your project, sets up the documentation output directory, and defines the main source set. You can extend it further by adding custom assets, visibility filters, or plugin configurations within the same `dokka {}` block. For more information, see Configuration options.

Multi-project configuration

Multi-project builds usually contain several nested `build.gradle.kts` files and have a structure similar to the following:

Kotlin

```
.
├── build.gradle.kts
├── settings.gradle.kts
├── subproject-A/
│   ├── build.gradle.kts
│   └── src/
│       └── main/
│           └── kotlin/
│               └── HelloFromA.kt
└── subproject-B/
    ├── build.gradle.kts
    └── src/
        └── main/
            └── kotlin/
                └── HelloFromB.kt
```

Groovy

```
.
├── build.gradle
├── settings.gradle
├── subproject-A/
│   ├── build.gradle
│   └── src/
│       └── main/
│           └── kotlin/
│               └── HelloFromA.kt
└── subproject-B/
    ├── build.gradle
    └── src/
        └── main/
            └── kotlin/
                └── HelloFromB.kt
```

Single and multi-project documentation share the same configuration model using the top-level `dokka {}` DSL.

There are two ways of configuring Dokka in multi-project builds:

- Shared configuration via a convention plugin **(preferred)**: defining a convention plugin and applying it to all subprojects. This centralizes your Dokka settings.
- Manual configuration: applying the Dokka plugin and repeating the same `dokka {}` block in each subproject. You don't need convention plugins.

After configuring your subprojects, you can aggregate the documentation from multiple subprojects into a single output. For more information, see Aggregate documentation output in multi-project-builds.

For a multi-project example, see the Dokka GitHub repository.

Shared configuration via a convention plugin

Follow the next steps to set up a convention plugin and apply it to your subprojects.

Set up the buildSrc directory

1. In your project root, create a `buildSrc` directory containing two files:
   - `settings.gradle.kts`
   - `build.gradle.kts`
2. In the `buildSrc/settings.gradle.kts` file, add the following snippet:

```
rootProject.name = "buildSrc"
```

3. In the `buildSrc/build.gradle.kts` file, add the following snippet:

```
plugins {
    `kotlin-dsl`
}

repositories {
    mavenCentral()
    gradlePluginPortal()
}

dependencies {
    implementation("org.jetbrains.dokka:dokka-gradle-plugin:2.2.0")
}
```

Set up the Dokka convention plugin

After setting up the `buildSrc` directory, set up the Dokka convention plugin:

1. Create a `buildSrc/src/main/kotlin/dokka-convention.gradle.kts` file to host the convention plugin.
2. In the `dokka-convention.gradle.kts` file, add the following snippet:

```
plugins {
    id("org.jetbrains.dokka")
}

dokka {
    // The shared configuration goes here
}
```

You need to add the shared Dokka configuration common to all subprojects within the `dokka {}` block. Also, you don't need to specify a Dokka version. The version is already set in the `buildSrc/build.gradle.kts` file.

Apply the convention plugin to your subprojects

Apply the Dokka convention plugin across your subprojects by adding it to each subproject's `build.gradle.kts` file:

```
plugins {
    id("dokka-convention")
}
```

Manual configuration

If your project doesn't use convention plugins, you can reuse the same Dokka configuration pattern by manually copying the same `dokka {} block` into each subproject:

1. Apply the Dokka plugin in every subproject's `build.gradle.kts` file:

```
plugins {
    id("org.jetbrains.dokka") version "2.2.0"
}
```

2. Declare the shared configuration in each subproject's `dokka {}` block. Because there's no convention plugin centralizing the configuration, you duplicate any configuration you want across subprojects. For more information, see configuration options.

Parent project configuration

In multi-project builds, you can configure settings that apply to the entire documentation in the root project. This can include defining the output format, output directory, documentation subproject name, aggregating documentation from all subprojects, and other configuration options:

```
plugins {
    id("org.jetbrains.dokka") version "2.2.0"
}

dokka {
    // Sets properties for the whole project
    dokkaPublications.html {
        moduleName.set("My Project")
        outputDirectory.set(layout.buildDirectory.dir("docs/html"))
        includes.from("README.md")
    }

    dokkaSourceSets.configureEach {
        documentedVisibilities.set(setOf(VisibilityModifier.Public)) // OR
documentedVisibilities(VisibilityModifier.Public)
    }
}

// Aggregates subproject documentation
dependencies {
    dokka(project(":childProjectA"))
    dokka(project(":childProjectB"))
}
```

Additionally, each subproject can have its own `dokka {}` block if it needs custom configuration. In the following example, the subproject applies the Dokka plugin, sets a custom subproject name, and includes additional documentation from its `README.md` file:

```
// subproject/build.gradle.kts
plugins {
    id("org.jetbrains.dokka")
}

dokka {
    dokkaPublications.html {
        moduleName.set("Child Project A")
        includes.from("README.md")
    }
}
```

Dokka Gradle configuration options

Dokka has many configuration options to customize your and your reader's experience.

Below are detailed descriptions for each configuration section and some examples. You can also find an example with all configuration options applied.

For more details on applying configuration blocks for single-project and multi-project builds, see Configuration examples.

General configuration

Here is an example of the general Dokka Gradle plugin configuration:

- Use the top-level `dokka {}` DSL configuration.
- In DGP, you declare Dokka publication configurations in the `dokkaPublications{}` block.
- The default publications are `html` and `javadoc`.
- The syntax of `build.gradle.kts` files differs from regular `.kt` files (such as those used for Kotlin custom plugins) because Gradle's Kotlin DSL uses type-safe accessors.

Gradle Kotlin DSL

```
plugins {
    id("org.jetbrains.dokka") version "2.2.0"
}

dokka {
    dokkaPublications.html {
        moduleName.set(project.name)
        moduleVersion.set(project.version.toString())
        // Standard output directory for HTML documentation
        outputDirectory.set(layout.buildDirectory.dir("dokka/html"))
        failOnWarning.set(false)
        suppressInheritedMembers.set(false)
        suppressObviousFunctions.set(true)
        offlineMode.set(false)
        includes.from("packages.md", "extra.md")

        // Output directory for additional files
        // Use this block instead of the standard when you
        // want to change the output directory and include extra files
        outputDirectory.set(rootDir.resolve("docs/api/0.x"))

        // Use fileTree to add multiple files
        includes.from(
            fileTree("docs") {
                include("**/*.md")
            }
        )
    }
}
```

For more information about working with files, see the Gradle docs.

Kotlin custom plugin

```kotlin
// CustomPlugin.kt

import org.gradle.api.Plugin
import org.gradle.api.Project
import org.jetbrains.dokka.gradle.DokkaExtension

abstract class CustomPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        project.plugins.apply("org.jetbrains.dokka")

        project.extensions.configure(DokkaExtension::class.java) { dokka ->

            dokka.moduleName.set(project.name)
            dokka.moduleVersion.set(project.version.toString())

            dokka.dokkaPublications.named("html") { publication ->
                // Standard output directory for HTML documentation
                publication.outputDirectory.set(project.layout.buildDirectory.dir("dokka/html"))
                publication.failOnWarning.set(true)
                publication.suppressInheritedMembers.set(true)
                publication.offlineMode.set(false)
                publication.suppressObviousFunctions.set(true)
                publication.includes.from("packages.md", "extra.md")

                // Output directory for additional files
                // Use this instead of the standard block when you
                // want to change the output directory and include extra files
                html.outputDirectory.set(project.rootDir.resolve("docs/api/0.x"))
            }
        }
    }
}
```

Gradle Groovy DSL

```
plugins {
    id 'org.jetbrains.dokka' version '2.2.0'
}

dokka {
    dokkaPublications {
        html {
            // Sets general module information
            moduleName.set(project.name)
            moduleVersion.set(project.version.toString())

            // Standard output directory for HTML documentation
            outputDirectory.set(layout.buildDirectory.dir("dokka/html"))

            // Core Dokka options
            failOnWarning.set(false)
            suppressInheritedMembers.set(false)
            suppressObviousFunctions.set(true)
            offlineMode.set(false)
            includes.from(files("packages.md", "extra.md"))

            // Output directory for additional files
            // Use this block instead of the standard when you want to
            // change the output directory and include extra files
            outputDirectory.set(file("$rootDir/docs/api/0.x"))
        }
    }
}
```

moduleName

The display name for the project's documentation. It appears in the table of contents, navigation, headers, and log messages. In multi-project builds, each subproject's `moduleName` is used as its section title in aggregated documentation.

Default: Gradle project name

moduleVersion

The subproject version displayed in the generated documentation. In single-project builds, it is used as the project version. In multi-project builds, each subproject's `moduleVersion` is used when aggregating documentation.

Default: Gradle project version

outputDirectory

The directory where the generated documentation is stored.

This setting applies to all documentation formats (HTML, Javadoc, etc.) generated by the `dokkaGenerate` task.

Default: `build/dokka/html`

Output directory for additional files

You can specify the output directory and include additional files for both single and multi-project builds. For multi-project builds, set the output directory and include additional files in the configuration of the root project.

failOnWarning

Determines whether Dokka should fail the build when a warning occurs during documentation generation. The process waits until all errors and warnings have been emitted first.

This setting works well with `reportUndocumented`.

Default: `false`

suppressInheritedMembers
Whether to suppress inherited members that aren't explicitly overridden in a given class.

Note: This suppresses functions such as `equals`, `hashCode`, and `toString`, but does not suppress synthetic functions such as `dataClass.componentN` and `dataClass.copy`. Use `suppressObviousFunctions` for that.

Default: `false`

suppressObviousFunctions
Whether to suppress obvious functions.

A function is considered to be obvious if it is:

- Inherited from `kotlin.Any`, `Kotlin.Enum`, `java.lang.Object` or `java.lang.Enum`, such as `equals`, `hashCode`, `toString`.
- Synthetic (generated by the compiler) and does not have any documentation, such as `dataClass.componentN` or `dataClass.copy`.

Default: `true`

offlineMode
Whether to resolve remote files and links over your network.

This includes package-lists used for generating links to external documentation.. For example, this allows to make classes from the standard library clickable in your documentation.

Setting this to `true` can significantly speed up build times in certain cases, but can also worsen user experience. For example, by not resolving class and member links from your dependencies, including the standard library.

Note: You can cache fetched files locally and provide them to Dokka as local paths. See the externalDocumentationLinks section.

Default: `false`

includes
A list of Markdown files that contain subproject and package documentation. The Markdown files must match the required format.

The contents of the specified files are parsed and embedded into documentation as subproject and package descriptions.

See Dokka Gradle example for an example of what it looks like and how to use it.

Source set configuration

Dokka allows configuring some options for Kotlin source sets:

Gradle Kotlin DSL

```
import org.jetbrains.dokka.gradle.engine.parameters.VisibilityModifier

dokka {
    // ..
    // General configuration section
    // ..

    // Source sets configuration
    dokkaSourceSets {
        // Example: Configuration exclusive to the 'linux' source set
        named("linux") {
            dependentSourceSets{named("native")}
            sourceRoots.from(file("linux/src"))
        }

        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set(setOf(VisibilityModifier.Public)) // OR
documentedVisibilities(VisibilityModifier.Public)
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            sourceRoots.from(file("src"))
            classpath.from(file("libs/dependency.jar"))
            samples.from("samples/Basic.kt", "samples/Advanced.kt")

            sourceLink {
                // Source link section
            }
            perPackageOption {
                // Package options section
            }
            externalDocumentationLinks {
                // External documentation links section
            }
        }
    }
}
```

Gradle Groovy DSL

```
import org.jetbrains.dokka.gradle.engine.parameters.VisibilityModifier

dokka {
    // ..
    // General configuration section
    // ..

    dokkaSourceSets {
        // Example: Configuration exclusive to the 'linux' source set
        named("linux") {
            dependentSourceSets { named("native") }
            sourceRoots.from(file("linux/src"))
        }

        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set([VisibilityModifier.Public] as Set) // OR
documentedVisibilities(VisibilityModifier.Public)
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            sourceRoots.from(file("src"))
            classpath.from(file("libs/dependency.jar"))
            samples.from("samples/Basic.kt", "samples/Advanced.kt")

            sourceLink {
                // Source link section
            }
            perPackageOption {
                // Package options section
            }
            externalDocumentationLinks {
                // External documentation links section
            }
        }
    }
}
```

suppress
Whether this source set should be skipped when generating documentation.

Default: `false`

displayName
The display name used to refer to this source set.

The name is used both externally (for example, as the source set name visible to documentation readers) and internally (for example, for logging messages of `reportUndocumented`).

By default, the value is deduced from information provided by the Kotlin Gradle plugin.

documentedVisibilities
Defines which visibility modifiers Dokka should include in the generated documentation.

Use them if you want to document `protected`, `internal`, and `private` declarations, as well as if you want to exclude `public` declarations and only document internal API.

Additionally, you can use Dokka's `documentedVisibilities()` function to add documented visibilities.

This can be configured for each individual package.

Default: `VisibilityModifier.Public`

reportUndocumented

Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by `documentedVisibilities` and other filters.

This setting works well with `failOnWarning`.

This can be configured for each individual package.

Default: `false`

skipEmptyPackages

Whether to skip packages that contain no visible declarations after various filters have been applied.

For example, if `skipDeprecated` is set to `true` and your package contains only deprecated declarations, it is considered to be empty.

Default: `true`

skipDeprecated

Whether to document declarations annotated with `@Deprecated`.

This can be configured for each individual package.

Default: `false`

suppressGeneratedFiles

Whether to document generated files.

Generated files are expected to be present under the `{project}/{buildDir}/generated` directory.

If set to `true`, it effectively adds all files from that directory to the `suppressedFiles` option, so you can configure it manually.

Default: `true`

suppressAnnotatedWith

A set of annotation fully qualified names (FQNs) to suppress declarations annotated with.

Any declaration annotated with one of these annotations is excluded from the generated documentation.

jdkVersion

The JDK version to use when generating external documentation links for Java types.

For example, if you use `java.util.UUID` in some public declaration signature, and this option is set to `8`, Dokka generates an external documentation link to JDK 8 Javadocs for it.

Default: `8`

languageVersion

The Kotlin language version used for setting up analysis and @sample environment.

By default, the latest language version available to Dokka's embedded compiler is used.

apiVersion

The Kotlin API version used for setting up analysis and @sample environment.

By default, it is deduced from `languageVersion`.

sourceRoots

The source code roots to be analyzed and documented. Acceptable inputs are directories and individual `.kt` and `.java` files.

By default, source roots are deduced from information provided by the Kotlin Gradle plugin.

classpath

The classpath for analysis and interactive samples.

This is useful if some types that come from dependencies are not resolved or picked up automatically.

This option accepts both `.jar` and `.klib` files.

By default, the classpath is deduced from information provided by the Kotlin Gradle plugin.

samples

A list of directories or files that contain sample functions which are referenced via the @sample KDoc tag.

Source link configuration

Configure source links to help readers find the source for each declaration in a remote repository. Use the `dokkaSourceSets.main {}` block for this configuration.

The `sourceLinks {}` configuration block allows you to add a `source` link to each signature that leads to the `remoteUrl` with a specific line number. The line number is configurable by setting `remoteLineSuffix`.

For an example, see the documentation for the `count()` function in `kotlinx.coroutines`.

The syntax of `build.gradle.kts` files differs from regular `.kt` files (such as those used for custom Gradle plugins) because Gradle's Kotlin DSL uses type-safe accessors:

Gradle Kotlin DSL

```
// build.gradle.kts

dokka {
    dokkaSourceSets.main {
        sourceLink {
            localDirectory.set(file("src/main/kotlin"))
            remoteUrl("https://github.com/your-repo")
            remoteLineSuffix.set("#L")
        }
    }
}
```

Kotlin custom plugin

```kotlin
// CustomPlugin.kt

import org.gradle.api.Plugin
import org.gradle.api.Project
import org.jetbrains.dokka.gradle.DokkaExtension

abstract class CustomPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        project.plugins.apply("org.jetbrains.dokka")
        project.extensions.configure(DokkaExtension::class.java) { dokka ->
            dokka.dokkaSourceSets.named("main") { dss ->
                dss.includes.from("README.md")
                dss.sourceLink {
                    it.localDirectory.set(project.file("src/main/kotlin"))
                    it.remoteUrl("https://example.com/src")
                    it.remoteLineSuffix.set("#L")
                }
            }
        }
    }
}
```

Gradle Groovy DSL

```groovy
dokka {
    dokkaSourceSets {
        main {
            sourceLink {
                localDirectory.set(file("src/main/kotlin"))
                remoteUrl.set(new URI("https://github.com/your-repo"))
                remoteLineSuffix.set("#L")
            }
        }
    }
}
```

localDirectory

The path to the local source directory. The path must be relative to the root of the current project.

remoteUrl

The URL of the source code hosting service that can be accessed by documentation readers, like GitHub, GitLab, Bitbucket, or any hosting service that provides stable URLs for source files. This URL is used to generate source code links of declarations.

remoteLineSuffix

The suffix used to append the source code line number to the URL. This helps readers navigate not only to the file, but to the specific line number of the declaration.

The number itself is appended to the specified suffix. For example, if this option is set to `#L` and the line number is 10, the resulting URL suffix is `#L10`.

Suffixes used by popular services:

- GitHub: `#L`
- GitLab: `#L`

- Bitbucket: `#lines-`

Default: `#L`

Package options

The `perPackageOption` configuration block allows setting some options for specific packages matched by `matchingRegex`:

Gradle Kotlin DSL

```
import org.jetbrains.dokka.gradle.engine.parameters.VisibilityModifier

dokka {
    dokkaPublications.html {
        dokkaSourceSets.configureEach {
            perPackageOption {
                matchingRegex.set(".*api.*")
                suppress.set(false)
                skipDeprecated.set(false)
                reportUndocumented.set(false)
                documentedVisibilities.set(setOf(VisibilityModifier.Public)) // OR
documentedVisibilities(VisibilityModifier.Public)
            }
        }
    }
}
```

Gradle Groovy DSL

```
import org.jetbrains.dokka.gradle.engine.parameters.VisibilityModifier

dokka {
    dokkaPublications {
        html {
            dokkaSourceSets.configureEach {
                perPackageOption {
                    matchingRegex.set(".*api.*")
                    suppress.set(false)
                    skipDeprecated.set(false)
                    reportUndocumented.set(false)
                    documentedVisibilities.set([VisibilityModifier.Public] as Set)
                }
            }
        }
    }
}
```

matchingRegex

The regular expression that is used to match the package.

Default: `.*`

suppress

Whether the package should be skipped when generating documentation.

Default: `false`

skipDeprecated
Whether to document declarations annotated with `@Deprecated`.

This can be configured on the source set level.

Default: `false`

reportUndocumented
Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by `documentedVisibilities` and other filters.

This setting works well with `failOnWarning`.

This can be configured on the source set level.

Default: `false`

documentedVisibilities
Defines which visibility modifiers Dokka should include in the generated documentation.

Use them if you want to document `protected`, `internal`, and `private` declarations within this package, as well as if you want to exclude `public` declarations and only document internal API.

Additionally, you can use Dokka's `documentedVisibilities()` function to add documented visibilities.

This can be configured on the source set level.

Default: `VisibilityModifier.Public`

External documentation links configuration

The `externalDocumentationLinks {}` block allows the creation of links that lead to the externally hosted documentation of your dependencies.

For example, if you are using types from `kotlinx.serialization`, by default they are not clickable in your documentation, as if they are unresolved. However, since the API reference documentation for `kotlinx.serialization` is built by Dokka and is published on kotlinlang.org, you can configure external documentation links for it. This allows Dokka to generate links for types from the library, making them resolve successfully and clickable.

By default, external documentation links for Kotlin standard library, JDK, Android SDK, and AndroidX are configured.

Register external documentation links using the `register()` method to define each link. The `externalDocumentationLinks` API uses this method aligning with Gradle DSL conventions:

Gradle Kotlin DSL

```
dokka {
    dokkaSourceSets.configureEach {
        externalDocumentationLinks.register("example-docs") {
            url("https://example.com/docs/")
            packageListUrl("https://example.com/docs/package-list")
        }
    }
}
```

Gradle Groovy DSL

```
dokka {
    dokkaSourceSets.configureEach {
        externalDocumentationLinks.register("example-docs") {
            url.set(new URI("https://example.com/docs/"))
            packageListUrl.set(new URI("https://example.com/docs/package-list"))
        }
    }
}
```

url

The root URL of documentation to link to. It must contain a trailing slash.

Dokka does its best to automatically find `package-list` for the given URL, and link declarations together.

If the automatic resolution fails or if you want to use locally cached files instead, consider setting the `packageListUrl` option.

packageListUrl

The exact location of a `package-list`. This is an alternative to relying on Dokka automatically resolving it.

Package lists contain information about the documentation and the project itself, such as subproject and package names.

This can also be a locally cached file to avoid network calls.

Complete configuration

Below you can see all possible configuration options applied at the same time:

Gradle Kotlin DSL

```
import org.jetbrains.dokka.gradle.engine.parameters.VisibilityModifier

plugins {
    id("org.jetbrains.dokka") version "2.2.0"
}

dokka {
    dokkaPublications.html {
        moduleName.set(project.name)
        moduleVersion.set(project.version.toString())
        outputDirectory.set(layout.buildDirectory.dir("dokka/html"))
        failOnWarning.set(false)
        suppressInheritedMembers.set(false)
        suppressObviousFunctions.set(true)
        offlineMode.set(false)
        includes.from("packages.md", "extra.md")
   }

    dokkaSourceSets {
        // Example: Configuration exclusive to the 'linux' source set
        named("linux") {
            dependentSourceSets{named("native")}
            sourceRoots.from(file("linux/src"))
        }

        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set(setOf(VisibilityModifier.Public)) // OR
documentedVisibilities(VisibilityModifier.Public)
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            sourceRoots.from(file("src"))
            classpath.from(file("libs/dependency.jar"))
            samples.from("samples/Basic.kt", "samples/Advanced.kt")

            sourceLink {
                localDirectory.set(file("src/main/kotlin"))
                remoteUrl("https://example.com/src")
                remoteLineSuffix.set("#L")
            }

            externalDocumentationLinks {
                url = URL("https://example.com/docs/")
                packageListUrl = File("/path/to/package-list").toURI().toURL()
            }

            perPackageOption {
                matchingRegex.set(".*api.*")
                suppress.set(false)
                skipDeprecated.set(false)
                reportUndocumented.set(false)
                documentedVisibilities.set(
                    setOf(
                        VisibilityModifier.Public,
```

```
                        VisibilityModifier.Private,
                        VisibilityModifier.Protected,
                        VisibilityModifier.Internal,
                        VisibilityModifier.Package
                    )
                )
            }
        }
    }
}
```

Gradle Groovy DSL

```
import org.jetbrains.dokka.gradle.engine.parameters.VisibilityModifier

plugins {
    id 'org.jetbrains.dokka' version '2.2.0'
}

dokka {
    dokkaPublications {
        html {
            moduleName.set(project.name)
            moduleVersion.set(project.version.toString())
            outputDirectory.set(layout.buildDirectory.dir("dokka/html"))
            failOnWarning.set(false)
            suppressInheritedMembers.set(false)
            suppressObviousFunctions.set(true)
            offlineMode.set(false)
            includes.from("packages.md", "extra.md")
        }
    }

    dokkaSourceSets {
        // Example: Configuration exclusive to the 'linux' source set
        named("linux") {
            dependentSourceSets { named("native") }
            sourceRoots.from(file("linux/src"))
        }

        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set([VisibilityModifier.Public] as Set)
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            sourceRoots.from(file("src"))
            classpath.from(file("libs/dependency.jar"))
            samples.from("samples/Basic.kt", "samples/Advanced.kt")

            sourceLink {
                localDirectory.set(file("src/main/kotlin"))
                remoteUrl.set(new URI("https://example.com/src"))
                remoteLineSuffix.set("#L")
            }

            externalDocumentationLinks {
                url.set(new URI("https://example.com/docs/"))
                packageListUrl.set(new File("/path/to/package-list").toURI().toURL())
            }

            perPackageOption {
                matchingRegex.set(".*api.*")
                suppress.set(false)
                skipDeprecated.set(false)
                reportUndocumented.set(false)
                documentedVisibilities.set([
                        VisibilityModifier.Public,
```

```
                        VisibilityModifier.Private,
                        VisibilityModifier.Protected,
                        VisibilityModifier.Internal,
                        VisibilityModifier.Package
                ] as Set)
            }
        }
    }
}
```

Dokka Gradle troubleshooting

This page describes common issues you may encounter when generating documentation with Dokka in a Gradle build.

If your issue is not listed here, report any feedback or problems in our issue tracker or chat with the Dokka community in the official Kotlin Slack. Get a Slack invite here.

Memory issues

In large projects, Dokka can consume a significant amount of memory to generate documentation. This can exceed Gradle's memory limits, especially when processing large volumes of data.

When Dokka generation runs out of memory, the build fails, and Gradle can throw exceptions like `java.lang.OutOfMemoryError: Metaspace`.

Active efforts are underway to improve Dokka's performance, although some limitations stem from Gradle.

If you encounter memory issues, try these workarounds:

- Increasing heap space
- Running Dokka within the Gradle process

Increase heap space

One way to resolve memory issues is to increase the amount of Java heap memory for the Dokka generator process. In the `build.gradle.kts` file, adjust the following configuration option:

```
dokka {
        // Dokka generates a new process managed by Gradle
        dokkaGeneratorIsolation = ProcessIsolation {
            // Configures heap size
            maxHeapSize = "4g"
        }
    }
```

In this example, the maximum heap size is set to 4 GB (`"4g"`). Adjust and test the value to find the optimal setting for your build.

If you find that Dokka requires a considerably expanded heap size, for example, significantly higher than Gradle's own memory usage, create an issue on Dokka's GitHub repository.

> You have to apply this configuration to each subproject. It is recommended that you configure Dokka in a convention plugin applied to all subprojects.

Run Dokka within the Gradle process

When both the Gradle build and Dokka generation require a lot of memory, they may run as separate processes, consuming significant memory on a single machine.

To optimize memory usage, you can run Dokka within the same Gradle process instead of as a separate process. This allows you to configure the memory for Gradle once instead of allocating it separately for each process.

To run Dokka within the same Gradle process, adjust the following configuration option in the `build.gradle.kts` file:

```
dokka {
        // Runs Dokka in the current Gradle process
        dokkaGeneratorIsolation = ClassLoaderIsolation()
    }
```

As with increasing heap space, test this configuration to confirm it works well for your project.

For more details on configuring Gradle's JVM memory, see the Gradle documentation.

> Changing the Java options for Gradle launches a new Gradle daemon, which may stay alive for a long time. You can manually stop any other Gradle processes.
>
> Additionally, Gradle issues with the `ClassLoaderIsolation()` configuration may cause memory leaks.

Migrate to Dokka Gradle plugin v2

> This page is only relevant if you are using DGPv1 and want to migrate to DGPv2. Starting from Dokka 2.1.0, DGP v2 is enabled by default. If you are using Dokka 2.1.0 or later, you can skip this page and go directly to the Dokka Gradle docs.

The Dokka Gradle plugin (DGP) is a tool for generating comprehensive API documentation for Kotlin projects built with Gradle.

DGP seamlessly processes both Kotlin's KDoc comments and Java's Javadoc comments to extract information and create structured documentation in HTML or Javadoc format.

The Dokka Gradle plugin v2 mode is enabled by default and aligns with Gradle best practices:

- Adopts Gradle types, which leads to better performance.
- Uses an intuitive top-level DSL configuration instead of a low-level task-based setup, which simplifies the build scripts and their readability.
- Takes a more declarative approach to documentation aggregation, which makes multi-project documentation easier to manage.
- Uses a type-safe plugin configuration, which improves the reliability and maintainability of your build scripts.
- Fully supports Gradle configuration cache and build cache, which improves performance and simplifies build work.

Read this guide for further information on changes and migration from DGP v1 to v2 modes.

Before you start

Before starting the migration, complete the following steps.

Verify supported versions

Ensure that your project meets the minimum version requirements:

Tool	Version
Gradle	7.6 or higher
Android Gradle plugin	7.0 or higher
Kotlin Gradle plugin	1.9 or higher

Enable DGP v2

Update the Dokka version to 2.2.0 in the `plugins {}` block of your project's `build.gradle.kts` file:

```
plugins {
    kotlin("jvm") version "2.1.10"
    id("org.jetbrains.dokka") version "2.2.0"
}
```

Alternatively, you can use version catalog to enable the Dokka Gradle plugin v2.

By default, DGP v2 generates documentation in HTML format. To generate Javadoc or both HTML and Javadoc formats, add the appropriate plugins. For more information about the plugins, see Select documentation output format.

Enable migration helpers

In the project's `gradle.properties` file, set the following Gradle property to activate DGP v2 with helpers:

```
org.jetbrains.dokka.experimental.gradle.pluginMode=V2EnabledWithHelpers
```

If your project doesn't have a `gradle.properties` file, create one in the root directory of your project.

This property activates the DGP v2 plugin with migration helpers. These helpers prevent compilation errors when build scripts reference tasks from DGP v1 that are no longer available in DGP v2.

Migration helpers do not actively assist with the migration. They only keep your build script from breaking while you transition to the new API.

After completing the migration, disable the migration helpers.

Sync your project with Gradle

After enabling DGP v2 and migration helpers, sync your project with Gradle to ensure DGP v2 is properly applied:

- If you use IntelliJ IDEA, click the **Reload All Gradle Projects** button in the Gradle tool window.
- If you use Android Studio, select **File** | **Sync Project with Gradle Files**.

Migrate your project

After updating the Dokka Gradle plugin to v2, follow the migration steps applicable to your project.

Adjust configuration options

DGP v2 introduces some changes in the Gradle configuration options. In the `build.gradle.kts` file, adjust the configuration options according to your project setup.

Top-level DSL configuration in DGP v2

Replace the configuration syntax of DGP v1 with the top-level `dokka {}` DSL configuration of DGP v2:

Configuration in DGP v1:

```
tasks.withType<DokkaTask>().configureEach {
    suppressInheritedMembers.set(true)
    failOnWarning.set(true)
    dokkaSourceSets {
        named("main") {
            moduleName.set("Project Name")
            includes.from("README.md")
            sourceLink {
                localDirectory.set(file("src/main/kotlin"))
                remoteUrl.set(URL("https://example.com/src"))
                remoteLineSuffix.set("#L")
            }
        }
    }
}

tasks.dokkaHtml {
    pluginConfiguration<DokkaBase, DokkaBaseConfiguration> {
        customStyleSheets.set(listOf("styles.css"))
        customAssets.set(listOf("logo.png"))
        footerMessage.set("(c) Your Company")
    }
}
```

Configuration in DGP v2:

The syntax of `build.gradle.kts` files differs from regular `.kt` files (such as those used for custom Gradle plugins) because Gradle's Kotlin DSL uses type-safe accessors.

Gradle Kotlin DSL

```
// build.gradle.kts

dokka {
    moduleName.set("Project Name")
    dokkaPublications.html {
        suppressInheritedMembers.set(true)
        failOnWarning.set(true)
    }
    dokkaSourceSets.main {
        includes.from("README.md")
        sourceLink {
            localDirectory.set(file("src/main/kotlin"))
            remoteUrl("https://example.com/src")
            remoteLineSuffix.set("#L")
        }
    }
    pluginsConfiguration.html {
        customStyleSheets.from("styles.css")
        customAssets.from("logo.png")
        footerMessage.set("(c) Your Company")
    }
}
```

Kotlin custom plugin

```
// CustomPlugin.kt

import org.gradle.api.Plugin
import org.gradle.api.Project
import org.jetbrains.dokka.gradle.DokkaExtension
import org.jetbrains.dokka.gradle.engine.plugins.DokkaHtmlPluginParameters

abstract class CustomPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        project.plugins.apply("org.jetbrains.dokka")

        project.extensions.configure(DokkaExtension::class.java) { dokka ->

            dokka.dokkaPublications.named("html") { publication ->
                publication.suppressInheritedMembers.set(true)
                publication.failOnWarning.set(true)
            }

            dokka.dokkaSourceSets.named("main") { dss ->
                dss.includes.from("README.md")
                dss.sourceLink {
                    it.localDirectory.set(project.file("src/main/kotlin"))
                    it.remoteUrl("https://example.com/src")
                    it.remoteLineSuffix.set("#L")
                }
            }

            dokka.pluginsConfiguration.named("html", DokkaHtmlPluginParameters::class.java) { html ->
                html.customStyleSheets.from("styles.css")
                html.customAssets.from("logo.png")
                html.footerMessage.set("(c) Your Company")
            }
        }
    }
}
```

Visibility settings

Set the `documentedVisibilities` property from `Visibility.PUBLIC` to `VisibilityModifier.Public`.

Configuration in DGP v1:

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility

// ...
documentedVisibilities.set(
    setOf(Visibility.PUBLIC)
)
```

Configuration in DGP v2:

```
import org.jetbrains.dokka.gradle.engine.parameters.VisibilityModifier

// ...
documentedVisibilities.set(
    setOf(VisibilityModifier.Public)
)

// OR

documentedVisibilities(VisibilityModifier.Public)
```

Additionally, use DGP v2's utility function to add documented visibilities:

```
fun documentedVisibilities(vararg visibilities: VisibilityModifier): Unit =
    documentedVisibilities.set(visibilities.asList())
```

Source links

Configure source links to allow navigation from the generated documentation to the corresponding source code in a remote repository. Use the `dokkaSourceSets.main{}` block for this configuration.

Configuration in DGP v1:

```
tasks.withType<DokkaTask>().configureEach {
    dokkaSourceSets {
        named("main") {
            sourceLink {
                localDirectory.set(file("src/main/kotlin"))
                remoteUrl.set(URL("https://github.com/your-repo"))
                remoteLineSuffix.set("#L")
            }
        }
    }
}
```

Configuration in DGP v2:

The syntax of `build.gradle.kts` files differs from regular `.kt` files (such as those used for custom Gradle plugins) because Gradle's Kotlin DSL uses type-safe accessors.

Gradle Kotlin DSL

```
// build.gradle.kts

dokka {
    dokkaSourceSets.main {
        sourceLink {
            localDirectory.set(file("src/main/kotlin"))
            remoteUrl("https://github.com/your-repo")
            remoteLineSuffix.set("#L")
        }
    }
}
```

Kotlin custom plugin

```kotlin
// CustomPlugin.kt

import org.gradle.api.Plugin
import org.gradle.api.Project
import org.jetbrains.dokka.gradle.DokkaExtension

abstract class CustomPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        project.plugins.apply("org.jetbrains.dokka")
        project.extensions.configure(DokkaExtension::class.java) { dokka ->
            dokka.dokkaSourceSets.named("main") { dss ->
                dss.includes.from("README.md")
                dss.sourceLink {
                    it.localDirectory.set(project.file("src/main/kotlin"))
                    it.remoteUrl("https://example.com/src")
                    it.remoteLineSuffix.set("#L")
                }
            }
        }
    }
}
```

Since the source link configuration has changed, use the `URI` class instead of `URL` to specify the remote URL.

Configuration in DGP v1:

```kotlin
remoteUrl.set(URL("https://github.com/your-repo"))
```

Configuration in DGP v2:

```kotlin
remoteUrl.set(URI("https://github.com/your-repo"))

// or

remoteUrl("https://github.com/your-repo")
```

Additionally, DGP v2 has two utility functions for setting the URL:

```kotlin
fun remoteUrl(@Language("http-url-reference") value: String): Unit =
    remoteUrl.set(URI(value))

// and

fun remoteUrl(value: Provider<String>): Unit =
    remoteUrl.set(value.map(::URI))
```

External documentation links

Register external documentation links using the `register()` method to define each link. The `externalDocumentationLinks` API uses this method aligning with Gradle DSL conventions.

Configuration in DGP v1:

```
tasks.dokkaHtml {
    dokkaSourceSets {
        configureEach {
            externalDocumentationLink {
                url = URL("https://example.com/docs/")
                packageListUrl = File("/path/to/package-list").toURI().toURL()
            }
        }
    }
}
```

Configuration in DGP v2:

```
dokka {
    dokkaSourceSets.configureEach {
        externalDocumentationLinks.register("example-docs") {
            url("https://example.com/docs/")
            packageListUrl("https://example.com/docs/package-list")
        }
    }
}
```

Custom assets

Use the `customAssets` property with collections of files (`FileCollection`) instead of lists (`var List<File>`).

Configuration in DGP v1:

```
customAssets = listOf(file("example.png"), file("example2.png"))
```

Configuration in DGP v2:

```
customAssets.from("example.png", "example2.png")
```

Output directory

Use the `dokka {}` block to specify the output directory for the generated Dokka documentation.

Configuration in DGP v1:

```
tasks.dokkaHtml {
    outputDirectory.set(layout.buildDirectory.dir("dokkaDir"))
}
```

Configuration in DGP v2:

```
dokka {
    dokkaPublications.html {
        outputDirectory.set(layout.buildDirectory.dir("dokkaDir"))
    }
}
```

Output directory for additional files

Specify the output directory and include additional files for both single-module and multi-module projects inside the `dokka {}` block.

In DGP v2, the configuration for single-module and multi-module projects is unified. Instead of configuring `dokkaHtml` and `dokkaHtmlMultiModule` tasks separately, specify the settings in `dokkaPublications.html {}` within the `dokka {}` block.

For multi-module projects, set the output directory and include additional files (such as `README.md`) in the configuration of the root project.

Configuration in DGP v1:

```
tasks.dokkaHtmlMultiModule {
    outputDirectory.set(rootDir.resolve("docs/api/0.x"))
    includes.from(project.layout.projectDirectory.file("README.md"))
}
```

Configuration in DGP v2:

The syntax of `build.gradle.kts` files differs from regular `.kt` files (such as those used for custom Gradle plugins) because Gradle's Kotlin DSL uses type-safe accessors.

Gradle Kotlin DSL

```
// build.gradle.kts

dokka {
    dokkaPublications.html {
        outputDirectory.set(rootDir.resolve("docs/api/0.x"))
        includes.from(project.layout.projectDirectory.file("README.md"))
    }
}
```

Kotlin custom plugin

```
// CustomPlugin.kt

import org.gradle.api.Plugin
import org.gradle.api.Project
import org.jetbrains.dokka.gradle.DokkaExtension

abstract class CustomPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        project.plugins.apply("org.jetbrains.dokka")
        project.extensions.configure(DokkaExtension::class.java) { dokka ->
            dokka.dokkaPublications.named("html") { html ->
                html.outputDirectory.set(project.rootDir.resolve("docs/api/0.x"))
                html.includes.from(project.layout.projectDirectory.file("README.md"))
            }
        }
    }
}
```

Configure Dokka plugins

Configuring built-in Dokka plugins with JSON is deprecated in favor of a type-safe DSL. This change improves compatibility with Gradle's incremental build system and improves task input tracking.

Configuration in DGP v1:

In DGP v1, Dokka plugins were configured manually using JSON. This approach caused issues with registering task inputs for Gradle up-to-date checks.

Here is an example of the deprecated JSON-based configuration for the Dokka Versioning plugin:

```
tasks.dokkaHtmlMultiModule {
    pluginsMapConfiguration.set(
        mapOf(
            "org.jetbrains.dokka.versioning.VersioningPlugin" to """
                { "version": "1.2", "olderVersionsDir": "$projectDir/dokka-docs" }
                """.trimIndent()
        )
    )
}
```

Configuration in DGP v2:

In DGP v2, Dokka plugins are configured using type-safe DSL. To configure Dokka plugins in a type-safe way, use the `pluginsConfiguration{}` block:

```
dokka {
    pluginsConfiguration {
        versioning {
            version.set("1.2")
            olderVersionsDir.set(projectDir.resolve("dokka-docs"))
        }
    }
}
```

For an example of the DGP v2 configuration, see the Dokka's versioning plugin.

DGP v2 allows you to extend its functionality by configuring custom plugins. Custom plugins enable additional processing or modifications to the documentation generation process.

Share Dokka configuration across subprojects

DPG v2 moves away from using `subprojects {}` or `allprojects {}` to share configuration across subprojects. In future Gradle versions, using these approaches will lead to errors.

Follow the steps below to properly share Dokka configuration in multi-module projects with existing convention plugins or without convention plugins.

After sharing the Dokka configuration, you can aggregate the documentation from multiple subprojects into a single output. For more information, see Update documentation aggregation in multi-module projects.


For a multi-module project example, see the Dokka GitHub repository.

Multi-module projects without convention plugins

If your project doesn't use convention plugins, you can still share Dokka configurations by directly configuring each subproject. This involves manually setting up the shared configuration in each subproject's `build.gradle.kts` file. While this approach is less centralized, it avoids the need for additional setups like convention plugins.

Otherwise, if your project uses convention plugins, you can also share the Dokka configuration in multi-module projects by creating a convention plugin in the `buildSrc` directory, and then applying the plugin to your subprojects.

Set up the buildSrc directory

1. In your project root, create a `buildSrc` directory containing two files:
   - `settings.gradle.kts`
   - `build.gradle.kts`
2. In the `buildSrc/settings.gradle.kts` file, add the following snippet:

```
rootProject.name = "buildSrc"
```

3. In the `buildSrc/build.gradle.kts` file, add the following snippet:

```
plugins {
    `kotlin-dsl`
}

repositories {
    mavenCentral()
    gradlePluginPortal()
}

dependencies {
    implementation("org.jetbrains.dokka:dokka-gradle-plugin:2.2.0")
}
```

Set up the Dokka convention plugin

After setting up the `buildSrc` directory:

1. Create a `buildSrc/src/main/kotlin/dokka-convention.gradle.kts` file to host the convention plugin.
2. In the `dokka-convention.gradle.kts` file, add the following snippet:

```
plugins {
    id("org.jetbrains.dokka")
}

dokka {
    // The shared configuration goes here
}
```

You need to add the shared Dokka configuration common to all subprojects within the `dokka {}` block. Also, you don't need to specify a Dokka version. The version is already set in the `buildSrc/build.gradle.kts` file.

Apply the convention plugin to your subprojects

Apply the Dokka convention plugin across your subprojects by adding it to each subproject's `build.gradle.kts` file:

```
plugins {
    id("dokka-convention")
}
```

Multi-module projects with convention plugins

If you already have convention plugins, create a dedicated Dokka convention plugin following Gradle's documentation.

Then, follow the steps to set up the Dokka convention plugin and apply it across your subprojects.

Update documentation aggregation in multi-module projects

Dokka can aggregate the documentation from multiple subprojects into a single output or publication.

As explained, apply the Dokka plugin to all documentable subprojects before aggregating the documentation.

Aggregation in DGP v2 uses the `dependencies {}` block instead of tasks and can be added in any `build.gradle.kts` file.

In DGP v1, aggregation was implicitly created in the root project. To replicate this behavior in DGP v2, add the `dependencies {}` block in the `build.gradle.kts` file of the root project.

Aggregation in DGP v1:

```
tasks.dokkaHtmlMultiModule {
        // ...
    }
```

Aggregation in DGP v2:

```
dependencies {
    dokka(project(":some-subproject:"))
    dokka(project(":another-subproject:"))
}
```

Change directory of aggregated documentation

When DGP aggregates subprojects, each subproject has its own subdirectory within the aggregated documentation.

In DGP v2, the aggregation mechanism has been updated to better align with Gradle conventions. DGP v2 now preserves the full subproject directory to prevent conflicts when aggregating documentation in any location.

Aggregation directory in DGP v1:

In DGP v1, aggregated documentation was placed in a collapsed directory structure. For example, given a project with an aggregation in `:turbo-lib` and a nested subproject `:turbo-lib:maths`, the generated documentation was placed under:

```
turbo-lib/build/dokka/html/maths/
```

Aggregation directory in DGP v2:

DGP v2 ensures each subproject has a unique directory by retaining the full project structure. The same aggregated documentation now follows this structure:

```
turbo-lib/build/dokka/html/turbo-lib/maths/
```

This change prevents subprojects with the same name from clashing. However, since the directory structure has changed, external links may become outdated, potentially causing `404` errors.

Revert to the DGP v1 directory behavior

If your project depends on the directory structure used in DGP v1, you can revert this behavior by manually specifying the subproject directory. Add the following configuration to the `build.gradle.kts` file of each subproject:

```
// /turbo-lib/maths/build.gradle.kts

plugins {
    id("org.jetbrains.dokka")
}

dokka {
    // Overrides the subproject directory to match the V1 structure
    modulePath.set("maths")
}
```

Generate documentation with the updated task

DGP v2 has renamed the Gradle tasks that generate the API documentation.

Task in DGP v1:

```
./gradlew dokkaHtml

// or

./gradlew dokkaHtmlMultiModule
```

Task in DGP v2:

```
./gradlew :dokkaGenerate
```

The `dokkaGenerate` task generates the API documentation in the `build/dokka/` directory.

In the DGP v2 version, the `dokkaGenerate` task name works for both single and multi-module projects. You can use different tasks to generate output in HTML, Javadoc or both HTML and Javadoc. For more information, see Select documentation output format.

Select documentation output format

> The Javadoc output format is in Alpha. You may find bugs and experience migration issues when using it. Successful integration with tools that accept Javadoc as input is not guaranteed. Use it at your own risk.

The default output format for DGP v2 is HTML. However, you can choose to generate the API documentation in HTML, Javadoc, or both formats at the same time:

1. Place the corresponding plugin `id` in the `plugins {}` block of your project's `build.gradle.kts` file:

```
plugins {
    // Generates HTML documentation
    id("org.jetbrains.dokka") version "2.2.0"

    // Generates Javadoc documentation
    id("org.jetbrains.dokka-javadoc") version "2.2.0"

    // Keeping both plugin IDs generates both formats
}
```

2. Run the corresponding Gradle task.

Here is a list of the plugin `id` and Gradle task that correspond to each format:

	HTML	Javadoc	Both
Plugin `id`	`id("org.jetbrains.dokka")`	`id("org.jetbrains.dokka-javadoc")`	Use both HTML and Javadoc plugins
Gradle task	`./gradlew :dokkaGeneratePublicationHtml`	`./gradlew :dokkaGeneratePublicationJavadoc`	`./gradlew :dokkaGenerate`

> The `dokkaGenerate` task generates documentation in all available formats based on the applied plugins. If both the HTML and Javadoc plugins are applied, you can choose to generate only HTML by running the `dokkaGeneratePublicationHtml` task, or only Javadoc by running the `dokkaGeneratePublicationJavadoc` task.

If you're using IntelliJ IDEA, you may see the `dokkaGenerateHtml` Gradle task. This task is simply an alias of `dokkaGeneratePublicationHtml`. Both tasks perform exactly the same operation.

Address deprecations and removals

- **Output format support:** DGP v2 only supports HTML and Javadoc output. Experimental formats like Markdown and Jekyll are no longer supported.
- **Collector task:** `DokkaCollectorTask` has been removed. Now, you need to generate the documentation separately for each subproject and then aggregate the documentation if necessary.

Finalize your migration

After you've migrated your project, perform these steps to wrap up and improve performance.

Set the opt-in flag

After successful migration, set the following opt-in flag without helpers in the project's `gradle.properties` file:

```
org.jetbrains.dokka.experimental.gradle.pluginMode=V2Enabled
```

If you removed references to Gradle tasks from DGP v1 that are no longer available in DGP v2, you shouldn't see compilation errors related to it.

Enable build cache and configuration cache

DGP v2 now supports Gradle build cache and configuration cache, improving build performance.

- To enable build cache, follow instructions in the Gradle build cache documentation.
- To enable configuration cache, follow instructions in the Gradle configuration cache documentation.

What's next

- Explore more DGP v2 project examples.
- Get started with Dokka.
- Learn more about Dokka plugins.

Maven

To generate documentation for a Maven-based project, you can use the Maven plugin for Dokka.

> Compared to the Gradle plugin for Dokka, the Maven plugin has only basic features and does not provide support for multi-module builds.

You can play around with Dokka and see how it can be configured for a Maven project by visiting our Maven example project.

Apply Dokka

To apply Dokka, you need to add `dokka-maven-plugin` to the `plugins` section of your POM file:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.dokka</groupId>
            <artifactId>dokka-maven-plugin</artifactId>
            <version>2.2.0</version>
            <executions>
                <execution>
                    <phase>pre-site</phase>
                    <goals>
                        <goal>dokka</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Generate documentation

The following goals are provided by the Maven plugin:

Goal	Description
`dokka:dokka`	Generates documentation with Dokka plugins applied. HTML format by default.

Experimental

Goal	Description
`dokka:javadoc`	Generates documentation in Javadoc format.
`dokka:javadocJar`	Generates a `javadoc.jar` file that contains documentation in Javadoc format.

Other output formats

By default, the Maven plugin for Dokka builds documentation in HTML output format.

All other output formats are implemented as Dokka plugins. In order to generate documentation in the desired format, you have to add it as a Dokka plugin to the configuration.

For example, to use the experimental GFM format, you have to add `gfm-plugin` artifact:

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>gfm-plugin</artifactId>
                <version>2.2.0</version>
            </plugin>
        </dokkaPlugins>
    </configuration>
</plugin>
```

With this configuration, running the `dokka:dokka` goal produces documentation in GFM format.

To learn more about Dokka plugins, see Dokka plugins.

Build javadoc.jar

If you want to publish your library to a repository, you may need to provide a `javadoc.jar` file that contains API reference documentation of your library.

For example, if you want to publish to Maven Central, you must supply a `javadoc.jar` alongside your project. However, not all repositories have that rule.

Unlike the Gradle plugin for Dokka, the Maven plugin comes with a ready-to-use `dokka:javadocJar` goal. By default, it generates documentation in Javadoc output format in the `target` folder.

If you are not satisfied with the built-in goal or want to customize the output (for example, you want to generate documentation in HTML format instead of Javadoc), similar behavior can be achieved by adding the Maven JAR plugin with the following configuration:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>3.3.0</version>
    <executions>
        <execution>
            <goals>
                <goal>test-jar</goal>
            </goals>
        </execution>
        <execution>
            <id>dokka-jar</id>
            <phase>package</phase>
            <goals>
                <goal>jar</goal>
            </goals>
            <configuration>
                <classifier>dokka</classifier>
                <classesDirectory>${project.build.directory}/dokka</classesDirectory>
                <skipIfEmpty>true</skipIfEmpty>
            </configuration>
        </execution>
    </executions>
</plugin>
```

The documentation and the `.jar` archive for it are then generated by running `dokka:dokka` and `jar:jar@dokka-jar` goals:

```
mvn dokka:dokka jar:jar@dokka-jar
```

If you publish your library to Maven Central, you can use services like javadoc.io to host your library's API documentation for free and without any setup. It takes documentation pages straight from the `javadoc.jar`. It works well with the HTML format as demonstrated in this example.

Configuration example

Maven's plugin configuration block can be used to configure Dokka.

Here is an example of a basic configuration that only changes the output location of your documentation:

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <outputDir>${project.basedir}/target/documentation/dokka</outputDir>
    </configuration>
</plugin>
```

Configuration options

Dokka has many configuration options to tailor your and your reader's experience.

Below are some examples and detailed descriptions for each configuration section. You can also find an example with all configuration options applied at the bottom of the page.

General configuration

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!--   ...   -->
    <configuration>
        <skip>false</skip>
        <moduleName>${project.artifactId}</moduleName>
        <outputDir>${project.basedir}/target/documentation</outputDir>
        <failOnWarning>false</failOnWarning>
        <suppressObviousFunctions>true</suppressObviousFunctions>
        <suppressInheritedMembers>false</suppressInheritedMembers>
        <offlineMode>false</offlineMode>
        <sourceDirectories>
            <dir>${project.basedir}/src</dir>
        </sourceDirectories>
        <documentedVisibilities>
            <visibility>PUBLIC</visibility>
            <visibility>PROTECTED</visibility>
        </documentedVisibilities>
        <reportUndocumented>false</reportUndocumented>
        <skipDeprecated>false</skipDeprecated>
        <skipEmptyPackages>true</skipEmptyPackages>
        <suppressedFiles>
            <file>/path/to/dir</file>
            <file>/path/to/file</file>
        </suppressedFiles>
        <suppressAnnotatedWith>
            <annotation>com.example.SuppressMe</annotation>
        </suppressAnnotatedWith>
        <jdkVersion>8</jdkVersion>
        <languageVersion>1.7</languageVersion>
        <apiVersion>1.7</apiVersion>
        <noStdlibLink>false</noStdlibLink>
        <noJdkLink>false</noJdkLink>
        <includes>
            <include>packages.md</include>
            <include>extra.md</include>
        </includes>
        <classpath>${project.compileClasspathElements}</classpath>
        <samples>
            <dir>${project.basedir}/samples</dir>
        </samples>
        <sourceLinks>
            <!-- Separate section -->
        </sourceLinks>
        <externalDocumentationLinks>
            <!-- Separate section -->
        </externalDocumentationLinks>
        <perPackageOptions>
            <!-- Separate section -->
        </perPackageOptions>
    </configuration>
</plugin>
```

skip
Whether to skip documentation generation.

Default: `false`

moduleName
The display name used to refer to the project/module. It's used for the table of contents, navigation, logging, etc.

Default: `{project.artifactId}`

outputDir
The directory to where documentation is generated, regardless of format.

Default: `{project.basedir}/target/dokka`

failOnWarning
Whether to fail documentation generation if Dokka has emitted a warning or an error. The process waits until all errors and warnings have been emitted first.

This setting works well with `reportUndocumented`.

Default: `false`

suppressObviousFunctions
Whether to suppress obvious functions.

A function is considered to be obvious if it is:

- Inherited from `kotlin.Any`, `Kotlin.Enum`, `java.lang.Object` or `java.lang.Enum`, such as `equals`, `hashCode`, `toString`.
- Synthetic (generated by the compiler) and does not have any documentation, such as `dataClass.componentN` or `dataClass.copy`.

Default: `true`

suppressInheritedMembers
Whether to suppress inherited members that aren't explicitly overridden in a given class.

Note: This can suppress functions such as `equals` / `hashCode` / `toString`, but cannot suppress synthetic functions such as `dataClass.componentN` and `dataClass.copy`. Use `suppressObviousFunctions` for that.

Default: `false`

offlineMode
Whether to resolve remote files/links over your network.

This includes package-lists used for generating external documentation links. For example, to make classes from the standard library clickable.

Setting this to `true` can significantly speed up build times in certain cases, but can also worsen documentation quality and user experience. For example, by not resolving class/member links from your dependencies, including the standard library.

Note: You can cache fetched files locally and provide them to Dokka as local paths. See `externalDocumentationLinks` section.

Default: `false`

sourceDirectories
The source code roots to be analyzed and documented. Acceptable inputs are directories and individual `.kt` / `.java` files.

Default: `{project.compileSourceRoots}`

documentedVisibilities
The set of visibility modifiers that should be documented.

This can be used if you want to document `protected` / `internal` / `private` declarations, as well as if you want to exclude `public` declarations and only document internal API.

Can be configured on per-package basis.

Default: `PUBLIC`

reportUndocumented
Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by `documentedVisibilities` and other filters.

This setting works well with `failOnWarning`.

This can be overridden at package level.

Default: `false`

skipDeprecated
Whether to document declarations annotated with `@Deprecated`.

This can be overridden at package level.

Default: `false`

skipEmptyPackages
Whether to skip packages that contain no visible declarations after various filters have been applied.

For example, if `skipDeprecated` is set to `true` and your package contains only deprecated declarations, it is considered to be empty.

Default: `true`

suppressedFiles
The directories or individual files that should be suppressed, meaning that declarations from them are not documented.

suppressAnnotatedWith
A list of annotation fully qualified names (FQNs) to suppress declarations annotated with.

Any declaration annotated with one of these annotations is excluded from the generated documentation.

jdkVersion
The JDK version to use when generating external documentation links for Java types.

For example, if you use `java.util.UUID` in some public declaration signature, and this option is set to `8`, Dokka generates an external documentation link to JDK 8 Javadocs for it.

Default: JDK 8

languageVersion
The Kotlin language version used for setting up analysis and @sample environment.

By default, the latest language version available to Dokka's embedded compiler is used.

apiVersion
The Kotlin API version used for setting up analysis and @sample environment.

By default, it is deduced from `languageVersion`.

noStdlibLink
Whether to generate external documentation links that lead to the API reference documentation of Kotlin's standard library.

Note: Links are generated when `noStdLibLink` is set to `false`.

Default: `false`

noJdkLink

Whether to generate external documentation links to JDK's Javadocs.

The version of JDK Javadocs is determined by the `jdkVersion` option.

Note: Links are generated when `noJdkLink` is set to `false`.

Default: `false`

includes
A list of Markdown files that contain module and package documentation

The contents of specified files are parsed and embedded into documentation as module and package descriptions.

classpath
The classpath for analysis and interactive samples.

This is useful if some types that come from dependencies are not resolved/picked up automatically. This option accepts both `.jar` and `.klib` files.

Default: `{project.compileClasspathElements}`

samples
A list of directories or files that contain sample functions which are referenced via @sample KDoc tag.

Source link configuration

The `sourceLinks` configuration block allows you to add a `source` link to each signature that leads to the `url` with a specific line number. (The line number is configurable by setting `lineSuffix`).

This helps readers to find the source code for each declaration.

For an example, see the documentation for the `count()` function in `kotlinx.coroutines`.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!--   ...   -->
    <configuration>
        <sourceLinks>
            <link>
                <path>src</path>
                <url>https://github.com/kotlin/dokka/tree/master/src</url>
                <lineSuffix>#L</lineSuffix>
            </link>
        </sourceLinks>
    </configuration>
</plugin>
```

path
The path to the local source directory. The path must be relative to the root of the current module.

Note: Only Unix based paths are allowed, Windows-style paths will throw an error.

url
The URL of the source code hosting service that can be accessed by documentation readers, like GitHub, GitLab, Bitbucket, etc. This URL is used to generate source code links of declarations.

lineSuffix
The suffix used to append source code line number to the URL. This helps readers navigate not only to the file, but to the specific line number of the declaration.

The number itself is appended to the specified suffix. For example, if this option is set to `#L` and the line number is 10, the resulting URL suffix is `#L10`.

Suffixes used by popular services:

- GitHub: `#L`
- GitLab: `#L`
- Bitbucket: `#lines-`

External documentation links configuration

The `externalDocumentationLinks` block allows the creation of links that lead to the externally hosted documentation of your dependencies.

For example, if you are using types from `kotlinx.serialization`, by default they are unclickable in your documentation, as if they are unresolved. However, since the API reference documentation for `kotlinx.serialization` is built by Dokka and is published on kotlinlang.org, you can configure external documentation links for it. Thus allowing Dokka to generate links for types from the library, making them resolve successfully and clickable.

By default, external documentation links for Kotlin standard library and JDK are configured.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!-- ... -->
    <configuration>
        <externalDocumentationLinks>
            <link>
                <url>https://kotlinlang.org/api/kotlinx.serialization/</url>
                <packageListUrl>file:/${project.basedir}/serialization.package.list</packageListUrl>
            </link>
        </externalDocumentationLinks>
    </configuration>
</plugin>
```

url

The root URL of documentation to link to. It must contain a trailing slash.

Dokka does its best to automatically find the `package-list` for the given URL, and link declarations together.

If automatic resolution fails or if you want to use locally cached files instead, consider setting the `packageListUrl` option.

packageListUrl

The exact location of a `package-list`. This is an alternative to relying on Dokka automatically resolving it.

Package lists contain information about the documentation and the project itself, such as module and package names.

This can also be a locally cached file to avoid network calls.

Package options

The `perPackageOptions` configuration block allows setting some options for specific packages matched by `matchingRegex`.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!-- ... -->
    <configuration>
        <perPackageOptions>
            <packageOptions>
                <matchingRegex>.*api.*</matchingRegex>
                <suppress>false</suppress>
                <reportUndocumented>false</reportUndocumented>
                <skipDeprecated>false</skipDeprecated>
                <documentedVisibilities>
                    <visibility>PUBLIC</visibility>
                    <visibility>PRIVATE</visibility>
                    <visibility>PROTECTED</visibility>
                    <visibility>INTERNAL</visibility>
                    <visibility>PACKAGE</visibility>
                </documentedVisibilities>
            </packageOptions>
        </perPackageOptions>
    </configuration>
</plugin>
```

matchingRegex

The regular expression that is used to match the package.

Default: `.*`

suppress

Whether this package should be skipped when generating documentation.

Default: `false`

documentedVisibilities

The set of visibility modifiers that should be documented.

This can be used if you want to document `protected` / `internal` / `private` declarations within this package, as well as if you want to exclude `public` declarations and only document internal API.

Default: `PUBLIC`

skipDeprecated

Whether to document declarations annotated with `@Deprecated`.

This can be set on project/module level.

Default: `false`

reportUndocumented

Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by `documentedVisibilities` and other filters.

This setting works well with `failOnWarning`.

Default: `false`

Complete configuration

Below you can see all the possible configuration options applied at the same time.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!-- ... -->
    <configuration>
        <skip>false</skip>
        <moduleName>${project.artifactId}</moduleName>
        <outputDir>${project.basedir}/target/documentation</outputDir>
        <failOnWarning>false</failOnWarning>
        <suppressObviousFunctions>true</suppressObviousFunctions>
        <suppressInheritedMembers>false</suppressInheritedMembers>
        <offlineMode>false</offlineMode>
        <sourceDirectories>
            <dir>${project.basedir}/src</dir>
        </sourceDirectories>
        <documentedVisibilities>
            <visibility>PUBLIC</visibility>
            <visibility>PRIVATE</visibility>
            <visibility>PROTECTED</visibility>
            <visibility>INTERNAL</visibility>
            <visibility>PACKAGE</visibility>
        </documentedVisibilities>
        <reportUndocumented>false</reportUndocumented>
        <skipDeprecated>false</skipDeprecated>
        <skipEmptyPackages>true</skipEmptyPackages>
        <suppressedFiles>
            <file>/path/to/dir</file>
            <file>/path/to/file</file>
        </suppressedFiles>
        <jdkVersion>8</jdkVersion>
        <languageVersion>1.7</languageVersion>
        <apiVersion>1.7</apiVersion>
        <noStdlibLink>false</noStdlibLink>
        <noJdkLink>false</noJdkLink>
        <includes>
            <include>packages.md</include>
            <include>extra.md</include>
        </includes>
        <classpath>${project.compileClasspathElements}</classpath>
        <samples>
            <dir>${project.basedir}/samples</dir>
        </samples>
        <sourceLinks>
            <link>
                <path>src</path>
                <url>https://github.com/kotlin/dokka/tree/master/src</url>
                <lineSuffix>#L</lineSuffix>
            </link>
        </sourceLinks>
        <externalDocumentationLinks>
            <link>
                <url>https://kotlinlang.org/api/core/kotlin-stdlib/</url>
                <packageListUrl>file:/${project.basedir}/stdlib.package.list</packageListUrl>
            </link>
        </externalDocumentationLinks>
        <perPackageOptions>
            <packageOptions>
                <matchingRegex>.*api.*</matchingRegex>
                <suppress>false</suppress>
                <reportUndocumented>false</reportUndocumented>
                <skipDeprecated>false</skipDeprecated>
                <documentedVisibilities>
                    <visibility>PUBLIC</visibility>
```

```
                    <visibility>PRIVATE</visibility>
                    <visibility>PROTECTED</visibility>
                    <visibility>INTERNAL</visibility>
                    <visibility>PACKAGE</visibility>
                </documentedVisibilities>
            </packageOptions>
        </perPackageOptions>
    </configuration>
</plugin>
```

CLI

If for some reason you cannot use Gradle or Maven build tools, Dokka has a command line (CLI) runner for generating documentation.

In comparison, it has the same, if not more, capabilities as the Gradle plugin for Dokka. Although it is considerably more difficult to set up as there is no autoconfiguration, especially in multiplatform and multi-module environments.

Get started

The CLI runner is published to Maven Central as a separate runnable artifact.

You can find it on Maven Central or download it directly.

With the `dokka-cli-2.2.0.jar` file saved on your computer, run it with the `-help` option to see all available configuration options and their description:

```
java -jar dokka-cli-2.2.0.jar -help
```

It also works for some nested options, such as `-sourceSet`:

```
java -jar dokka-cli-2.2.0.jar -sourceSet -help
```

Generate documentation

Prerequisites

Since there is no build tool to manage dependencies, you have to provide dependency `.jar` files yourself.

Listed below are the dependencies that you need for any output format:

Group	Artifact	Version	Link
`org.jetbrains.dokka`	`dokka-base`	2.2.0	download
`org.jetbrains.dokka`	`analysis-kotlin-symbols`	2.2.0	download

Below are the additional dependencies that you need for HTML output format:

Group	Artifact	Version	Link
`org.jetbrains.kotlinx`	`kotlinx-html-jvm`	0.8.0	download
`org.freemarker`	`freemarker`	2.3.31	download

Run with command line options

You can pass command line options to configure the CLI runner.

At the very least you need to provide the following options:

- `-pluginsClasspath` - a list of absolute/relative paths to downloaded dependencies, separated by semi-colons `;`
- `-sourceSet` - an absolute path to code sources to generate documentation for
- `-outputDir` - an absolute/relative path of the documentation output directory

```
java -jar dokka-cli-2.2.0.jar \
     -pluginsClasspath "./dokka-base-2.2.0.jar;./analysis-kotlin-symbols-2.2.0.jar;./kotlinx-html-jvm-0.8.0.jar;./freemarker-2.3.31.jar" \
     -sourceSet "-src /home/myCoolProject/src/main/kotlin" \
     -outputDir "./dokka/html"
```

Executing the given example generates documentation in HTML output format.

See Command line options for more configuration details.

Run with JSON configuration

It's possible to configure the CLI runner with JSON. In this case, you need to provide an absolute/relative path to the JSON configuration file as the first and only argument. All other configuration options are parsed from it.

```
java -jar dokka-cli-2.2.0.jar dokka-configuration.json
```

At the very least, you need the following JSON configuration file:

```
{
  "outputDir": "./dokka/html",
  "sourceSets": [
    {
      "sourceSetID": {
        "scopeId": "moduleName",
        "sourceSetName": "main"
      },
      "sourceRoots": [
        "/home/myCoolProject/src/main/kotlin"
      ]
    }
  ],
  "pluginsClasspath": [
    "./dokka-base-2.2.0.jar",
    "./kotlinx-html-jvm-0.8.0.jar",
    "./analysis-kotlin-symbols-2.2.0.jar",
    "./freemarker-2.3.31.jar"
  ]
}
```

See JSON configuration options for more details.

Other output formats

By default, the `dokka-base` artifact contains the HTML output format only.

All other output formats are implemented as Dokka plugins. In order to use them, you have to put them on the plugins classpath.

For example, if you want to generate documentation in the experimental GFM output format, you need to download and pass gfm-plugin's JAR (download) into the `pluginsClasspath` configuration option.

Via command line options:

```
java -jar dokka-cli-2.2.0.jar \
     -pluginsClasspath "./dokka-base-2.2.0.jar;...;./gfm-plugin-2.2.0.jar" \
     ...
```

Via JSON configuration:

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-2.2.0.jar",
    "...",
    "./gfm-plugin-2.2.0.jar"
  ],
  ...
}
```

With the GFM plugin passed to `pluginsClasspath`, the CLI runner generates documentation in the GFM output format.

For more information, see GFM and Javadoc pages.

Command line options

To see the list of all possible command line options and their detailed description, run:

```
java -jar dokka-cli-2.2.0.jar -help
```

Short summary:

Option	Description
`moduleName`	Name of the project/module.
`moduleVersion`	Documented version.
`outputDir`	Output directory path, `./dokka` by default.
`sourceSet`	Configuration for a Dokka source set. Contains nested configuration options.
`pluginsConfiguration`	Configuration for Dokka plugins.
`pluginsClasspath`	List of jars with Dokka plugins and their dependencies. Accepts multiple paths separated by semicolons.
`offlineMode`	Whether to resolve remote files/links over network.
`failOnWarning`	Whether to fail documentation generation if Dokka has emitted a warning or an error.
`delayTemplateSubstitution`	Whether to delay substitution of some elements. Used in incremental builds of multi-module projects.
`noSuppressObviousFunctions`	Whether to suppress obvious functions such as those inherited from `kotlin.Any` and `java.lang.Object`.
`includes`	Markdown files that contain module and package documentation. Accepts multiple values separated by semicolons.
`suppressInheritedMembers`	Whether to suppress inherited members that aren't explicitly overridden in a given class.
`globalPackageOptions`	Global list of package configuration options in format `"matchingRegex,-deprecated,-privateApi,+warnUndocumented,+suppress;+visibility:PUBLIC;..."`. Accepts multiple values separated by semicolons.
`globalLinks`	Global external documentation links in format `{url}^{packageListUrl}`. Accepts multiple values separated by `^^`.

Option	Description
`globalSrcLink`	Global mapping between a source directory and a Web service for browsing the code. Accepts multiple paths separated by semicolons.
`helpSourceSet`	Prints help for the nested `-sourceSet` configuration.
`loggingLevel`	Logging level, possible values: `DEBUG, PROGRESS, INFO, WARN, ERROR`.
`help, h`	Usage info.

Source set options

To see the list of command line options for the nested `-sourceSet` configuration, run:

```
java -jar dokka-cli-2.2.0.jar -sourceSet -help
```

Short summary:

Option	Description
`sourceSetName`	Name of the source set.
`displayName`	Display name of the source set, used both internally and externally.
`classpath`	Classpath for analysis and interactive samples. Accepts multiple paths separated by semicolons.
`src`	Source code roots to be analyzed and documented. Accepts multiple paths separated by semicolons.
`dependentSourceSets`	Names of the dependent source sets in format `moduleName/sourceSetName`. Accepts multiple values separated by semicolons.
`samples`	List of directories or files that contain sample functions. Accepts multiple paths separated by semicolons.
`includes`	Markdown files that contain module and package documentation. Accepts multiple paths separated by semicolons.
`documentedVisibilities`	Visibilities to be documented. Accepts multiple values separated by semicolons. Possible values: `PUBLIC`, `PRIVATE`, `PROTECTED`, `INTERNAL`, `PACKAGE`.
`reportUndocumented`	Whether to report undocumented declarations.

Option	Description
`noSkipEmptyPackages`	Whether to create pages for empty packages.
`skipDeprecated`	Whether to skip deprecated declarations.
`jdkVersion`	Version of JDK to use for linking to JDK Javadocs.
`languageVersion`	Language version used for setting up analysis and samples.
`apiVersion`	Kotlin API version used for setting up analysis and samples.
`noStdlibLink`	Whether to generate links to the Kotlin standard library.
`noJdkLink`	Whether to generate links to JDK Javadocs.
`suppressedFiles`	Paths to files to be suppressed. Accepts multiple paths separated by semicolons.
`suppressAnnotatedWith`	Annotation fully qualified names (FQNs) to suppress declarations annotated with. Accepts multiple values separated by semicolons.
`analysisPlatform`	Platform used for setting up analysis.
`perPackageOptions`	List of package source set configurations in format `matchingRegexp,-deprecated,-privateApi,+warnUndocumented,+suppress;...`. Accepts multiple values separated by semicolons.
`externalDocumentationLinks`	External documentation links in format `{url}^{packageListUrl}`. Accepts multiple values separated by `^^`.
`srcLink`	Mapping between a source directory and a Web service for browsing the code. Accepts multiple paths separated by semicolons.

JSON configuration

Below are some examples and detailed descriptions for each configuration section. You can also find an example with all configuration options applied at the bottom of the page.

General configuration

```
{
  "moduleName": "Dokka Example",
  "moduleVersion": null,
  "outputDir": "./build/dokka/html",
  "failOnWarning": false,
  "suppressObviousFunctions": true,
  "suppressInheritedMembers": false,
  "offlineMode": false,
  "includes": [
    "module.md"
  ],
  "sourceLinks":  [
    { "_comment": "Options are described in a separate section" }
  ],
  "perPackageOptions": [
    { "_comment": "Options are described in a separate section" }
  ],
  "externalDocumentationLinks":  [
    { "_comment": "Options are described in a separate section" }
  ],
  "sourceSets": [
    { "_comment": "Options are described in a separate section" }
  ],
  "pluginsClasspath": [
    "./dokka-base-2.2.0.jar",
    "./kotlinx-html-jvm-0.8.0.jar",
    "./analysis-kotlin-symbols-2.2.0.jar",
    "./freemarker-2.3.31.jar"
  ]
}
```

moduleName
The display name used to refer to the module. It is used for the table of contents, navigation, logging, etc.

Default: `root`

moduleVersion
The module version.

Default: empty

outputDirectory
The directory to where documentation is generated, regardless of output format.

Default: `./dokka`

failOnWarning
Whether to fail documentation generation if Dokka has emitted a warning or an error. The process waits until all errors and warnings have been emitted first.

This setting works well with `reportUndocumented`

Default: `false`

suppressObviousFunctions
Whether to suppress obvious functions.

A function is considered to be obvious if it is:

- Inherited from `kotlin.Any`, `Kotlin.Enum`, `java.lang.Object` or `java.lang.Enum`, such as `equals`, `hashCode`, `toString`.

- Synthetic (generated by the compiler) and does not have any documentation, such as `dataClass.componentN` or `dataClass.copy`.

Default: `true`

suppressInheritedMembers
Whether to suppress inherited members that aren't explicitly overridden in a given class.

Note: This can suppress functions such as `equals` / `hashCode` / `toString`, but cannot suppress synthetic functions such as `dataClass.componentN` and `dataClass.copy`. Use `suppressObviousFunctions` for that.

Default: `false`

offlineMode
Whether to resolve remote files/links over your network.

This includes package-lists used for generating external documentation links. For example, to make classes from the standard library clickable.

Setting this to `true` can significantly speed up build times in certain cases, but can also worsen documentation quality and user experience. For example, by not resolving class/member links from your dependencies, including the standard library.

Note: You can cache fetched files locally and provide them to Dokka as local paths. See `externalDocumentationLinks` section.

Default: `false`

includes
A list of Markdown files that contain module and package documentation.

The contents of specified files are parsed and embedded into documentation as module and package descriptions.

This can be configured on per-package basis.

sourceSets
Individual and additional configuration of Kotlin source sets.

For a list of possible options, see source set configuration.

sourceLinks
The global configuration of source links that is applied for all source sets.

For a list of possible options, see source link configuration.

perPackageOptions
The global configuration of matched packages, regardless of the source set they are in.

For a list of possible options, see per-package configuration.

externalDocumentationLinks
The global configuration of external documentation links, regardless of the source set they are used in.

For a list of possible options, see external documentation links configuration.

pluginsClasspath
A list of JAR files with Dokka plugins and their dependencies.

Source set configuration

How to configure Kotlin source sets:

```
{
  "sourceSets": [
    {
      "displayName": "jvm",
      "sourceSetID": {
        "scopeId": "moduleName",
        "sourceSetName": "main"
      },
      "dependentSourceSets": [
        {
          "scopeId": "dependentSourceSetScopeId",
          "sourceSetName": "dependentSourceSetName"
        }
      ],
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAGE"],
      "reportUndocumented": false,
      "skipEmptyPackages": true,
      "skipDeprecated": false,
      "jdkVersion": 8,
      "languageVersion": "1.7",
      "apiVersion": "1.7",
      "noStdlibLink": false,
      "noJdkLink": false,
      "includes": [
        "module.md"
      ],
      "analysisPlatform": "jvm",
      "sourceRoots": [
        "/home/ignat/IdeaProjects/dokka-debug-mvn/src/main/kotlin"
      ],
      "classpath": [
        "libs/kotlin-stdlib-2.4.20.jar",
        "libs/kotlin-stdlib-common-2.4.20.jar"
      ],
      "samples": [
        "samples/basic.kt"
      ],
      "suppressedFiles": [
        "src/main/kotlin/org/jetbrains/dokka/Suppressed.kt"
      ],
      "sourceLinks":  [
        { "_comment": "Options are described in a separate section" }
      ],
      "perPackageOptions": [
        { "_comment": "Options are described in a separate section" }
      ],
      "externalDocumentationLinks":  [
        { "_comment": "Options are described in a separate section" }
      ]
    }
  ]
}
```

displayName

The display name used to refer to this source set.

The name is used both externally (for example, the source set name is visible to documentation readers) and internally (for example, for logging messages of `reportUndocumented`).

The platform name can be used if you don't have a better alternative.

sourceSetID

The technical ID of the source set

documentedVisibilities
The set of visibility modifiers that should be documented.

This can be used if you want to document `protected` / `internal` / `private` declarations, as well as if you want to exclude `public` declarations and only document internal API.

This can be configured on per-package basis.

Possible values:

- `PUBLIC`
- `PRIVATE`
- `PROTECTED`
- `INTERNAL`
- `PACKAGE`

Default: `PUBLIC`

reportUndocumented
Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by `documentedVisibilities` and other filters.

This setting works well with `failOnWarning`.

This can be configured on per-package basis.

Default: `false`

skipEmptyPackages
Whether to skip packages that contain no visible declarations after various filters have been applied.

For example, if `skipDeprecated` is set to `true` and your package contains only deprecated declarations, it is considered to be empty.

Default for CLI runner is `false`.

skipDeprecated
Whether to document declarations annotated with `@Deprecated`.

This can be configured on per-package basis.

Default: `false`

jdkVersion
The JDK version to use when generating external documentation links for Java types.

For example, if you use `java.util.UUID` in some public declaration signature, and this option is set to `8`, Dokka generates an external documentation link to JDK 8 Javadocs for it.

languageVersion
The Kotlin language version used for setting up analysis and @sample environment.

apiVersion
The Kotlin API version used for setting up analysis and @sample environment.

noStdlibLink
Whether to generate external documentation links that lead to the API reference documentation of Kotlin's standard library.

Note: Links are generated when `noStdLibLink` is set to `false`.

Default: `false`

noJdkLink
Whether to generate external documentation links to JDK's Javadocs.

The version of JDK Javadocs is determined by the `jdkVersion` option.

Note: Links are generated when `noJdkLink` is set to `false`.

Default: `false`

includes
A list of Markdown files that contain module and package documentation.

The contents of the specified files are parsed and embedded into documentation as module and package descriptions.

analysisPlatform
Platform to be used for setting up code analysis and @sample environment.

Possible values:

- `jvm`
- `common`
- `js`
- `native`

sourceRoots
The source code roots to be analyzed and documented. Acceptable inputs are directories and individual `.kt` / `.java` files.

classpath
The classpath for analysis and interactive samples.

This is useful if some types that come from dependencies are not resolved/picked up automatically.

This option accepts both `.jar` and `.klib` files.

samples
A list of directories or files that contain sample functions which are referenced via the @sample KDoc tag.

suppressedFiles
The files to be suppressed when generating documentation.

suppressAnnotatedWith
A list of annotation fully qualified names (FQNs) to suppress declarations annotated with.

Any declaration annotated with one of these annotations is excluded from the generated documentation.

sourceLinks
A set of parameters for source links that is applied only for this source set.

For a list of possible options, see source link configuration.

perPackageOptions
A set of parameters specific to matched packages within this source set.

For a list of possible options, see per-package configuration.

externalDocumentationLinks
A set of parameters for external documentation links that is applied only for this source set.

For a list of possible options, see external documentation links configuration.

Source link configuration

The `sourceLinks` configuration block allows you to add a `source` link to each signature that leads to the `remoteUrl` with a specific line number. (The line number is configurable by setting `remoteLineSuffix`).

This helps readers to find the source code for each declaration.

For an example, see the documentation for the `count()` function in `kotlinx.coroutines`.

You can configure source links for all source sets together at the same time, or individually:

```
{
  "sourceLinks": [
    {
      "localDirectory": "src/main/kotlin",
      "remoteUrl": "https://github.com/Kotlin/dokka/tree/master/src/main/kotlin",
      "remoteLineSuffix": "#L"
    }
  ]
}
```

localDirectory

The path to the local source directory.

remoteUrl

The URL of the source code hosting service that can be accessed by documentation readers, like GitHub, GitLab, Bitbucket, etc. This URL is used to generate source code links of declarations.

remoteLineSuffix

The suffix used to append the source code line number to the URL. This helps readers navigate not only to the file, but to the specific line number of the declaration.

The number itself is appended to the specified suffix. For example, if this option is set to `#L` and the line number is 10, the resulting URL suffix is `#L10`.

Suffixes used by popular services:

- GitHub: `#L`
- GitLab: `#L`
- Bitbucket: `#lines-`

Default: empty (no suffix)

Per-package configuration

The `perPackageOptions` configuration block allows setting some options for specific packages matched by `matchingRegex`.

You can add package configurations for all source sets together at the same time, or individually:

```
{
  "perPackageOptions": [
    {
      "matchingRegex": ".*internal.*",
      "suppress": false,
      "skipDeprecated": false,
      "reportUndocumented": false,
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAGE"]
    }
  ]
}
```

matchingRegex
The regular expression that is used to match the package.

suppress
Whether this package should be skipped when generating documentation.

Default: `false`

skipDeprecated
Whether to document declarations annotated with `@Deprecated`.

This can be set on project/module level.

Default: `false`

reportUndocumented
Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by `documentedVisibilities` and other filters.

This setting works well with `failOnWarning`.

This can be configured on source set level.

Default: `false`

documentedVisibilities
The set of visibility modifiers that should be documented.

This can be used if you want to document `protected` / `internal` / `private` declarations within this package, as well as if you want to exclude `public` declarations and only document internal API.

Can be configured on source set level.

Default: `PUBLIC`

External documentation links configuration

The `externalDocumentationLinks` block allows the creation of links that lead to the externally hosted documentation of your dependencies.

For example, if you are using types from `kotlinx.serialization`, by default they are unclickable in your documentation, as if they are unresolved. However, since the API reference documentation for `kotlinx.serialization` is built by Dokka and is published on kotlinlang.org, you can configure external documentation links for it. Thus allowing Dokka to generate links for types from the library, making them resolve successfully and clickable.

You can configure external documentation links for all source sets together at the same time, or individually:

```
{
  "externalDocumentationLinks": [
    {
      "url": "https://kotlinlang.org/api/kotlinx.serialization/",
      "packageListUrl": "https://kotlinlang.org/api/kotlinx.serialization/package-list"
    }
  ]
}
```

url

The root URL of documentation to link to. It must contain a trailing slash.

Dokka does its best to automatically find `package-list` for the given URL, and link declarations together.

If automatic resolution fails or if you want to use locally cached files instead, consider setting the `packageListUrl` option.

packageListUrl

The exact location of a `package-list`. This is an alternative to relying on Dokka automatically resolving it.

Package lists contain information about the documentation and the project itself, such as module and package names.

This can also be a locally cached file to avoid network calls.

Complete configuration

Below you can see all possible configuration options applied at the same time.

```
{
  "moduleName": "Dokka Example",
  "moduleVersion": null,
  "outputDir": "./build/dokka/html",
  "failOnWarning": false,
  "suppressObviousFunctions": true,
  "suppressInheritedMembers": false,
  "offlineMode": false,
  "suppressAnnotatedWith": [
    "com.example.SuppressMe"
  ],
  "sourceLinks": [
    {
      "localDirectory": "src/main/kotlin",
      "remoteUrl": "https://github.com/Kotlin/dokka/tree/master/src/main/kotlin",
      "remoteLineSuffix": "#L"
    }
  ],
  "externalDocumentationLinks": [
    {
      "url": "https://docs.oracle.com/javase/8/docs/api/",
      "packageListUrl": "https://docs.oracle.com/javase/8/docs/api/package-list"
    },
    {
      "url": "https://kotlinlang.org/api/core/kotlin-stdlib/",
      "packageListUrl": "https://kotlinlang.org/api/core/kotlin-stdlib/package-list"
    }
  ],
  "perPackageOptions": [
    {
      "matchingRegex": ".*internal.*",
      "suppress": false,
      "reportUndocumented": false,
      "skipDeprecated": false,
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAGE"]
    }
  ],
  "sourceSets": [
    {
      "displayName": "jvm",
      "sourceSetID": {
        "scopeId": "moduleName",
        "sourceSetName": "main"
      },
      "dependentSourceSets": [
        {
          "scopeId": "dependentSourceSetScopeId",
          "sourceSetName": "dependentSourceSetName"
        }
      ],
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAGE"],
      "reportUndocumented": false,
      "skipEmptyPackages": true,
      "skipDeprecated": false,
      "jdkVersion": 8,
      "languageVersion": "1.7",
      "apiVersion": "1.7",
      "noStdlibLink": false,
      "noJdkLink": false,
      "includes": [
        "module.md"
      ],
      "analysisPlatform": "jvm",
```

```
      "sourceRoots": [
        "/home/ignat/IdeaProjects/dokka-debug-mvn/src/main/kotlin"
      ],
      "classpath": [
        "libs/kotlin-stdlib-2.4.20.jar",
        "libs/kotlin-stdlib-common-2.4.20.jar"
      ],
      "samples": [
        "samples/basic.kt"
      ],
      "suppressedFiles": [
        "src/main/kotlin/org/jetbrains/dokka/Suppressed.kt"
      ],
      "suppressAnnotatedWith": [
        "com.example.SuppressMe"
      ],
      "sourceLinks": [
        {
          "localDirectory": "src/main/kotlin",
          "remoteUrl": "https://github.com/Kotlin/dokka/tree/master/src/main/kotlin",
          "remoteLineSuffix": "#L"
        }
      ],
      "externalDocumentationLinks": [
        {
          "url": "https://docs.oracle.com/javase/8/docs/api/",
          "packageListUrl": "https://docs.oracle.com/javase/8/docs/api/package-list"
        },
        {
          "url": "https://kotlinlang.org/api/core/kotlin-stdlib/",
          "packageListUrl": "https://kotlinlang.org/api/core/kotlin-stdlib/package-list"
        }
      ],
      "perPackageOptions": [
        {
          "matchingRegex": ".*internal.*",
          "suppress": false,
          "reportUndocumented": false,
          "skipDeprecated": false,
          "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAGE"]
        }
      ]
    }
  ],
  "pluginsClasspath": [
    "./dokka-base-2.2.0.jar",
    "./kotlinx-html-jvm-0.8.0.jar",
    "./analysis-kotlin-symbols-2.2.0.jar",
    "./freemarker-2.3.31.jar"
  ],
  "pluginsConfiguration": [
    {
      "fqPluginName": "org.jetbrains.dokka.base.DokkaBase",
      "serializationFormat": "JSON",
      "values": "{\"separateInheritedMembers\":false,\"footerMessage\":\"© 2021 pretty good
Copyright\"}"
    }
  ],
  "includes": [
    "module.md"
  ]
}
```

HTML

This guide applies to Dokka Gradle plugin (DGP) v2 mode. The DGP v1 mode is no longer supported. To upgrade from v1 to v2 mode, follow the Migration guide.

HTML is Dokka's default and recommended output format. It provides support for Kotlin Multiplatform, Android, and Java projects. Additionally, you can use the HTML format to document both single and multi-project builds.

For examples of the HTML output format, check the following docs:

- kotlinx.coroutines
- Bitmovin
- Hexagon
- Ktor
- OkHttp
- Gradle

Generate HTML documentation

HTML as an output format is supported by all runners. To generate HTML documentation, follow these steps depending on your build tool or runner:

- For Gradle, you can run the following tasks:
  - `dokkaGenerate` to generate documentation in all available formats based on the applied plugins. This is the recommended task for most users. When using this task in IntelliJ IDEA, it logs a clickable link to the output.
  - `dokkaGeneratePublicationHtml` to generate documentation only in HTML format. This task exposes the output directory as an `@OutputDirectory`. Use this task when you need to consume the generated files in other Gradle tasks, such as uploading them to a server, moving them into a GitHub Pages directory, or packaging them into a `javadoc.jar`. This task is intentionally not listed in Gradle task groups because it is not meant for everyday use.

    If you're using IntelliJ IDEA, you may see the `dokkaGenerateHtml` Gradle task. This task is simply an alias of `dokkaGeneratePublicationHtml`. Both tasks perform exactly the same operation.

- For Maven, run the `dokka:dokka` goal.
- For CLI runner, run with HTML dependencies set.

HTML pages generated by this format need to be hosted on a web server in order to render everything correctly.

You can use any free static site hosting service, such as GitHub Pages.

Locally, you can use the built-in IntelliJ web server.

Configuration

HTML format is Dokka's base format. You can configure it using the following options:

Gradle Kotlin DSL

```
// build.gradle.kts

dokka {
    pluginsConfiguration.html {
        customAssets.from("logo.png")
        customStyleSheets.from("styles.css")
        footerMessage.set("(c) Your Company")
        separateInheritedMembers.set(false)
        templatesDir.set(file("dokka/templates"))
        mergeImplicitExpectActualDeclarations.set(false)
    }
}
```

Gradle Groovy DSL

```
// build.gradle

dokka {
    pluginsConfiguration {
        html {
            customAssets.from("logo.png")
            customStyleSheets.from("styles.css")
            footerMessage.set("(c) Your Company")
            separateInheritedMembers.set(false)
            templatesDir.set(file("dokka/templates"))
            mergeImplicitExpectActualDeclarations.set(false)
        }
    }
}
```

Maven

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <pluginsConfiguration>
            <!-- Fully qualified plugin name -->
            <org.jetbrains.dokka.base.DokkaBase>
                <!-- Options by name -->
                <customAssets>
                    <asset>${project.basedir}/my-image.png</asset>
                </customAssets>
                <customStyleSheets>
                    <stylesheet>${project.basedir}/my-styles.css</stylesheet>
                </customStyleSheets>
                <footerMessage>(c) MyOrg 2022 Maven</footerMessage>
                <separateInheritedMembers>false</separateInheritedMembers>
                <templatesDir>${project.basedir}/dokka/templates</templatesDir>
                <mergeImplicitExpectActualDeclarations>false</mergeImplicitExpectActualDeclarations>
            </org.jetbrains.dokka.base.DokkaBase>
        </pluginsConfiguration>
    </configuration>
</plugin>
```

Via command line options:

CLI

```
java -jar dokka-cli-2.2.0.jar \
     ...
     -pluginsConfiguration "org.jetbrains.dokka.base.DokkaBase={\"customAssets\": [\"my-image.png\"],
\"customStyleSheets\": [\"my-styles.css\"], \"footerMessage\": \"(c) 2022 MyOrg\",
\"separateInheritedMembers\": false, \"templatesDir\": \"dokka/templates\",
\"mergeImplicitExpectActualDeclarations\": false}
"
```

Via JSON configuration:

```
{
  "moduleName": "Dokka Example",
  "pluginsConfiguration": [
    {
      "fqPluginName": "org.jetbrains.dokka.base.DokkaBase",
      "serializationFormat": "JSON",
      "values": "{\"customAssets\": [\"my-image.png\"], \"customStyleSheets\": [\"my-styles.css\"],
\"footerMessage\": \"(c) 2022 MyOrg\", \"separateInheritedMembers\": false, \"templatesDir\":
\"dokka/templates\", \"mergeImplicitExpectActualDeclarations\": false}"
    }
  ]
}
```

Configuration options

The table below contains all the possible configuration options and their purpose:

Option	Description
`customAssets`	List of paths for image assets to be bundled with documentation. The image assets can have any file extension. For more information, see Customizing assets.
`customStyleSheets`	List of paths for `.css` stylesheets to be bundled with documentation and used for rendering. For more information, see Customizing styles.
`templatesDir`	Path to the directory containing custom HTML templates. For more information, see Templates.
`footerMessage`	The text displayed in the footer.
`separateInheritedMembers`	This is a boolean option. If set to `true`, Dokka renders properties/functions and inherited properties/inherited functions separately. This is disabled by default.
`mergeImplicitExpectActualDeclarations`	This is a boolean option. If set to `true`, Dokka merges declarations that are not declared as expect/actual, but have the same fully qualified name. This can be useful for legacy codebases. This is disabled by default.

For more information about configuring Dokka plugins, see Configuring Dokka plugins.

Customization

To help you add your own look and feel to your documentation, the HTML format supports a number of customization options.

Customize styles

You can use your own stylesheets by using the `customStyleSheets` configuration option. These are applied to every page.

It's also possible to override Dokka's default stylesheets by providing files with the same name:

Stylesheet name	Description
`style.css`	Main stylesheet, contains most of the styles used across all pages
`logo-styles.css`	Header logo styling
`prism.css`	Styles for PrismJS syntax highlighter

The source code for all of Dokka's stylesheets is available on GitHub.

Customize assets

You can provide your own images to be bundled with documentation by using the `customAssets` configuration option.

These files are copied to the `<output>/images` directory.

You can use the `customAssets` property with collections of files (`FileCollection`):

```
customAssets.from("example.png", "example2.png")
```

It's possible to override Dokka's images and icons by providing files with the same name. The most useful and relevant one is `logo-icon.svg` , which is the image used in the header. The rest is mostly icons.

You can find all images used by Dokka on GitHub.

Change the logo

To customize the logo, you can begin by providing your own asset for `logo-icon.svg` .

If you don't like how it looks, or you want to use a `.png` file instead of the default `.svg` file, you can override the `logo-styles.css` stylesheet to customize it.

For an example of how to do this, see our custom format example project.

The maximum supported logo dimensions are 120 pixels in width and 36 pixels in height. If you use a larger image, it will be automatically resized.

Modify the footer

You can modify text in the footer by using the `footerMessage` configuration option.

Templates

Dokka provides the ability to modify FreeMarker templates used for generating documentation pages.

You can change the header completely, add your own banners/menus/search, load analytics, change body styling, and so on.

Dokka uses the following templates:

Template	Description
`base.ftl`	Defines the general design of all pages to be rendered.
`includes/header.ftl`	The page header that by default contains the logo, version, source set selector, light/dark theme switch, and search.
`includes/footer.ftl`	The page footer that contains the `footerMessage` configuration option and copyright.
`includes/page_metadata.ftl`	Metadata used within `<head>` container.

Template	Description
`includes/source_set_selector.ftl`	The source set selector in the header.

The base template is `base.ftl` and it includes all the remaining listed templates. You can find the source code for all of Dokka's templates on GitHub.

You can override any template by using the `templatesDir` configuration option. Dokka searches for the exact template names within the given directory. If it fails to find user-defined templates, it uses the default templates.

Variables

The following variables are available inside all templates:

Variable	Description
`${pageName}`	The page name
`${footerMessage}`	The text which is set by the `footerMessage` configuration option
`${sourceSets}`	A nullable list of source sets for multi-platform pages. Each item has `name`, `platform`, and `filter` properties.
`${projectName}`	The project name. It's available only within the `template_cmd` directive.
`${pathToRoot}`	The path to root from the current page. It's useful for locating assets and is available only within the `template_cmd` directive.

Variables `projectName` and `pathToRoot` are available only within the `template_cmd` directive as they require more context, and thus they need to be resolved at later stages:

```
<@template_cmd name="projectName">
    <span>${projectName}</span>
</@template_cmd>
```

Directives

You can also use the following Dokka-defined directives:

Variable	Description
`<@content/>`	The main page content.

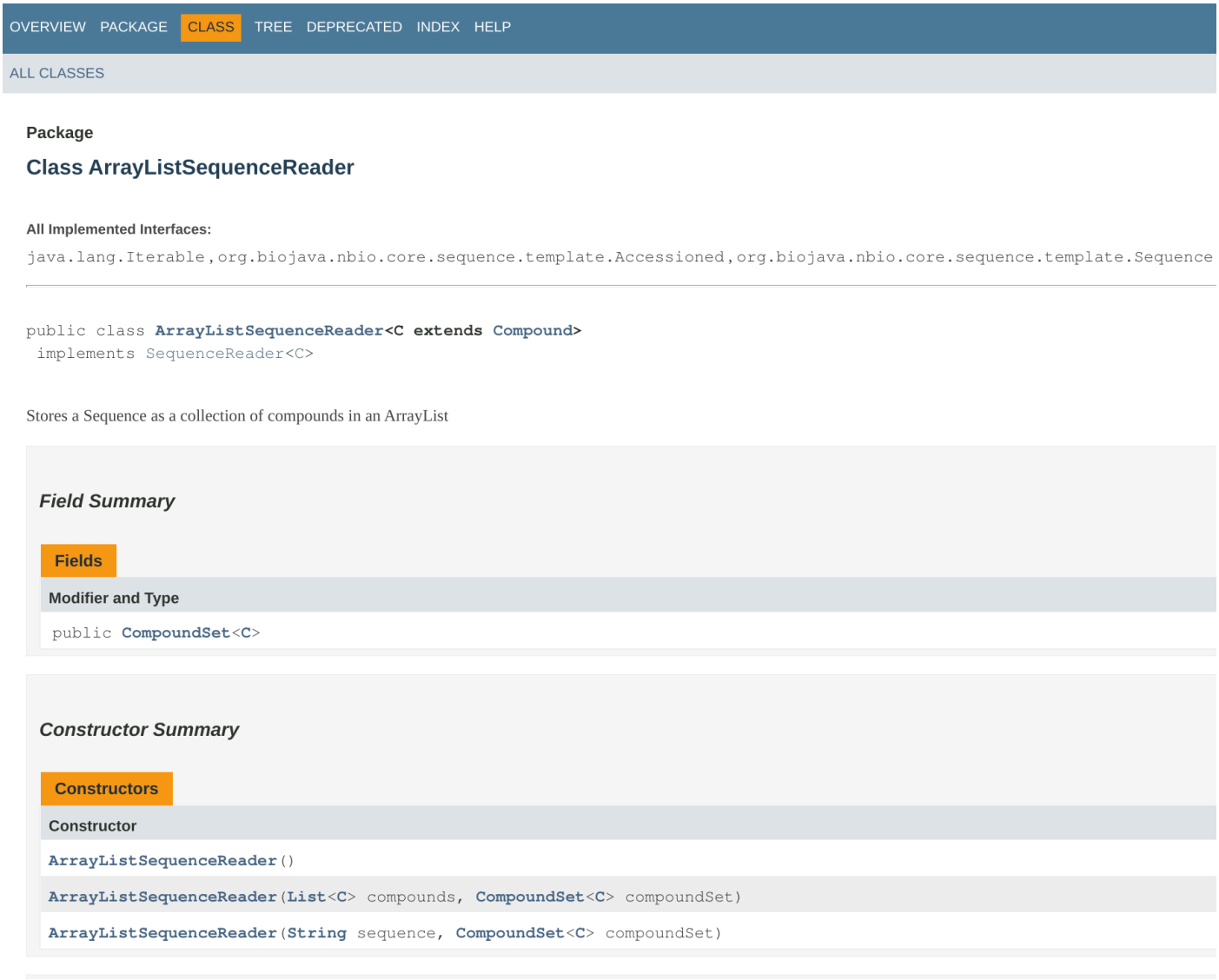
Variable	Description
`<@resources/>`	Resources such as scripts and stylesheets.
`<@version/>`	The subproject version taken from configuration. If the versioning plugin is applied, it is replaced with a version navigator.

Javadoc

> This guide applies to Dokka Gradle plugin (DGP) v2 mode. The DGP v1 mode is no longer supported. To upgrade from v1 to v2 mode, follow the Migration guide.

Dokka's Javadoc output format is a lookalike of Java's Javadoc HTML format.

It tries to visually mimic HTML pages generated by the Javadoc tool, but it's not a direct implementation or an exact copy.



Screenshot of Javadoc output format

All Kotlin code and signatures are rendered as seen from Java's perspective. This is achieved with our Kotlin as Java Dokka plugin, which comes bundled and applied by default for this format.

The Javadoc output format is implemented as a Dokka plugin, and it is maintained by the Dokka team. It is open source, and you can find the source code on GitHub.

Generate Javadoc documentation

Dokka doesn't support the Javadoc format for multi-project builds or Kotlin Multiplatform projects.

The Gradle plugin for Dokka comes with the Javadoc output format included. You need to apply the corresponding plugin ID in the `plugins {}` block of your project's `build.gradle.kts` file:

Gradle Kotlin DSL

```
plugins {
    id("org.jetbrains.dokka-javadoc") version "2.2.0"
}
```

Once you applied the plugin, you can run the following tasks:

- `dokkaGenerate` to generate documentation in all available formats based on the applied plugins.
- `dokkaGeneratePublicationJavadoc` to generate documentation only in Javadoc format.

The `javadoc.jar` file can be generated separately. For more information, see Building `javadoc.jar` .

Maven

The Maven plugin for Dokka comes with the Javadoc output format built in. You can generate documentation by using the following goals:

Goal	Description
`dokka:javadoc`	Generates documentation in Javadoc format
`dokka:javadocJar`	Generates a `javadoc.jar` file that contains documentation in Javadoc format

Since the Javadoc output format is a Dokka plugin, you need to download the plugin's JAR file.

The Javadoc output format has two dependencies that you need to provide as additional JAR files:

- kotlin-as-java plugin
- korte-jvm

Via command line options:

CLI

```
java -jar dokka-cli-2.2.0.jar \
     -pluginsClasspath "./dokka-base-2.2.0.jar;...;./javadoc-plugin-2.2.0.jar" \
     ...
```

Via JSON configuration:

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-2.2.0.jar",
    "...",
    "./kotlin-as-java-plugin-2.2.0.jar",
    "./korte-jvm-3.3.0.jar",
    "./javadoc-plugin-2.2.0.jar"
  ],
  ...
}
```

For more information, see Other output formats in the CLI runner's documentation.

Dokka plugins

> This guide applies to Dokka Gradle plugin (DGP) v2 mode. The DGP v1 mode is no longer supported. To upgrade from v1 to v2 mode, follow the Migration guide.

Dokka was built from the ground up to be easily extensible and highly customizable, which allows the community to implement plugins for missing or very specific features that are not provided out of the box.

Dokka plugins range anywhere from supporting other programming language sources to exotic output formats. You can add support for your own KDoc tags or annotations, teach Dokka how to render different DSLs that are found in KDoc descriptions, visually redesign Dokka's pages to be seamlessly integrated into your company's website, integrate it with other tools and so much more.

If you want to learn how to create Dokka plugins, see Developer guides.

Apply Dokka plugins

Dokka plugins are published as separate artifacts, so to apply a Dokka plugin, you only need to add it as a dependency. From there, the plugin extends Dokka by itself – no further action is needed.

> Plugins that use the same extension points or work in a similar way can interfere with each other. This may lead to visual bugs, general undefined behavior or even failed builds. However, it should not lead to concurrency issues since Dokka does not expose any mutable data structures or objects.
>
> If you notice problems like this, it's a good idea to check which plugins are applied and what they do.

Let's have a look at how you can apply the mathjax plugin to your project:

Gradle Kotlin DSL

```
plugins {
    id("org.jetbrains.dokka") version "2.2.0"
}

dependencies {
    dokkaPlugin("org.jetbrains.dokka:mathjax-plugin")
}
```

- Built-in plugins (like HTML and Javadoc) are always applied automatically. You only configure them and do not need to declare dependencies on them.
- When documenting multi-module projects (multi-project builds), you need to share Dokka configuration and plugins across subprojects.

Gradle Groovy DSL

```
plugins {
    id 'org.jetbrains.dokka' version '2.2.0'
}

dependencies {
    dokkaPlugin 'org.jetbrains.dokka:mathjax-plugin'
}
```

When documenting multi-project builds, you need to share Dokka configuration across subprojects.

Maven

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>mathjax-plugin</artifactId>
                <version>2.2.0</version>
            </plugin>
        </dokkaPlugins>
    </configuration>
</plugin>
```

If you are using the CLI runner with command line options, Dokka plugins should be passed as `.jar` files to `-pluginsClasspath`:

CLI

```
java -jar dokka-cli-2.2.0.jar \
     -pluginsClasspath "./dokka-base-2.2.0.jar;...;./mathjax-plugin-2.2.0.jar" \
     ...
```

If you are using JSON configuration, Dokka plugins should be specified under `pluginsClasspath`.

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-2.2.0.jar",
    "...",
    "./mathjax-plugin-2.2.0.jar"
  ],
  ...
}
```

Configure Dokka plugins

Dokka plugins can also have configuration options of their own. To see which options are available, consult the documentation of the plugins you are using.

Let's have a look at how you can configure the built-in HTML plugin by adding a custom image to the assets (`customAssets` option), custom style sheets (`customStyleSheets` option), and a modified footer message (`footerMessage` option):

To configure Dokka plugins in a type-safe way, use the `dokka.pluginsConfiguration {}` block:

Gradle Kotlin DSL

```
dokka {
    pluginsConfiguration.html {
        customAssets.from("logo.png")
        customStyleSheets.from("styles.css")
        footerMessage.set("(c) Your Company")
    }
}
```

For an example of Dokka plugins configuration, see the Dokka's versioning plugin.

Dokka allows you to extend its functionality and modify the documentation generation process by configuring custom plugins.

Gradle Groovy DSL

```
dokka {
    pluginsConfiguration {
        html {
            customAssets.from("logo.png")
            customStyleSheets.from("styles.css")
            footerMessage.set("(c) Your Company")
        }
    }
}
```

Maven

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <pluginsConfiguration>
            <!-- Fully qualified plugin name -->
            <org.jetbrains.dokka.base.DokkaBase>
                <!-- Options by name -->
                <customAssets>
                    <asset>${project.basedir}/my-image.png</asset>
                </customAssets>
                <customStyleSheets>
                    <stylesheet>${project.basedir}/my-styles.css</stylesheet>
                </customStyleSheets>
                <footerMessage>(c) MyOrg 2022 Maven</footerMessage>
            </org.jetbrains.dokka.base.DokkaBase>
        </pluginsConfiguration>
    </configuration>
</plugin>
```

If you are using the CLI runner with command line options, use the `-pluginsConfiguration` option that accepts JSON configuration in the form of `fullyQualifiedPluginName=json`.

If you need to configure multiple plugins, you can pass multiple values separated by `^^`.

CLI

```
java -jar dokka-cli-2.2.0.jar \
     ...
     -pluginsConfiguration "org.jetbrains.dokka.base.DokkaBase={\"customAssets\": [\"my-image.png\"],
\"customStyleSheets\": [\"my-styles.css\"], \"footerMessage\": \"(c) 2022 MyOrg CLI\"}"
```

If you are using JSON configuration, there exists a similar `pluginsConfiguration` array that accepts JSON configuration in `values`.

```
{
  "moduleName": "Dokka Example",
  "pluginsConfiguration": [
    {
      "fqPluginName": "org.jetbrains.dokka.base.DokkaBase",
      "serializationFormat": "JSON",
      "values": "{\"customAssets\": [\"my-image.png\"], \"customStyleSheets\": [\"my-styles.css\"],
\"footerMessage\": \"(c) 2022 MyOrg\"}"
    }
  ]
}
```

Notable plugins

Here are some notable Dokka plugins that you might find useful:

Name	Description
Android documentation plugin	Improves the documentation experience on Android
Versioning plugin	Adds version selector and helps to organize documentation for different versions of your application/library
MermaidJS HTML plugin	Renders MermaidJS diagrams and visualizations found in KDocs
Mathjax HTML plugin	Pretty prints mathematics found in KDocs
Kotlin as Java plugin	Renders Kotlin signatures as seen from Java's perspective
GFM plugin	Adds the ability to generate documentation in GitHub Flavoured Markdown format
Jekyll plugin	Adds the ability to generate documentation in Jekyll Flavoured Markdown format

If you are a Dokka plugin author and would like to add your plugin to this list, get in touch with maintainers via Slack or GitHub.

Module documentation

Documentation for a subproject as a whole, as well as packages in that subproject, can be provided as separate Markdown files.

File format

Inside the Markdown file, the documentation for the subproject as a whole and for individual packages is introduced by the corresponding first-level headings. The text of the heading **must** be **Module** `<module name>` for a subproject, and **Package** `<package qualified name>` for a package.

The file doesn't have to contain both subproject and package documentation. You can have files that contain only package or subproject documentation. You can even have a Markdown file per subproject or package.

Using Markdown syntax, you can add:

- Headings up to level 6
- Emphasis with bold or italic formatting
- Links
- Inline code
- Code blocks
- Blockquotes

Here's an example file containing both subproject and package documentation:

```
# Module kotlin-demo

This content appears under your subproject name.

# Package org.jetbrains.kotlin.demo

This content appears under your package name in the packages list.
It also appears under the first-level heading on your package's page.

## Level 2 heading for package org.jetbrains.kotlin.demo

Content after this heading is also part of documentation for `org.jetbrains.kotlin.demo`

# Package org.jetbrains.kotlin.demo2

This content appears under your package name in the packages list.
It also appears under the first-level heading on your package's page.

## Level 2 heading for package org.jetbrains.kotlin.demo2

Content after this heading is also part of documentation for `org.jetbrains.kotlin.demo2`
```

To explore an example project with Gradle, see Dokka gradle example.

Pass files to Dokka

To pass these files to Dokka, you need to use the relevant **includes** option for Gradle, Maven, or CLI:

Gradle

Use the `includes` option in General configuration.

Maven

Use the `includes` option in General configuration.

CLI

If you are using command line configuration, use the `includes` option in Source set options.

If you are using JSON configuration, use the `includes` option in General configuration.

Run code snippets

Kotlin code is typically organized into projects with which you work in an IDE, a text editor, or another tool. However, if you want to quickly see how a function works or find an expression's value, there's no need to create a new project and build it. Check out these three handy ways to run Kotlin code instantly in different environments:

- Scratch files in the IDE.
- Kotlin Playground in the browser.
- ki shell in the command line.

IDE: scratches

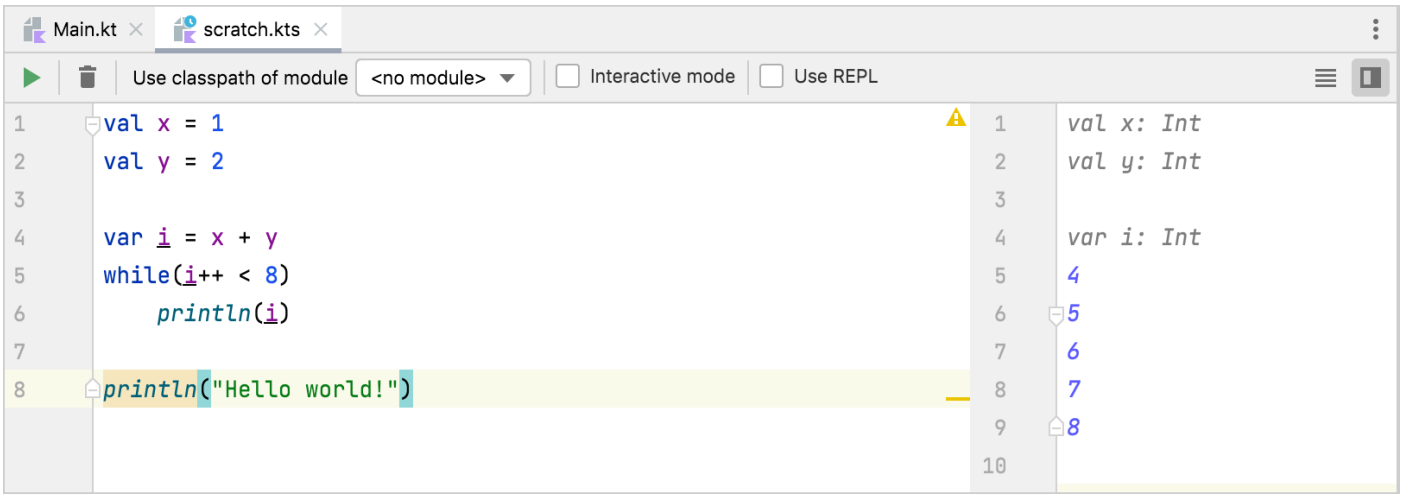
IntelliJ IDEA and Android Studio support Kotlin scratch files.

Scratch files (or just *scratches*) let you create code drafts in the same IDE window as your project and run them on the fly. Scratches are not tied to projects; you can access and run all your scratches from any IntelliJ IDEA window on your OS.

To create a Kotlin scratch, click **File** | **New** | **Scratch File** and select the **Kotlin** type.

Syntax highlighting, auto-completion, and other IntelliJ IDEA code editing features are supported in scratches. There's no need to declare the `main()` function – all the code you write is executed as if it were in the body of `main()`.

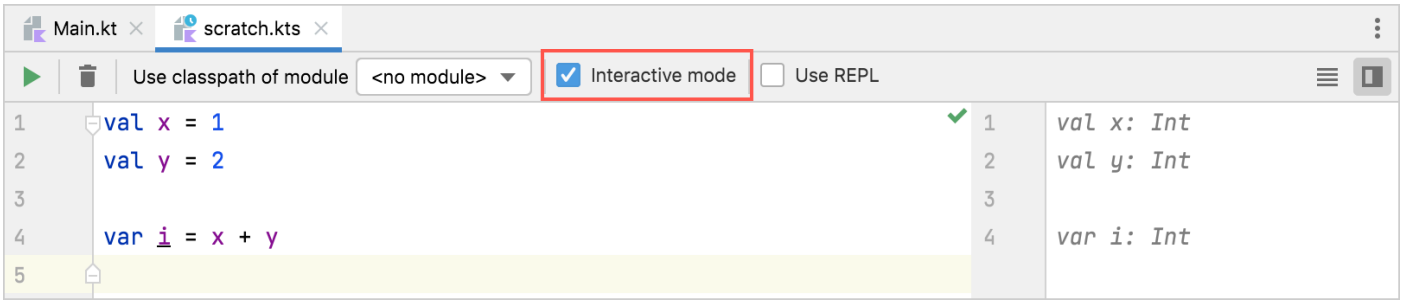
Once you have finished writing your code in a scratch, click **Run**. The execution results will appear in the lines opposite your code.



Run scratch

Interactive mode

The IDE can run code from scratches automatically. To get execution results as soon as you stop typing, switch on **Interactive mode**.



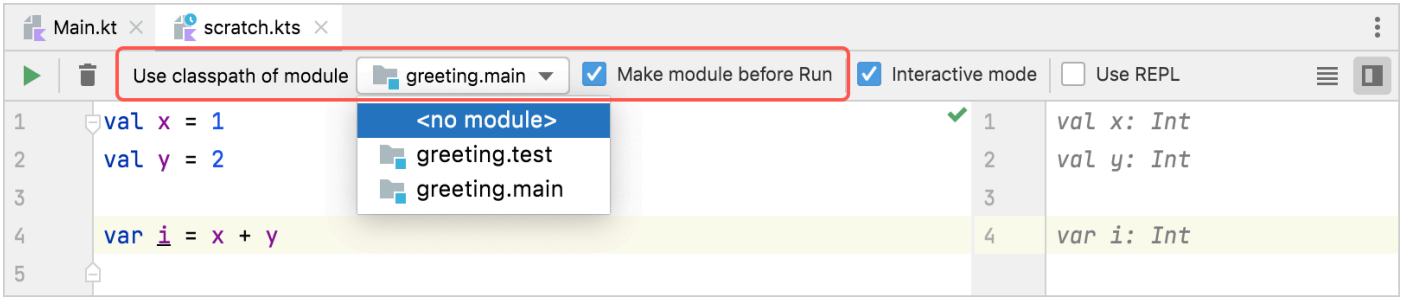
Scratch interactive mode

Use modules

You can use classes or functions from a Kotlin project in your scratches.

To use classes or functions from a project in a scratch, import them into the scratch file with the `import` statement, as usual. Then write your code and run it with the appropriate module selected in the **Use classpath of module** list.

Scratches use the compiled versions of connected modules. So, if you modify a module's source files, the changes will propagate to scratches when you rebuild the module. To rebuild the module automatically before each run of a scratch, select **Make module before Run**.



Scratch select module

Browser: Kotlin Playground

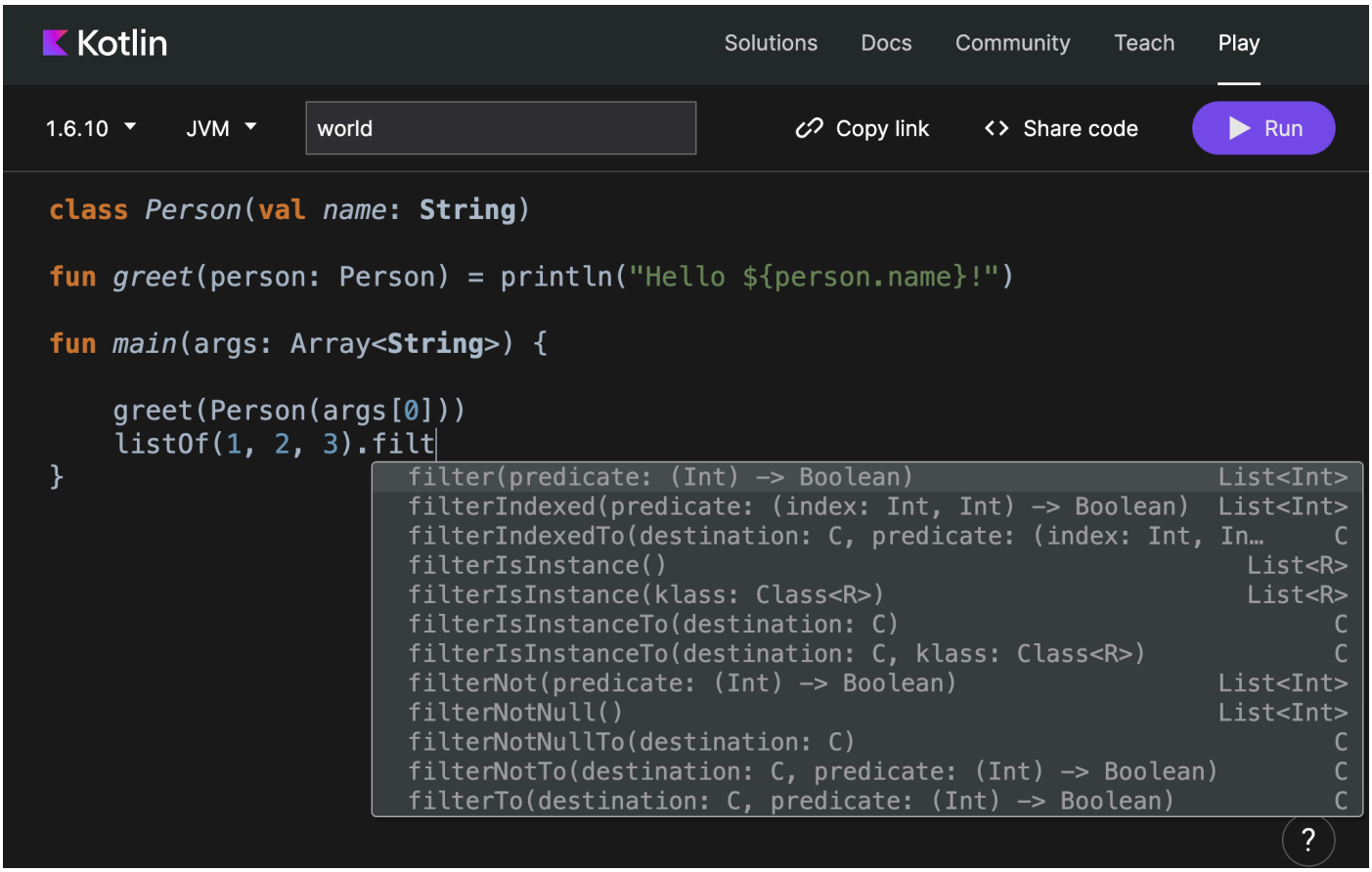
Kotlin Playground is an online application for writing, running, and sharing Kotlin code in your browser.

Write and edit code

In the Playground's editor area, you can write code just as you would in a source file:

- Add your own classes, functions, and top-level declarations in an arbitrary order.
- Write the executable part in the body of the `main()` function.

As in typical Kotlin projects, the `main()` function in the Playground can have the `args` parameter or no parameters at all. To pass program arguments upon execution, write them in the **Program arguments** field.



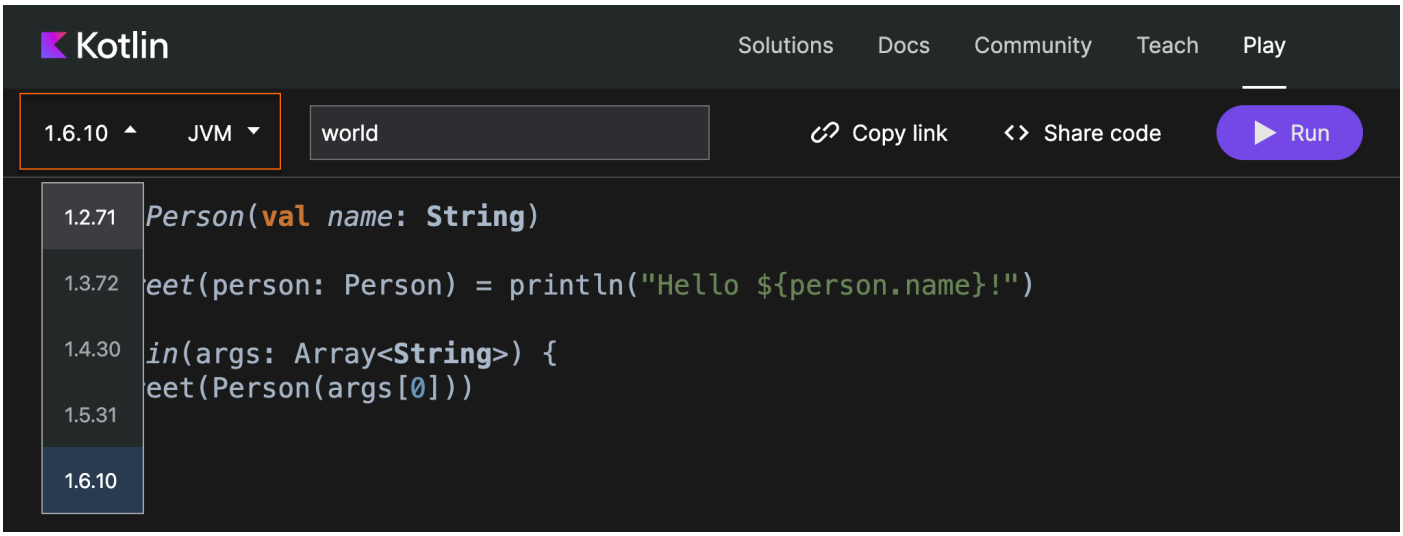
Playground: code completion

The Playground highlights the code and shows code completion options as you type. It automatically imports declarations from the standard library and `kotlinx.coroutines`.

Choose execution environment

The Playground provides ways to customize the execution environment:

- Multiple Kotlin versions, including available previews of future versions.
- Multiple backends to run the code in: JVM, JS (legacy or IR compiler, or Canvas), or JUnit.



Playground: environment setup

For JS backends, you can also see the generated JS code.

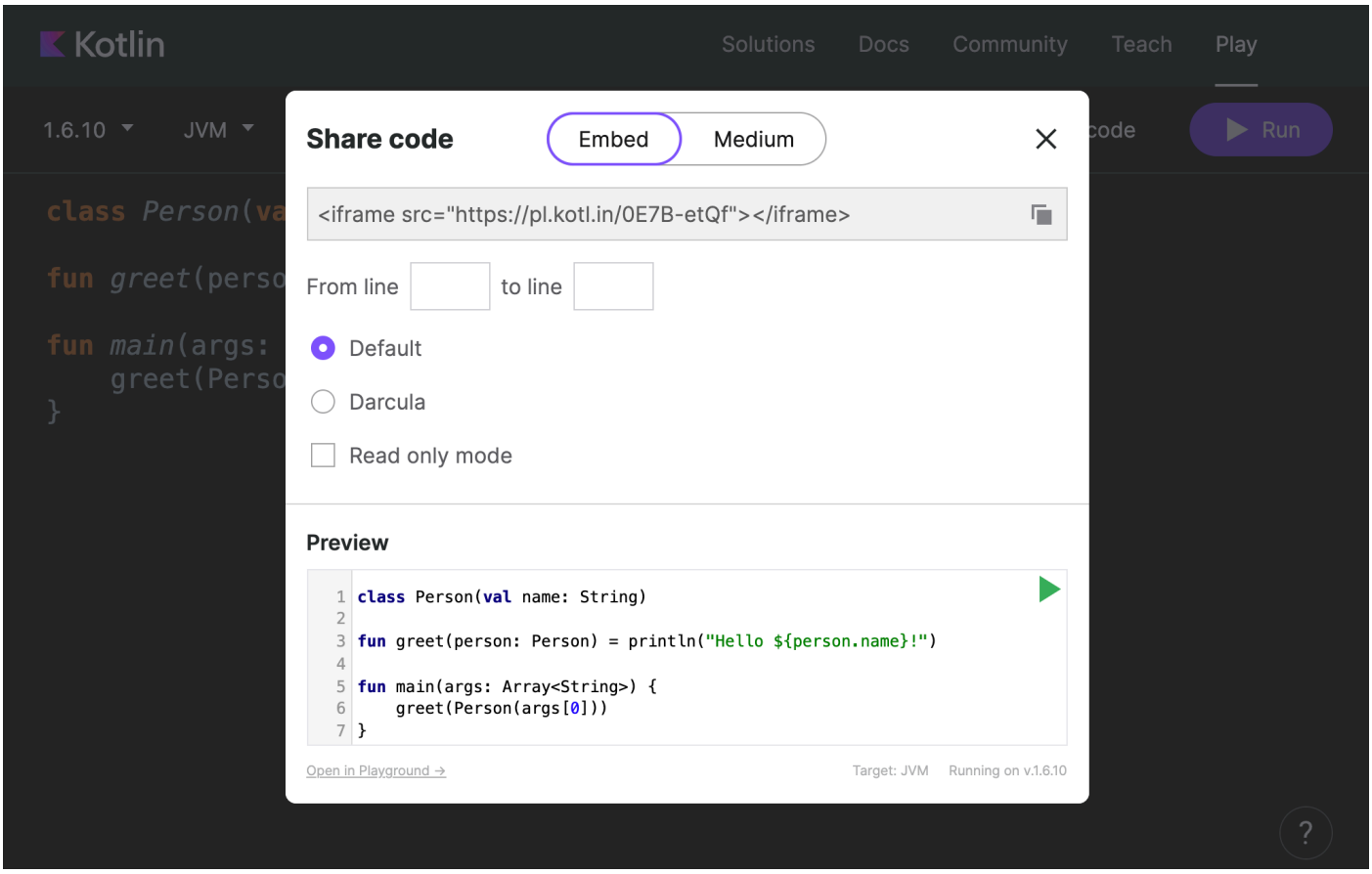


Playground: generated JS

Share code online

Use the Playground to share your code with others – click **Copy link** and send it to anyone you want to show the code to.

You can also embed code snippets from the Playground into other websites and even make them runnable. Click **Share code** to embed your sample into any web page or into a Medium article.



Playground: share code

Command line: ki shell

The ki shell (*Kotlin Interactive Shell*) is a command-line utility for running Kotlin code in the terminal. It's available for Linux, macOS, and Windows.

The ki shell provides basic code evaluation capabilities, along with advanced features such as:

- code completion
- type checks
- external dependencies
- paste mode for code snippets
- scripting support

See the ki shell GitHub repository for more details.

Install and run ki shell

To install the ki shell, download the latest version of it from GitHub and unzip it in the directory of your choice.

On macOS, you can also install the ki shell with Homebrew by running the following command:

```
brew install ki
```

To start the ki shell, run `bin/ki.sh` on Linux and macOS (or just `ki` if the ki shell was installed with Homebrew) or `bin\ki.bat` on Windows.

Once the shell is running, you can immediately start writing Kotlin code in your terminal. Type `:help` (or `:h`) to see the commands that are available in the ki shell.

Code completion and highlighting

The ki shell shows code completion options when you press **Tab**. It also provides syntax highlighting as you type. You can disable this feature by entering `:syntax off`.

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] val x = 1 + 2 + 3
[1] var greeting = "Hello"
[2] listOf(1, 2, 3).fil
file                filterIndexedTo(      filterIsInstanceTo(   filterNotNullTo(
filter {            filterIsInstance(     filterNot {           filterNotTo(
filterIndexed {     filterIsInstance()    filterNotNull()       filterTo(
```

ki shell highlighting and completion

When you press **Enter**, the ki shell evaluates the entered line and prints the result. Expression values are printed as variables with auto-generated names like `res*`. You can later use such variables in the code you run. If the construct entered is incomplete (for example, an `if` with a condition but without the body), the shell prints three dots and expects the remaining part.

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] val greeting = "Hello"
[1] greeting + " world!"
res1: String = Hello world!
[2] println(res1)
Hello world!
[3] if (res1.length > 10)
...
```

ki shell results

Check an expression's type

For complex expressions or APIs that you don't know well, the ki shell provides the `:type` (or `:t`) command, which shows the type of an expression:

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] :t sequenceOf("one", "two", "three").associateWith { it.length }
Map<String, Int>
[1]
```

ki shell type

Load code

If the code you need is stored somewhere else, there are two ways to load it and use it in the ki shell:

- Load a source file with the `:load` (or `:l`) command.
- Copy and paste the code snippet in paste mode with the `:paste` (or `:p`) command.

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] :l Desktop/sourceFile.kt
[1] :ls
val variableFromFile: String
fun functionFromFile(): Unit
[2] println(variableFromFile)
Hello from file!
[3] functionFromFile()
Calling function from file
[4]
```

ki shell load file

The `ls` command shows available symbols (variables and functions).

Add external dependencies

Along with the standard library, the ki shell also supports external dependencies. This lets you try out third-party libraries in it without creating a whole project.

To add a third-party library in the ki shell, use the `:dependsOn` command. By default, the ki shell works with Maven Central, but you can use other repositories if you connect them using the `:repository` command:

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] :repository https://maven.pkg.jetbrains.space/public/p/kotlinx-html/maven
[1] :dependsOn org.jetbrains.kotlinx:kotlinx-html-jvm:0.7.3
[2] import kotlinx.html.*
[3] import kotlinx.html.stream.*
[4] val html = createHTML().html {
...body {
...h1 { "Hello" }
...}
...}
[9] html
res5: String = <html>
  <body>
    <h1></h1>
  </body>
</html>

[10]
```

ki shell external dependency

Document Kotlin code: KDoc

The language used to document Kotlin code (the equivalent of Java's Javadoc) is called **KDoc**. In essence, KDoc combines Javadoc's syntax for block tags (extended to support Kotlin's specific constructs) and Markdown for inline markup.

Kotlin's documentation engine: Dokka, understands KDoc and can be used to generate documentation in various formats. For more information, read our Dokka documentation.

KDoc syntax

Just like with Javadoc, KDoc comments start with `/**` and end with `*/`. Every line of the comment may begin with an asterisk, which is not considered part of the contents of the comment.

By convention, the first paragraph of the documentation text (the block of text until the first blank line) is the summary description of the element, and the following text is the detailed description.

Every block tag begins on a new line and starts with the `@` character.

Here's an example of a class documented using KDoc:

```
/**
 * A group of *members*.
 *
 * This class has no useful logic; it's just a documentation example.
 *
 * @param T the type of a member in this group.
 * @property name the name of this group.
 * @constructor Creates an empty group.
 */
class Group<T>(val name: String) {
    /**
     * Adds a [member] to this group.
     * @return the new size of the group.
     */
    fun add(member: T): Int { ... }
}
```

Block tags

KDoc currently supports the following block tags:

@param *name*

Documents a value parameter of a function or a type parameter of a class, property, or function. To better separate the parameter name from the description, if you prefer, you can enclose the name of the parameter in brackets. The following two syntaxes are therefore equivalent:

```
@param name description.
@param[name] description.
```

@return

Documents the return value of a function.

@constructor

Documents the primary constructor of a class.

@receiver

Documents the receiver of an extension function.

@property *name*

Documents the property of a class which has the specified name. This tag can be used for documenting properties declared in the primary constructor, where putting a doc comment directly before the property definition would be awkward.

@throws *class*, @exception *class*

Documents an exception which can be thrown by a method. Since Kotlin does not have checked exceptions, there is also no expectation that all possible exceptions are documented, but you can still use this tag when it provides useful information for users of the class.

@sample *identifier*

Embeds the body of the function with the specified qualified name into the documentation for the current element, in order to show an example of how the element could be used.

@see *identifier*

Adds a link to the specified class or method to the **See also** block of the documentation.

@author

Specifies the author of the element being documented.

@since

Specifies the version of the software in which the element being documented was introduced.

@suppress

Excludes the element from the generated documentation. Can be used for elements which are not part of the official API of a module but still have to be visible externally.

KDoc does not support the `@deprecated` tag. Instead, please use the `@Deprecated` annotation.

Inline markup

For inline markup, KDoc uses the regular Markdown syntax, extended to support a shorthand syntax for linking to other elements in the code.

Links to elements

To link to another element (class, method, property, or parameter), simply put its name in square brackets:

```
Use the method [foo] for this purpose.
```

If you want to specify a custom label for the link, add it in another set of square brackets before the element link:

```
Use [this method][foo] for this purpose.
```

You can also use qualified names in the element links. Note that, unlike Javadoc, qualified names always use the dot character to separate the components, even before a method name:

```
Use [kotlin.reflect.KClass.properties] to enumerate the properties of the class.
```

Names in element links are resolved using the same rules as if the name was used inside the element being documented. In particular, this means that if you have imported a name into the current file, you don't need to fully qualify it when you use it in a KDoc comment.

Note that KDoc does not have any syntax for resolving overloaded members in links. Since Kotlin's documentation generation tool puts the documentation for all overloads of a function on the same page, identifying a specific overloaded function is not required for the link to work.

External links

To add an external link, use the typical Markdown syntax:

```
For more information about KDoc syntax, see [KDoc](<example-URL>).
```

What's next?

Learn how to use Kotlin's documentation generation tool: Dokka.

Kotlin and OSGi

To enable Kotlin OSGi support in your Kotlin project, include `kotlin-osgi-bundle` instead of the regular Kotlin libraries. It is recommended to remove `kotlin-runtime`, `kotlin-stdlib` and `kotlin-reflect` dependencies as `kotlin-osgi-bundle` already contains all of them. You also should pay attention in case when external Kotlin libraries are included. Most regular Kotlin dependencies are not OSGi-ready, so you shouldn't use them and should remove them from your project.

Maven

To include the Kotlin OSGi bundle to a Maven project:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-osgi-bundle</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

To exclude the standard library from external libraries (notice that "star exclusion" works in Maven 3 only):

```
<dependency>
    <groupId>some.group.id</groupId>
    <artifactId>some.library</artifactId>
    <version>some.library.version</version>

    <exclusions>
        <exclusion>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>*</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Gradle

To include `kotlin-osgi-bundle` to a Gradle project:

Kotlin

```
dependencies {
    implementation(kotlin("osgi-bundle"))
}
```

Groovy

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-osgi-bundle:2.4.20"
}
```

To exclude default Kotlin libraries that come as transitive dependencies, you can use the following approach:

Kotlin

```
dependencies {
    implementation("some.group.id:some.library:someversion") {
        exclude(group = "org.jetbrains.kotlin")
    }
}
```

Groovy

```
dependencies {
    implementation('some.group.id:some.library:someversion') {
        exclude group: 'org.jetbrains.kotlin'
    }
}
```

FAQ

Why not just add required manifest options to all Kotlin libraries

Even though it is the most preferred way to provide OSGi support, unfortunately it couldn't be done for now due to so-called "package split" issue that couldn't be easily eliminated and such a big change is not planned for now. There is `Require-Bundle` feature but it is not the best option too and not recommended to use. So it was decided to make a separate artifact for OSGi.

Kotlin and continuous integration with TeamCity

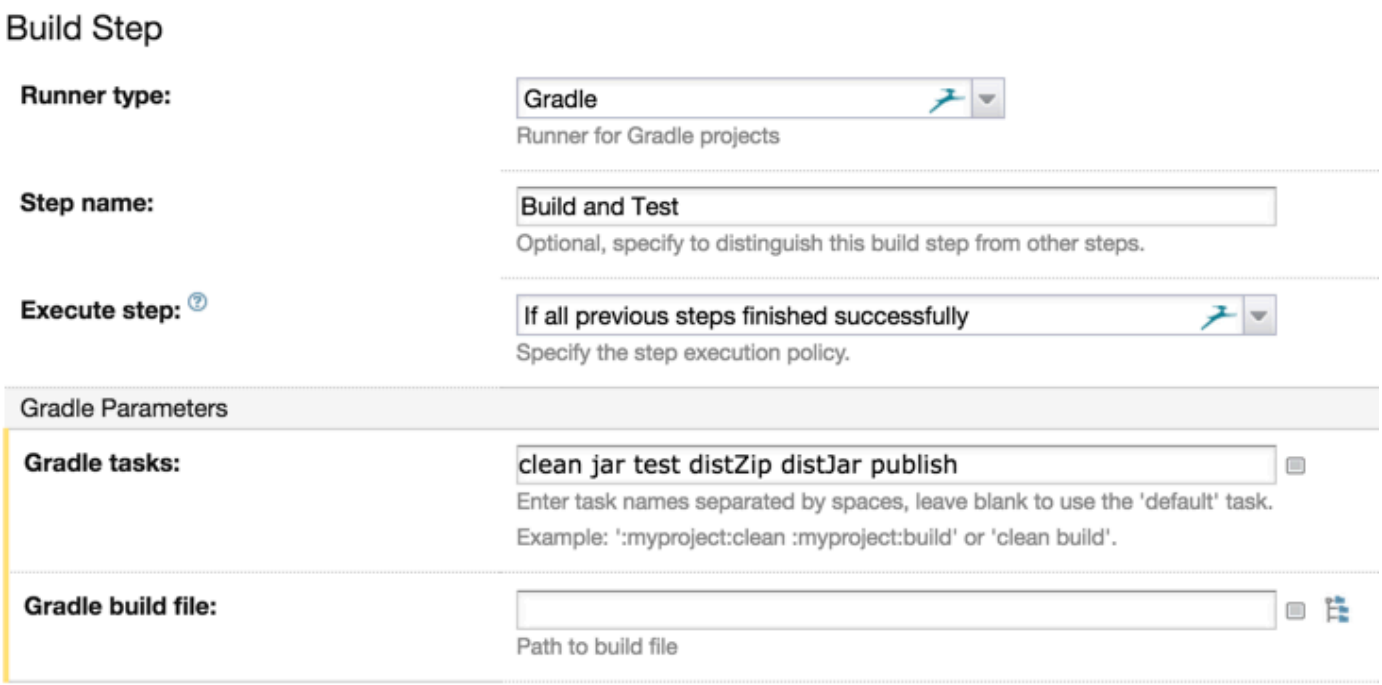
On this page, you'll learn how to set up TeamCity to build your Kotlin project. For more information and basics of TeamCity please check the Documentation page which contains information about installation, basic configuration, etc.

Kotlin works with different build tools, so if you're using a standard tool such as Maven or Gradle, the process for setting up a Kotlin project is no different to any other language or library that integrates with these tools. Where

there are some minor requirements and differences is when using the internal build system of IntelliJ IDEA, which is also supported on TeamCity.

Gradle and Maven

If using Maven or Gradle, the setup process is straightforward. All that is needed is to define the Build Step. For example, if using Gradle, simply define the required parameters such as the Step Name and Gradle tasks that need executing for the Runner Type.



Gradle Build Step

Since all the dependencies required for Kotlin are defined in the Gradle file, nothing else needs to be configured specifically for Kotlin to run correctly.

If using Maven, the same configuration applies. The only difference is that the Runner Type would be Maven.

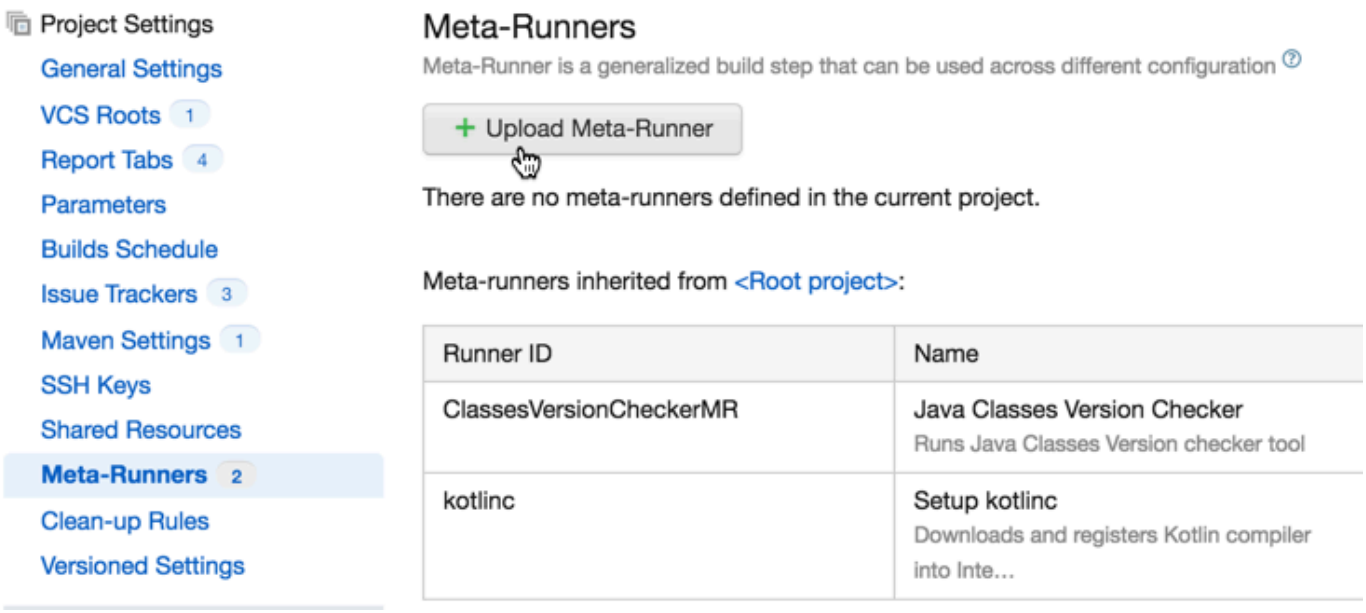
IntelliJ IDEA Build System

If using IntelliJ IDEA build system with TeamCity, make sure that the version of Kotlin being used by IntelliJ IDEA is the same as the one that TeamCity runs. You may need to download the specific version of the Kotlin plugin and install it on TeamCity.

Fortunately, there is a meta-runner already available that takes care of most of the manual work. If not familiar with the concept of TeamCity meta-runners, check the documentation. They are very easy and powerful way to introduce custom Runners without the need to write plugins.

Download and install the meta-runner

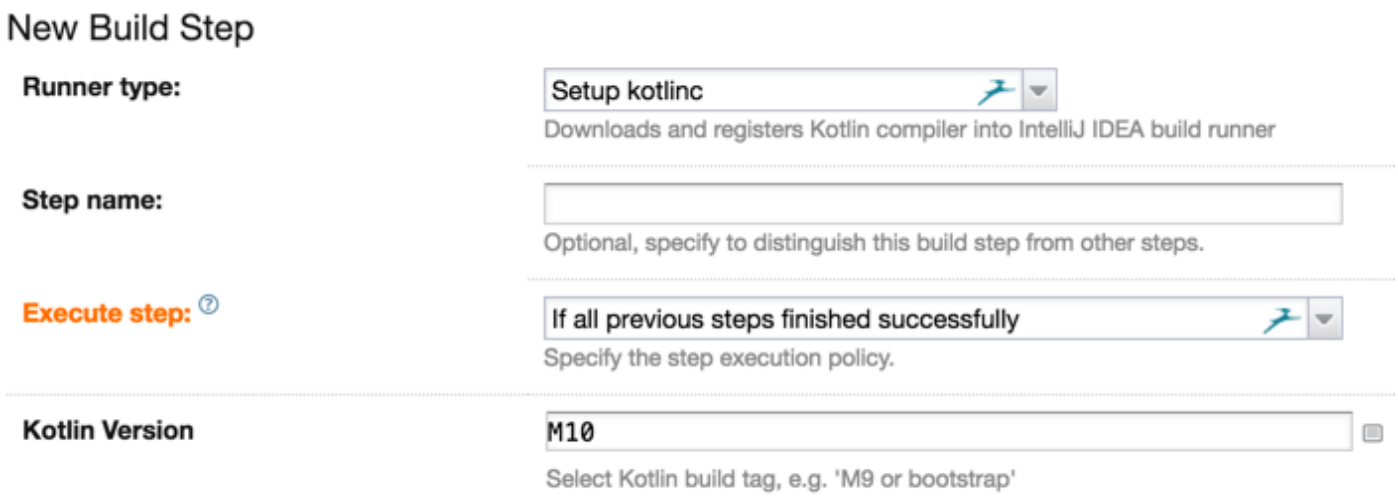
The meta-runner for Kotlin is available on GitHub. Download that meta-runner and import it from the TeamCity user interface



Meta-runner

Setup Kotlin compiler fetching step

Basically, this step is limited to defining the Step Name and the version of Kotlin you need. Tags can be used.

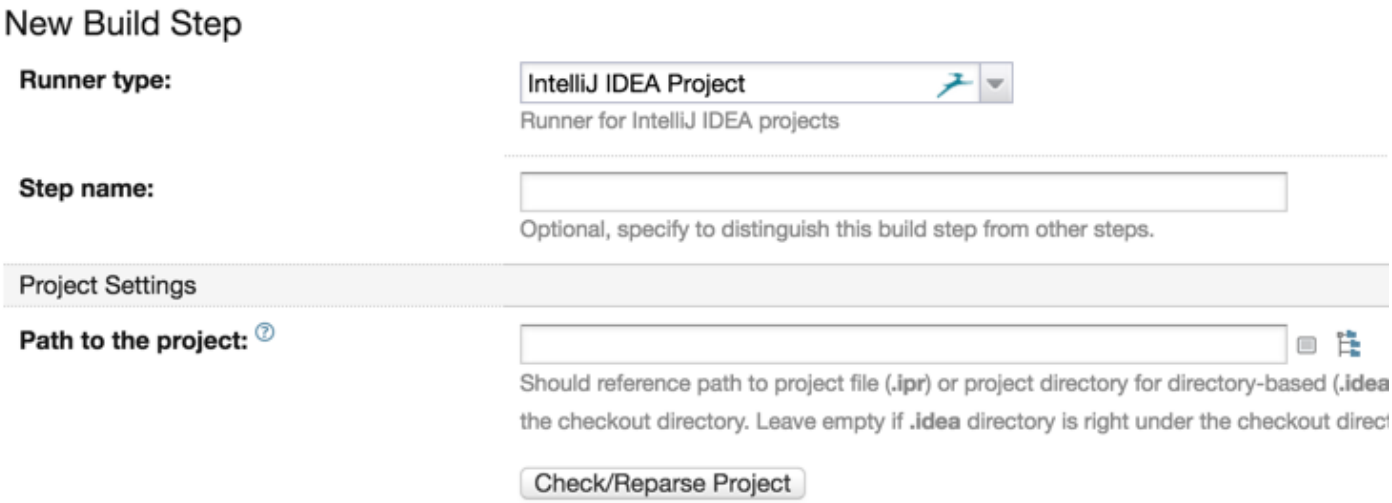


Setup Kotlin Compiler

The runner will set the value for the property `system.path.macro.KOTLIN.BUNDLED` to the correct one based on the path settings from the IntelliJ IDEA project. However, this value needs to be defined in TeamCity (and can be set to any value). Therefore, you need to define it as a system variable.

Setup Kotlin compilation step

The final step is to define the actual compilation of the project, which uses the standard IntelliJ IDEA Runner Type.



IntelliJ IDEA Runner

With that, our project should now build and produce the corresponding artifacts.

Other CI servers

If using a continuous integration tool different to TeamCity, as long as it supports any of the build tools, or calling command line tools, compiling Kotlin and automating things as part of a CI process should be possible.

Comparison to Java

Some Java issues addressed in Kotlin

Kotlin fixes a series of issues that Java suffers from:

- Null references are controlled by the type system.
- No raw types
- Arrays in Kotlin are invariant
- Kotlin has proper function types, as opposed to Java's SAM-conversions
- Use-site variance without wildcards
- Kotlin does not have checked exceptions
- Separate interfaces for read-only and mutable collections

What Java has that Kotlin does not

- Checked exceptions
- Primitive types that are not classes. The byte-code uses primitives where possible, but they are not explicitly available.
- Static members are replaced with companion objects, top-level functions, extension functions, or @JvmStatic.
- Wildcard-types are replaced with declaration-site variance and type projections.

- Ternary-operator `a ? b : c` is replaced with if expression.
- Records
- package-private visibility modifier

> Kotlin doesn't have pattern matching, but smart casts in Kotlin provide similar functionality to pattern Matching in Java.
>
> Learn more in this video on the official Kotlin by JetBrains channel.

What Kotlin has that Java does not

- Lambda expressions + Inline functions = performant custom control structures
- Extension functions
- Null-safety
- String templates
- Properties
- Primary constructors
- First-class delegation
- Type inference for variable and property types (**Java 10**: Local-Variable Type Inference)
- Singletons
- Declaration-site variance & Type projections
- Range expressions
- Operator overloading
- Companion objects
- Data classes
- Coroutines
- Top-level functions
- Parameters with default values
- Named parameters
- Infix functions
- Expect and actual declarations
- Explicit API mode and better control of API surface

> Java doesn't have smart casts, but pattern matching provides similar functionality to smart casts in Kotlin.
>
> Learn more in this video on the official Kotlin by JetBrains channel.

What's next?

Learn how to:

- Perform typical tasks with strings in Java and Kotlin.
- Perform typical tasks with collections in Java and Kotlin.
- Handle nullability in Java and Kotlin.

Calling Java from Kotlin

Kotlin is designed with Java interoperability in mind. Existing Java code can be called from Kotlin in a natural way, and Kotlin code can be used from Java rather smoothly as well. In this section, we describe some details about calling Java code from Kotlin.

Pretty much all Java code can be used without any issues:

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 'for'-loops work for Java collections:
    for (item in source) {
        list.add(item)
    }
    // Operator conventions work as well:
    for (i in 0..source.size - 1) {
        list[i] = source[i] // get and set are called
    }
}
```

Getters and setters

Methods that follow the Java conventions for getters and setters (no-argument methods with names starting with `get` and single-argument methods with names starting with `set`) are represented as properties in Kotlin. Such properties are also called *synthetic properties*. `Boolean` accessor methods (where the name of the getter starts with `is` and the name of the setter starts with `set`) are represented as properties which have the same name as the getter method.

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) { // call getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY // call setFirstDayOfWeek()
    }
    if (!calendar.isLenient) { // call isLenient()
        calendar.isLenient = true // call setLenient()
    }
}
```

`calendar.firstDayOfWeek` above is an example of a synthetic property.

Note that if the Java class only has a setter, it isn't visible as a property in Kotlin because Kotlin doesn't support set-only properties.

Java synthetic property references

This feature is Experimental. It may be dropped or changed at any time. We recommend that you use it only for evaluation purposes.

Starting from Kotlin 1.8.20, you can create references to Java synthetic properties. Consider the following Java code:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Kotlin has always allowed you to write `person.age`, where `age` is a synthetic property. Now, you can also create references to `Person::age` and `person::age`. The same applies for `name`, as well.

```
val persons = listOf(Person("Jack", 11), Person("Sofie", 12), Person("Peter", 11))
    persons
         // Call a reference to Java synthetic property:
        .sortedBy(Person::age)
         // Call Java getter via the Kotlin property syntax:
        .forEach { person -> println(person.name) }
```

How to enable Java synthetic property references

To enable this feature, set the `-language-version 2.1` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)`:

Kotlin

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_1
            )
    }
```

Groovy

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion
            = org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_1
}
```

Prior to Kotlin 1.9.0, to enable this feature you had to set the `-language-version 1.9` compiler option.

Methods returning void

If a Java method returns `void`, it will return `Unit` when called from Kotlin. If by any chance someone uses that return value, it will be assigned at the call site by the Kotlin compiler since the value itself is known in advance (being `Unit`).

Escaping for Java identifiers that are keywords in Kotlin

Some of the Kotlin keywords are valid identifiers in Java: `in`, `object`, `is`, and others. If a Java library uses a Kotlin keyword for a method, you can still call the method escaping it with the backtick (`) character:

```
foo.`is`(bar)
```

Null-safety and platform types

Any reference in Java may be `null`, which makes Kotlin's requirements of strict null-safety impractical for objects coming from Java. Types of Java declarations are treated in Kotlin as non-denotable and called *platform types*. You can't write the non-denotable types in the code explicitly. Therefore, when a platform value is assigned to a Kotlin variable, you can:

- Rely on the type inference. In this case, the variable has an inferred platform type.
- Choose the type you expect. Kotlin allows both nullable and non-nullable types.

Null-checks are relaxed for such types, so that safety guarantees for them are the same as in Java (see more below). Consider the following examples:

```
val list = ArrayList<String>() // non-null (constructor result)
list.add("Item")
val size = list.size // non-null (primitive int)
val item = list[0] // platform type inferred (ordinary Java object)
```

When you call methods on variables of platform types, Kotlin does not issue nullability errors at compile time, but the call may fail at runtime, because of a null-pointer exception or an assertion that Kotlin generates to prevent nulls from propagating:

```
item.substring(1) // allowed, throws an exception if item == null
```

You can assign a value of a platform type to a variable of both nullable and non-nullable Kotlin types. However, if you assign such a value to a variable of a non-nullable type and the value is actually `null` at runtime, Kotlin throws a `NullPointerException`. To avoid this, add explicit nullability to your Kotlin code:

```
val nullable: String? = item // allowed, always works
val notNull: String = item // allowed, may fail at runtime
```

If you choose a non-nullable type, the compiler will emit an assertion upon assignment. This prevents Kotlin's non-nullable variables from holding nulls. Assertions are also emitted when you pass platform values to Kotlin functions expecting non-null values and in other cases. Overall, the compiler does its best to prevent nulls from propagating far through the program, although sometimes this is impossible to eliminate entirely, because of generics.

Notation for platform types

As mentioned in the previous section, platform types can't be mentioned explicitly in the program, so there's no syntax for them in the language. Nevertheless, the compiler and IDE need to display them sometimes (for example, in error messages or parameter info), so there is a mnemonic notation for them:

- `T!` means "`T` or `T?`"
- `(Mutable)Collection<T>!` means "Java collection of `T` may be mutable or not, may be nullable or not"
- `Array<(out) T>!` means "Java array of `T` (or a subtype of `T`), nullable or not"

When you see this notation in an error message or IDE tooltip, add an explicit type annotation to your Kotlin variable to restore null-safety checks, or eliminate platform types at the source using nullability annotations.

Nullability annotations

Java types that have nullability annotations are represented not as platform types, but as actual nullable or non-nullable Kotlin types. The compiler supports several flavors of nullability annotations, including:

- JetBrains (`@Nullable` and `@NotNull` from the `org.jetbrains.annotations` package)
- JSpecify (`org.jspecify.annotations`)
- Android (`com.android.annotations` and `android.support.annotations`)
- JSR-305 (`javax.annotation`)
- FindBugs (`edu.umd.cs.findbugs.annotations`)
- Eclipse (`org.eclipse.jdt.annotation`)
- Lombok (`lombok.NonNull`)
- RxJava 3 (`io.reactivex.rxjava3.annotations`)
- Vert.x (`io.vertx.codegen.annotations`)

You can instruct the compiler to report nullability mismatches for specific nullability annotations with the following compiler option:

```
-Xnullability-annotations=@<package-name>:<report-level>
```

Specify the package name for the fully qualified nullability annotations and one of these report levels:

- `ignore` to ignore nullability mismatches

- `warn` to report warnings
- `strict` to report errors.

> JSpecify is the only supported flavor that uses `strict` report level by default. Use it to report errors on nullability annotations without additional configuration.

See the full list of supported nullability annotations in the Kotlin compiler source code.

Mutability annotations

You can annotate a Java declaration with a mutability annotation to specify whether a returned collection is read-only or mutable in Kotlin. If you assign the value to a collection type with different mutability, the compiler reports a type mismatch. The diagnostic severity depends on the specific mutability annotation.

The compiler supports several mutability annotations, including:

- `kotlin.annotations.jvm.ReadOnly`
- `kotlin.annotations.jvm.Mutable`
- `org.jetbrains.annotations.Unmodifiable`
- `org.jetbrains.annotations.UnmodifiableView`

See the full list of supported mutability annotations in the Kotlin compiler source code.

Annotating type arguments and type parameters

You can annotate the type arguments and type parameters of generic types to provide nullability information for them as well.

> All examples in the section use JetBrains nullability annotations from the `org.jetbrains.annotations` package.

Type arguments

Consider these annotations on a Java declaration:

```
@NotNull
Set<@NotNull String> toSet(@NotNull Collection<@NotNull String> elements) { ... }
```

They result in the following signature in Kotlin:

```
fun toSet(elements: (Mutable)Collection<String>) : (Mutable)Set<String> { ... }
```

When the `@NotNull` annotation is missing from a type argument, you get a platform type instead:

```
fun toSet(elements: (Mutable)Collection<String!>) : (Mutable)Set<String!> { ... }
```

Kotlin also takes into account nullability annotations on type arguments of base classes and interfaces. For example, there are two Java classes with the signatures provided below:

```
public class Base<T> {}
```

```
public class Derived extends Base<@Nullable String> {}
```

In the Kotlin code, passing the instance of `Derived` where the `Base<String>` is assumed produces the warning.

```
fun takeBaseOfNotNullStrings(x: Base<String>) {}

fun main() {
    takeBaseOfNotNullStrings(Derived()) // warning: nullability mismatch
}
```

The upper bound of `Derived` is set to `Base<String?>`, which is different from `Base<String>`.

Learn more about Java generics in Kotlin.

Type parameters

By default, the nullability of plain type parameters in both Kotlin and Java is undefined. In Java, you can specify it using nullability annotations. Let's annotate the type parameter of the `Base` class:

```
public class Base<@NotNull T> {}
```

When inheriting from `Base`, Kotlin expects a non-nullable type argument or type parameter. Thus, the following Kotlin code produces a warning:

```
class Derived<K> : Base<K> {} // warning: K has undefined nullability
```

You can fix it by specifying the upper bound `K : Any`.

Kotlin also supports nullability annotations on the bounds of Java type parameters. Let's add bounds to `Base`:

```
public class BaseWithBound<T extends @NotNull Number> {}
```

Kotlin translates this just as follows:

```
class BaseWithBound<T : Number> {}
```

So passing nullable type as a type argument or type parameter produces a warning.

Annotating type arguments and type parameters works with the Java 8 target or higher. The feature requires that the nullability annotations support the `TYPE_USE` target (`org.jetbrains.annotations` supports this in version 15 and above).

If a nullability annotation supports other targets that are applicable to a type in addition to the `TYPE_USE` target, `TYPE_USE` takes priority. For example, if `@Nullable` has both `TYPE_USE` and `METHOD` targets, the Java method signature `@Nullable String[] f()` becomes `fun f(): Array<String?>!` in Kotlin.

JSpecify support

Kotlin supports the JSpecify nullability annotations, which provide a unified set of annotations for Java nullability. JSpecify allows you to provide detailed nullability information for Java declarations, helping Kotlin maintain null-safety when working with Java code.

Kotlin supports the following annotations in the `org.jspecify.annotations` package:

- `@Nullable` marks a type as nullable.
- `@NonNull` marks a type as non-nullable.
- `@NullMarked` marks all types within a scope, for example a class or package, as non-nullable by default unless annotated otherwise.

  This annotation doesn't apply to local variables and type variables (generics). Type variables remain "null-agnostic" until a specific nullable or non-nullable type is provided.
- `@NullUnmarked` reverses the effect of `@NullMarked`, making all types within the scope as platform types.

Consider the following Java class with JSpecify annotations:

```
// Java
import org.jspecify.annotations.*;

@NullMarked
public class InventoryService {
    public String notNull() { return ""; }
    public @Nullable String nullable() { return null; }
}
```

In Kotlin, these are treated as regular nullable and non-nullable types rather than platform types:

```
// Kotlin
fun test(inventory: InventoryService) {
   inventory.notNull().length // OK
   inventory.nullable().length // Error: only safe (?.) or non-null asserted (!!) calls are allowed
}
```

By default, the Kotlin compiler reports nullability mismatches for JSpecify annotations as errors. You can customize the severity of JSpecify nullability diagnostics using the following compiler option:

```
-Xjspecify-annotations=<report-level>
```

Available report levels are:

Level	Description
`strict`	Reports errors for nullability mismatches (default).
`warn`	Reports warnings.
`ignore`	Ignores nullability mismatches.

For more information on JSpecify annotations, see the JSpecify user guide.

JSR-305 support

The `@Nonnull` annotation defined in JSR-305 is supported for denoting nullability of Java types.

If the `@Nonnull(when = ...)` value is `When.ALWAYS`, the annotated type is treated as non-nullable; `When.MAYBE` and `When.NEVER` denote a nullable type; and `When.UNKNOWN` forces the type to be platform one.

A library can be compiled against the JSR-305 annotations, but there's no need to make the annotations artifact (e.g. `jsr305.jar`) a compile dependency for the library consumers. The Kotlin compiler can read the JSR-305 annotations from a library without the annotations present on the classpath.

Custom nullability qualifiers (KEEP-79) are also supported (see below).

Type qualifier nicknames

If an annotation type is annotated with both `@TypeQualifierNickname` and JSR-305 `@Nonnull` (or its another nickname, such as `@CheckForNull`), then the annotation type is itself used for retrieving precise nullability and has the same meaning as that nullability annotation:

```
@TypeQualifierNickname
@Nonnull(when = When.ALWAYS)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNonnull {
}

@TypeQualifierNickname
@CheckForNull // a nickname to another type qualifier nickname
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNullable {
}

interface A {
    @MyNullable String foo(@MyNonnull String x);
    // in Kotlin (strict mode): `fun foo(x: String): String?`

    String bar(List<@MyNonnull String> x);
    // in Kotlin (strict mode): `fun bar(x: List<String>!): String!`
}
```

Type qualifier defaults

`@TypeQualifierDefault` allows introducing annotations that, when being applied, define the default nullability within the scope of the annotated element.

Such annotation type should itself be annotated with both `@Nonnull` (or its nickname) and `@TypeQualifierDefault(...)` with one or more `ElementType` values:

- `ElementType.METHOD` for return types of methods
- `ElementType.PARAMETER` for value parameters
- `ElementType.FIELD` for fields
- `ElementType.TYPE_USE` for any type including type arguments, upper bounds of type parameters and wildcard types

The default nullability is used when a type itself is not annotated by a nullability annotation, and the default is determined by the innermost enclosing element annotated with a type qualifier default annotation with the `ElementType` matching the type usage.

```java
@Nonnull
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
public @interface NonNullApi {
}

@Nonnull(when = When.MAYBE)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE_USE})
public @interface NullableApi {
}

@NullableApi
interface A {
    String foo(String x); // fun foo(x: String?): String?

    @NotNullApi // overriding default from the interface
    String bar(String x, @Nullable String y); // fun bar(x: String, y: String?): String

    // The List<String> type argument is seen as nullable because of `@NullableApi`
    // having the `TYPE_USE` element type:
    String baz(List<String> x); // fun baz(List<String?>?): String?

    // The type of `x` parameter remains platform because there's an explicit
    // UNKNOWN-marked nullability annotation:
    String qux(@Nonnull(when = When.UNKNOWN) String x); // fun baz(x: String!): String?
}
```

The types in this example only take place with the strict mode enabled; otherwise, the platform types remain. See the `@UnderMigration` annotation and Compiler configuration sections.

Package-level default nullability is also supported:

```java
// FILE: test/package-info.java
@NonNullApi // declaring all types in package 'test' as non-nullable by default
package test;
```

@UnderMigration annotation

The `@UnderMigration` annotation (provided in a separate artifact `kotlin-annotations-jvm`) can be used by library maintainers to define the migration status for the nullability type qualifiers.

The status value in `@UnderMigration(status = ...)` specifies how the compiler treats inappropriate usages of the annotated types in Kotlin (e.g. using a `@MyNullable`-annotated type value as non-null):

- `MigrationStatus.STRICT` makes annotation work as any plain nullability annotation, i.e. report errors for the inappropriate usages and affect the types in the annotated declarations as they are seen in Kotlin
- `MigrationStatus.WARN`: the inappropriate usages are reported as compilation warnings instead of errors, but the types in the annotated declarations remain platform
- `MigrationStatus.IGNORE` makes the compiler ignore the nullability annotation completely

A library maintainer can add `@UnderMigration` status to both type qualifier nicknames and type qualifier defaults:

```
@Nonnull(when = When.ALWAYS)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
@UnderMigration(status = MigrationStatus.WARN)
public @interface NonNullApi {
}

// The types in the class are non-nullable, but only warnings are reported
// because `@NonNullApi` is annotated `@UnderMigration(status = MigrationStatus.WARN)`
@NonNullApi
public class Test {}
```

The migration status of a nullability annotation is not inherited by its type qualifier nicknames but is applied to its usages in default type qualifiers.

If a default type qualifier uses a type qualifier nickname and they are both `@UnderMigration`, the status from the default type qualifier is used.

Compiler configuration

The JSR-305 checks can be configured by adding the `-Xjsr305` compiler flag with the following options (and their combination):

- `-Xjsr305={strict|warn|ignore}` to set up the behavior for non-`@UnderMigration` annotations. Custom nullability qualifiers, especially `@TypeQualifierDefault`, are already spread among many well-known libraries, and users may need to migrate smoothly when updating to the Kotlin version containing JSR-305 support. Since Kotlin 1.1.60, this flag only affects non-`@UnderMigration` annotations.
- `-Xjsr305=under-migration:{strict|warn|ignore}` to override the behavior for the `@UnderMigration` annotations. Users may have different view on the migration status for the libraries: they may want to have errors while the official migration status is `WARN`, or vice versa, they may wish to postpone errors reporting for some until they complete their migration.
- `-Xjsr305=@<fq.name>:{strict|warn|ignore}` to override the behavior for a single annotation, where `<fq.name>` is the fully qualified class name of the annotation. May appear several times for different annotations. This is useful for managing the migration state for a particular library.

The `strict`, `warn` and `ignore` values have the same meaning as those of `MigrationStatus`, and only the `strict` mode affects the types in the annotated declarations as they are seen in Kotlin.

Note: the built-in JSR-305 annotations `@Nonnull`, `@Nullable` and `@CheckForNull` are always enabled and affect the types of the annotated declarations in Kotlin, regardless of compiler configuration with the `-Xjsr305` flag.

For example, adding `-Xjsr305=ignore -Xjsr305=under-migration:ignore -Xjsr305=@org.library.MyNullable:warn` to the compiler arguments makes the compiler generate warnings for inappropriate usages of types annotated by `@org.library.MyNullable` and ignore all other JSR-305 annotations.

The default behavior is the same to `-Xjsr305=warn`. The `strict` value should be considered experimental (more checks may be added to it in the future).

Mapped types

Kotlin treats some Java types specifically. Such types are not loaded from Java "as is", but are *mapped* to corresponding Kotlin types. The mapping only matters at compile time, the runtime representation remains unchanged. Java's primitive types are mapped to corresponding Kotlin types (keeping platform types in mind):

Java type	Kotlin type
`byte`	`kotlin.Byte`
`short`	`kotlin.Short`
`int`	`kotlin.Int`
`long`	`kotlin.Long`
`char`	`kotlin.Char`
`float`	`kotlin.Float`
`double`	`kotlin.Double`
`boolean`	`kotlin.Boolean`

Some non-primitive built-in classes are also mapped:

Java type	Kotlin type
`java.lang.Object`	`kotlin.Any!`
`java.lang.Cloneable`	`kotlin.Cloneable!`
`java.lang.Comparable`	`kotlin.Comparable!`
`java.lang.Enum`	`kotlin.Enum!`
`java.lang.annotation.Annotation`	`kotlin.Annotation!`
`java.lang.CharSequence`	`kotlin.CharSequence!`
`java.lang.String`	`kotlin.String!`
`java.lang.Number`	`kotlin.Number!`

Java type	Kotlin type
`java.lang.Throwable`	`kotlin.Throwable!`

Java's boxed primitive types are mapped to nullable Kotlin types:

Java type	Kotlin type
`java.lang.Byte`	`kotlin.Byte?`
`java.lang.Short`	`kotlin.Short?`
`java.lang.Integer`	`kotlin.Int?`
`java.lang.Long`	`kotlin.Long?`
`java.lang.Character`	`kotlin.Char?`
`java.lang.Float`	`kotlin.Float?`
`java.lang.Double`	`kotlin.Double?`
`java.lang.Boolean`	`kotlin.Boolean?`

Note that a boxed primitive type used as a type parameter is mapped to a platform type: for example, `List<java.lang.Integer>` becomes a `List<Int!>` in Kotlin.

Collection types may be read-only or mutable in Kotlin, so Java's collections are mapped as follows (all Kotlin types in this table reside in the package `kotlin.collections`):

Java type	Kotlin read-only type	Kotlin mutable type	Loaded platform type
`Iterator<T>`	`Iterator<T>`	`MutableIterator<T>`	`(Mutable)Iterator<T>!`
`Iterable<T>`	`Iterable<T>`	`MutableIterable<T>`	`(Mutable)Iterable<T>!`
`Collection<T>`	`Collection<T>`	`MutableCollection<T>`	`(Mutable)Collection<T>!`
`Set<T>`	`Set<T>`	`MutableSet<T>`	`(Mutable)Set<T>!`
`List<T>`	`List<T>`	`MutableList<T>`	`(Mutable)List<T>!`

Java type	Kotlin read-only type	Kotlin mutable type	Loaded platform type
`ListIterator<T>`	`ListIterator<T>`	`MutableListIterator<T>`	`(Mutable)ListIterator<T>!`
`Map<K, V>`	`Map<K, V>`	`MutableMap<K, V>`	`(Mutable)Map<K, V>!`
`Map.Entry<K, V>`	`Map.Entry<K, V>`	`MutableMap.MutableEntry<K, V>`	`(Mutable)Map.(Mutable)Entry<K, V>!`

Java's arrays are mapped as mentioned below:

Java type	Kotlin type
`int[]`	`kotlin.IntArray!`
`String[]`	`kotlin.Array<(out) String!>!`

The static members of these Java types are not directly accessible on the companion objects of the Kotlin types. To call them, use the full qualified names of the Java types, e.g. `java.lang.Integer.toHexString(foo)`.

Java generics in Kotlin

Kotlin's generics are a little different from Java's (see Generics). When importing Java types to Kotlin, the following conversions are done:

- Java's wildcards are converted into type projections:
  - `Foo<? extends Bar>` becomes `Foo<out Bar!>!`
  - `Foo<? super Bar>` becomes `Foo<in Bar!>!`
- Java's raw types are converted into star projections:
  - `List` becomes `List<*>!` that is `List<out Any?>!`

Like Java's, Kotlin's generics are not retained at runtime: objects do not carry information about actual type arguments passed to their constructors. For example, `ArrayList<Integer>()` is indistinguishable from `ArrayList<Character>()`. This makes it impossible to perform `is`-checks that take generics into account. Kotlin only allows `is`-checks for star-projected generic types:

```
if (a is List<Int>) // Error: cannot check if it is really a List of Ints
// but
if (a is List<*>) // OK: no guarantees about the contents of the list
```

Java arrays

Arrays in Kotlin are invariant, unlike Java. This means that Kotlin won't let you assign an `Array<String>` to an `Array<Any>`, which prevents a possible runtime failure. Passing an array of a subclass as an array of superclass to a Kotlin method is also prohibited, but for Java methods this is allowed through platform types of the form `Array<(out) String>!`.

Arrays are used with primitive datatypes on the Java platform to avoid the cost of boxing/unboxing operations. As Kotlin hides those implementation details, a workaround is required to interface with Java code. There are specialized classes for every type of primitive array (`IntArray`, `DoubleArray`, `CharArray`, and so on) to handle this case. They are not related to the `Array` class and are compiled down to Java's primitive arrays for maximum performance.

Suppose there is a Java method that accepts an int array of indices:

```
public class JavaArrayExample {
    public void removeIndices(int[] indices) {
        // code here...
    }
}
```

To pass an array of primitive values, you can do the following in Kotlin:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndices(array)  // passes int[] to method
```

When compiling to the JVM bytecode, the compiler optimizes access to arrays so that there's no overhead introduced:

```
val array = arrayOf(1, 2, 3, 4)
array[1] = array[1] * 2 // no actual calls to get() and set() generated
for (x in array) { // no iterator created
    print(x)
}
```

Even when you navigate with an index, it does not introduce any overhead:

```
for (i in array.indices) { // no iterator created
    array[i] += 2
}
```

Finally, `in`-checks have no overhead either:

```
if (i in array.indices) { // same as (i ≥ 0 && i < array.size)
    print(array[i])
}
```

Java varargs

Java classes sometimes use a method declaration for the indices with a variable number of arguments (varargs):

```
public class JavaArrayExample {

    public void removeIndicesVarArg(int... indices) {
        // code here...
    }
}
```

In that case, you need to use the spread operator `*` to pass the `IntArray`:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

Operators

Since Java has no way of marking methods for which it makes sense to use the operator syntax, Kotlin allows using any Java methods with the right name and signature as operator overloads and other conventions (`invoke()` etc.) Calling Java methods using the infix call syntax is not allowed.

Checked exceptions

In Kotlin, all exceptions are unchecked, meaning that the compiler does not force you to catch any of them. So, when you call a Java method that declares a checked exception, Kotlin does not force you to do anything:

```
fun render(list: List<*>, to: Appendable) {
    for (item in list) {
        to.append(item.toString()) // Java would require us to catch IOException here
    }
}
```

Object methods

When Java types are imported into Kotlin, all the references of the type `java.lang.Object` are turned into `Any`. Since `Any` is not platform-specific, it only declares `toString()`, `hashCode()` and `equals()` as its members, so to make other members of `java.lang.Object` available, Kotlin uses extension functions.

`wait()` and `notify()`

Methods `wait()` and `notify()` are not available on references of type `Any`. Their usage is generally discouraged in favor of `java.util.concurrent`.

If you have to call these methods, access them through Java objects and suppress the `PLATFORM_CLASS_MAPPED_TO_KOTLIN` warning:

```
import java.util.LinkedList

class SimpleBlockingQueue<T>(private val capacity: Int) {
    private val queue = LinkedList<T>()

    // java.lang.Object is used specifically to access wait() and notify()
    // In Kotlin, the standard 'Any' type does not expose these methods.
    @Suppress("PLATFORM_CLASS_MAPPED_TO_KOTLIN")
    private val lock = Object()

    fun put(item: T) {
        synchronized(lock) {
            while (queue.size >= capacity) {
                lock.wait()
            }
            queue.add(item)
            println("Produced: $item")

            lock.notifyAll()
        }
    }

    fun take(): T {
        synchronized(lock) {
            while (queue.isEmpty()) {
                lock.wait()
            }
            val item = queue.removeFirst()
            println("Consumed: $item")

            lock.notifyAll()
            return item
        }
    }
}
```

Or explicitly cast to `java.lang.Object` and suppress the `PLATFORM_CLASS_MAPPED_TO_KOTLIN` warning:

```
@Suppress("PLATFORM_CLASS_MAPPED_TO_KOTLIN")
(foo as java.lang.Object).wait()
```

`getClass()`

To retrieve the Java class of an object, use the `java` extension property on a class reference:

```
val fooClass = foo::class.java
```

The code above uses a bound class reference. You can also use the `javaClass` extension property:

```
val fooClass = foo.javaClass
```

`clone()`

To override `clone()`, your class needs to extend `kotlin.Cloneable`:

```
class Example : Cloneable {
    override fun clone(): Any { ... }
}
```

Don't forget about Effective Java, 3rd Edition, Item 13: *Override clone judiciously*.

`finalize()`

To override `finalize()`, all you need to do is simply declare it, without using the `override` keyword:

```
class C {
    protected fun finalize() {
        // finalization logic
    }
}
```

According to Java's rules, `finalize()` must not be `private`.

Inheritance from Java classes

At most one Java class (and as many Java interfaces as you like) can be a supertype for a class in Kotlin.

Accessing static members

Static members of Java classes form "companion objects" for these classes. You can't pass such a "companion object" around as a value but can access the members explicitly, for example:

```
if (Character.isLetter(a)) { ... }
```

To access static members of a Java type that is mapped to a Kotlin type, use the full qualified name of the Java type: `java.lang.Integer.bitCount(foo)`.

Java reflection

Java reflection works on Kotlin classes and vice versa. As mentioned above, you can use `instance::class.java`, `ClassName::class.java` or `instance.javaClass` to enter Java reflection through `java.lang.Class`. Do not use `ClassName.javaClass` for this purpose because it refers to `ClassName`'s companion object class, which is the same as `ClassName.Companion::class.java` and not `ClassName::class.java`.

For each primitive type, there are two different Java classes, and Kotlin provides ways to get both. For example, `Int::class.java` will return the class instance representing the primitive type itself, corresponding to `Integer.TYPE` in Java. To get the class of the corresponding wrapper type, use `Int::class.javaObjectType`, which is equivalent of Java's `Integer.class`.

Other supported cases include acquiring a Java getter/setter method or a backing field for a Kotlin property, a `KProperty` for a Java field, a Java method or constructor for a `KFunction` and vice versa.

SAM conversions

Kotlin supports SAM conversions for both Java and Kotlin interfaces. This support for Java means that Kotlin function literals can be automatically converted into implementations of Java interfaces with a single non-default method, as long as the parameter types of the interface method match the parameter types of the Kotlin function.

You can use this for creating instances of SAM interfaces:

```
val runnable = Runnable { println("This runs in a runnable") }
```

...and in method calls:

```
val executor = ThreadPoolExecutor()
// Java signature: void execute(Runnable command)
executor.execute { println("This runs in a thread pool") }
```

If the Java class has multiple methods taking functional interfaces, you can choose the one you need to call by using an adapter function that converts a lambda to a specific SAM type. Those adapter functions are also generated by the compiler when needed:

```
executor.execute(Runnable { println("This runs in a thread pool") })
```

SAM conversions only work for interfaces, not for abstract classes, even if those also have just a single abstract method.

Using JNI with Kotlin

To declare a function that is implemented in native (C or C++) code, you need to mark it with the `external` modifier:

```
external fun foo(x: Int): Double
```

The rest of the procedure works in exactly the same way as in Java.

You can also mark property getters and setters as `external`:

```
var myProperty: String
    external get
    external set
```

Behind the scenes, this will create two functions `getMyProperty` and `setMyProperty`, both marked as `external`.

Using Lombok-generated declarations in Kotlin

You can use Java's Lombok-generated declarations in Kotlin code. If you need to generate and use these declarations in the same mixed Java/Kotlin module, you can learn how to do this on the Lombok compiler plugin's page. If you call such declarations from another module, then you don't need to use this plugin to compile that module.

Calling Kotlin from Java

Kotlin code can be easily called from Java. For example, instances of a Kotlin class can be seamlessly created and operated in Java methods. However, there are certain differences between Java and Kotlin that require attention when integrating Kotlin code into Java. On this page, we'll describe the ways to tailor the interop of your Kotlin code with its Java clients.

Properties

A Kotlin property is compiled to the following Java elements:

- a getter method, with the name calculated by prepending the `get` prefix.
- a setter method, with the name calculated by prepending the `set` prefix (only for `var` properties).
- a private field, with the same name as the property name (only for properties with backing fields).

For example, `var firstName: String` compiles to the following Java declarations:

```
private String firstName;

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

If the name of the property starts with `is`, a different name mapping rule is used: the name of the getter is the same as the property name, and the name of the setter is obtained by replacing `is` with `set`. For example, for a property `isOpen`, the getter is called `isOpen()` and the setter is called `setOpen()`. This rule applies for properties of any type, not just `Boolean`.

Package-level functions

All the functions and properties declared in a file `app.kt` inside a package `org.example`, including extension functions, are compiled into static methods of a Java class named `org.example.AppKt`.

```
// app.kt
package org.example

class Util

fun getTime() { /*...*/ }
```

```
// Java
new org.example.Util();
org.example.AppKt.getTime();
```

To set a custom name to the generated Java class, use the `@JvmName` annotation:

```
@file:JvmName("DemoUtils")

package org.example

class Util

fun getTime() { /*...*/ }
```

```
// Java
new org.example.Util();
org.example.DemoUtils.getTime();
```

Having multiple files with the same generated Java class name (the same package and the same name or the same `@JvmName` annotation) is normally an error. However, the compiler can generate a single Java facade class which has the specified name and contains all the declarations from all the files which have that name. To enable the generation of such a facade, use the `@JvmMultifileClass` annotation in all such files.

```
// oldutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package org.example

fun getTime() { /*...*/ }
```

```
// newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package org.example

fun getDate() { /*...*/ }
```

```
// Java
org.example.Utils.getTime();
org.example.Utils.getDate();
```

Instance fields

If you need to expose a Kotlin property as a field in Java, annotate it with the `@JvmField` annotation. The field has the same visibility as the underlying property. You can annotate a property with `@JvmField` if it:

- has a backing field
- is not private
- does not have `open`, `override` or `const` modifiers
- is not a delegated property

```
class User(id: String) {
    @JvmField val ID = id
}
```

```
// Java
class JavaClient {
    public String getID(User user) {
        return user.ID;
    }
}
```

Late-Initialized properties are also exposed as fields. The visibility of the field is the same as the visibility of the `lateinit` property setter.

Static fields

Kotlin properties declared in a named object or a companion object have static backing fields either in that named object or in the class containing the companion object.

Usually these fields are private, but they can be exposed in one of the following ways:

- `@JvmField` annotation
- `lateinit` modifier
- `const` modifier

Annotating such a property with `@JvmField` makes it a static field with the same visibility as the property itself.

```
class Key(val value: Int) {
    companion object {
        @JvmField
        val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value }
    }
}
```

```
// Java
Key.COMPARATOR.compare(key1, key2);
// public static final field in Key class
```

A late-initialized property in an object or a companion object has a static backing field with the same visibility as the property setter.

```
object Singleton {
    lateinit var provider: Provider
}
```

```
// Java
Singleton.provider = new Provider();
// public static non-final field in Singleton class
```

Properties declared as `const` (in classes as well as at the top level) are turned into static fields in Java:

```
// file example.kt

object Obj {
    const val CONST = 1
}

class C {
    companion object {
        const val VERSION = 9
    }
}

const val MAX = 239
```

In Java:

```
int constant = Obj.CONST;
int max = ExampleKt.MAX;
int version = C.VERSION;
```

Static methods

Kotlin represents package-level functions as static methods. You can also generate static methods for functions defined in named objects or companion objects if you annotate such functions with the `@JvmStatic` annotation.

If you use `@JvmStatic` on a function in a companion object, the compiler generates both a static method in the enclosing class and an instance method in the companion object:

```
// Kotlin
class C {
    companion object {
        @JvmStatic fun callStatic() {}
        fun callNonStatic() {}
    }
}
```

In Java, you can call `callStatic()` on both the enclosing class and the companion object, while `callNonStatic()` is only available through the companion object:

```
// Java
C.callStatic();              // Success
C.callNonStatic();           // Error: not a static method
C.Companion.callStatic();    // Instance method remains
C.Companion.callNonStatic(); // Success
```

For a named object (a singleton), `@JvmStatic` turns the function into a static method of the object's class but doesn't generate a separate instance method:

```
// Kotlin
object Obj {
    @JvmStatic fun callStatic() {}
    fun callNonStatic() {}
}
```

In Java, you can call the `callStatic()` method on the named object, while `callNonStatic()` is only available through the singleton instance:

```
// Java
Obj.callStatic();              // Success
Obj.callNonStatic();           // Error: not a static method
Obj.INSTANCE.callNonStatic(); // Success, the call passed through the singleton instance
```

You can also annotate a function in an interface's companion object with `@JvmStatic`. Such functions compile to static methods in interfaces:

```
interface ChatBot {
    companion object {
        @JvmStatic fun greet(username: String) {
            println("Hello, $username")
        }
    }
}
```

You can also apply the `@JvmStatic` annotation to the property of an object or a companion object, making its getter and setter methods static members in that object or the class containing the companion object.

Default methods in interfaces

When targeting the JVM, Kotlin compiles functions declared in interfaces to default methods unless configured otherwise. These are concrete methods in interfaces that Java classes can inherit directly, without reimplementation.

Here is an example of a Kotlin interface with a default method:

```
interface Robot {
    fun move() { println("~walking~") }  // will be default in the Java interface
    fun speak(): Unit
}
```

The default implementation is available for Java classes implementing the interface.

```
//Java implementation
public class C3PO implements Robot {
    // move() implementation from Robot is available implicitly
    @Override
    public void speak() {
        System.out.println("I beg your pardon, sir");
    }
}
```

```
C3PO c3po = new C3PO();
c3po.move(); // default implementation from the Robot interface
c3po.speak();
```

Implementations of the interface can override default methods.

```java
//Java
public class BB8 implements Robot {
    //own implementation of the default method
    @Override
    public void move() {
        System.out.println("~rolling~");
    }

    @Override
    public void speak() {
        System.out.println("Beep-beep");
    }
}
```

Compatibility modes for default methods

Kotlin provides three modes for controlling how functions in interfaces are compiled to JVM default methods. These modes determine whether the compiler generates compatibility bridges and static methods in `DefaultImpls` classes.

You can control this behavior using the `-jvm-default` compiler option:

> The `-jvm-default` compiler option replaces the deprecated `-Xjvm-default` option.

Learn more about compatibility modes:

enable

Default behavior. Generates default implementations in interfaces and includes compatibility bridges and `DefaultImpls` classes. This mode maintains compatibility with older compiled Kotlin code.

no-compatibility

Generates only default implementations in interfaces. Skips compatibility bridges and `DefaultImpls` classes. Use this mode for new codebases that don't interact with code that relies on `DefaultImpls` classes. This can break binary compatibility with older Kotlin code.

> If interface delegation is used, all interface methods are delegated.

disable

Disables default implementations in interfaces. Only compatibility bridges and `DefaultImpls` classes are generated.

Visibility

The Kotlin maps visibility modifiers to Java as follows:

- `private` members remain `private`.
- `private` top-level declarations become `private` top-level declarations in Java. Package-private accessors are also included if accessed from within a class.

- `protected` members remain `protected`.

  Note that Java allows accessing protected members from other classes in the same package, and Kotlin doesn't.

- `internal` declarations become `public` in Java.

  The Kotlin compiler mangles the names of `internal` members in bytecode. This prevents accidental overrides across modules, for example, when extending a Kotlin class from Java, and allows overloading for members with the same signature.

  Note that the names of public members of `internal` classes aren't mangled and remain callable from Java.

- `public` members remain `public`.

KClass

Sometimes you need to call a Kotlin method with a parameter of type `KClass`. There is no automatic conversion from `Class` to `KClass`, so you have to do it manually by invoking the equivalent of the `Class<T>.kotlin` extension property:

```
kotlin.jvm.JvmClassMappingKt.getKotlinClass(MainView.class)
```

Handling signature clashes with @JvmName

Sometimes we have a named function in Kotlin, for which we need a different JVM name in bytecode. The most prominent example happens due to *type erasure*:

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

These two functions can not be defined side-by-side, because their JVM signatures are the same: `filterValid(Ljava/util/List;)Ljava/util/List;`. If we really want them to have the same name in Kotlin, we can annotate one (or both) of them with `@JvmName` and specify a different name as an argument:

```
fun List<String>.filterValid(): List<String>

@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

From Kotlin, they are accessible by the same name `filterValid`, but from Java it is `filterValid` and `filterValidInt`.

The same trick applies when we need to have a property `x` along with a function `getX()`:

```
val x: Int
    @JvmName("getX_prop")
    get() = 15

fun getX() = 10
```

To change the names of generated accessor methods for properties without explicitly implemented getters and setters, you can use `@get:JvmName` and `@set:JvmName`:

```
@get:JvmName("x")
@set:JvmName("changeX")
var x: Int = 23
```

Overloads generation

Normally, if you write a Kotlin function with default parameter values, it's visible in Java only as a full signature, with all parameters present.

You can use the `@IntroducedAt` annotation or the `@JvmOverloads` annotation to generate overloads for optional parameters.

Use `@IntroducedAt` when you add new optional parameters to published APIs and want the generated overloads to reflect the version in which each parameter was introduced. The compiler uses this information to automatically generate the corresponding hidden overloads.

This gives you version-based overload generation and helps preserve binary compatibility for callers compiled against older versions of your library.

> The `@IntroducedAt` annotation is Experimental. To opt in, use the `@OptIn(ExperimentalVersionOverloading::class)` annotation.

Here's an example where the `Button()` function receives multiple optional parameters over several API versions:

```
@OptIn(ExperimentalVersionOverloading::class)
fun Button(
    label: String = "",
    color: Color = DefaultColor,
    @IntroducedAt("1.1") borderColor: Color = DefaultBorderColor,
    @IntroducedAt("1.2") borderStyle: Style = DefaultBorderStyle,
    @IntroducedAt("1.2") borderWidth: Int = 1,
    onClick: () -> Unit
) {
    // Function body
}
```

Based on these versions, the compiler generates hidden overloads for the original API and for each API version that introduced new optional parameters:

```
// Original API
Button(
    label: String,
    color: Color,
    onClick: () -> Unit
)

// Version 1.1
Button(
    label: String,
    color: Color,
    borderColor: Color,
    onClick: () -> Unit
)

// Version 1.2
Button(
    label: String,
    color: Color,
    borderColor: Color,
    borderStyle: Style,
    borderWidth: Int,
    onClick: () -> Unit
)
```

If you want to expose multiple overloads to Java callers, you can also use the `@JvmOverloads` annotation.

The annotation also works for constructors, static methods, and so on. It can't be used on abstract methods, including methods defined in interfaces. For example, consider a `Circle` class with default parameter values:

```
class Circle @JvmOverloads constructor(centerX: Int, centerY: Int, radius: Double = 1.0) {
    @JvmOverloads fun draw(label: String, lineWidth: Int = 1, color: String = "red") { /*...*/ }
}
```

For every parameter with a default value, this generates one additional overload, which has this parameter and all parameters to the right of it in the parameter list removed. In this example, the following is generated:

```
// Constructors:
Circle(int centerX, int centerY, double radius)
Circle(int centerX, int centerY)

// Methods
void draw(String label, int lineWidth, String color) { }
void draw(String label, int lineWidth) { }
void draw(String label) { }
```

Since both `@IntroducedAt` and `@JvmOverloads` annotations generate overloads, using them together can create conflicting overloads. If you use both annotations, the compiler reports a warning. If you suppress the warning, the compiler prioritizes overloads generated from the `@IntroducedAt` annotation.

Note that, as described in Secondary constructors, if a class has default values for all constructor parameters, a public constructor with no arguments is generated for it. This works even if the `@JvmOverloads` annotation is not specified.

Checked exceptions

Kotlin does not have checked exceptions. So, normally the Java signatures of Kotlin functions do not declare exceptions thrown. Thus, if you have a function in Kotlin like this:

```kotlin
// example.kt
package demo

fun writeToFile() {
    /*...*/
    throw IOException()
}
```

And you want to call it from Java and catch the exception:

```java
// Java
try {
    demo.Example.writeToFile();
} catch (IOException e) {
    // error: writeToFile() does not declare IOException in the throws list
    // ...
}
```

You get an error message from the Java compiler, because `writeToFile()` does not declare `IOException`. To work around this problem, use the `@Throws` annotation in Kotlin:

```kotlin
@Throws(IOException::class)
fun writeToFile() {
    /*...*/
    throw IOException()
}
```

Null-safety

When calling Kotlin functions from Java, nobody prevents us from passing `null` as a non-nullable parameter. That's why Kotlin generates runtime checks for all public functions that expect non-nulls. This way we get a `NullPointerException` in the Java code immediately.

Variant generics

When Kotlin classes make use of declaration-site variance, there are two options of how their usages are seen from the Java code. For example, imagine you have the following class and two functions that use it:

```kotlin
class Box<out T>(val value: T)

interface Base
class Derived : Base

fun boxDerived(value: Derived): Box<Derived> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value
```

A naive way of translating these functions into Java would be this:

```java
Box<Derived> boxDerived(Derived value) { ... }
Base unboxBase(Box<Base> box) { ... }
```

The problem is that in Kotlin you can write `unboxBase(boxDerived(Derived()))` but in Java that would be impossible because in Java the class `Box` is *invariant* in its parameter `T`, and thus `Box<Derived>` is not a subtype of `Box<Base>`. To make this work in Java, you would have to define `unboxBase` as follows:

```
Base unboxBase(Box<? extends Base> box) { ... }
```

This declaration uses Java's *wildcards types* (`? extends Base`) to emulate declaration-site variance through use-site variance, because it is all Java has.

To make Kotlin APIs work in Java, the compiler generates `Box<Super>` as `Box<? extends Super>` for covariantly defined `Box` (or `Foo<? super Bar>` for contravariantly defined `Foo`) when it appears *as a parameter*. When it's a return value, wildcards are not generated, because otherwise Java clients will have to deal with them (and it's against the common Java coding style). Therefore, the functions from our example are actually translated as follows:

```
// return type - no wildcards
Box<Derived> boxDerived(Derived value) { ... }

// parameter - wildcards
Base unboxBase(Box<? extends Base> box) { ... }
```

When the argument type is final, there's usually no point in generating the wildcard, so `Box<String>` is always `Box<String>` , no matter what position it takes.

If you need wildcards where they are not generated by default, use the `@JvmWildcard` annotation:

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> = Box(value)
// is translated to
// Box<? extends Derived> boxDerived(Derived value) { ... }
```

In the opposite case, if you don't need wildcards where they are generated, use `@JvmSuppressWildcards` :

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
// is translated to
// Base unboxBase(Box<Base> box) { ... }
```

`@JvmSuppressWildcards` can be used not only on individual type arguments, but on entire declarations, such as functions or classes, causing all wildcards inside them to be suppressed.

Translation of type Nothing

The type `Nothing` is special, because it has no natural counterpart in Java. Indeed, every Java reference type, including `java.lang.Void` , accepts `null` as a value, and `Nothing` doesn't accept even that. So, this type cannot be accurately represented in the Java world. This is why Kotlin generates a raw type where an argument of type `Nothing` is used:

```
fun emptyList(): List<Nothing> = listOf()
// is translated to
// List emptyList() { ... }
```

Experimental

Inline value classes

If you want Java code to work smoothly with Kotlin's inline value classes, you can use the `@JvmExposeBoxed` annotation or the `-Xjvm-expose-boxed` compiler option. These approaches ensure Kotlin generates the necessary boxed representations for Java interoperability.

By default, Kotlin compiles inline value classes to use **unboxed representations**, which are often inaccessible from Java. For example, you can't call the constructor for the `MyInt` class from Java:

```
@JvmInline
value class MyInt(val value: Int)
```

So the following Java code fails:

```
MyInt input = new MyInt(5);
```

You can use the `@JvmExposeBoxed` annotation so that Kotlin generates a public constructor that you can call from Java directly. You can apply the annotation at the following levels to ensure fine-grained control over what's exposed to Java:

- Class
- Constructor
- Function

Before using the `@JvmExposeBoxed` annotation in your code, you must opt in by using `@OptIn(ExperimentalStdlibApi::class)`. For example:

```
@OptIn(ExperimentalStdlibApi::class)
@JvmExposeBoxed
@JvmInline
value class MyInt(val value: Int)

@OptIn(ExperimentalStdlibApi::class)
@JvmExposeBoxed
fun MyInt.timesTwoBoxed(): MyInt = MyInt(this.value * 2)
```

With these annotations, Kotlin generates a Java-accessible constructor for the `MyInt` class **and** a variant for the extension function that uses the boxed form of the value class. So the following Java code runs successfully:

```
MyInt input = new MyInt(5);
MyInt output = ExampleKt.timesTwoBoxed(input);
```

To apply this behavior to all inline value classes and the functions that use them within a module, compile it with the `-Xjvm-expose-boxed` option. Compiling with this option has the same effect as if every declaration in the module has the `@JvmExposeBoxed` annotation.

Inherited functions

The `@JvmExposeBoxed` annotation doesn't automatically generate boxed representations for inherited functions.

To generate the necessary representation for an inherited function, override it in the implementing or extending class:

```
interface IdTransformer {
    fun transformId(rawId: UInt): UInt = rawId
}

// Doesn't generate a boxed representation for the transformId() function
@OptIn(ExperimentalStdlibApi::class)
@JvmExposeBoxed
class LightweightTransformer : IdTransformer

// Generates a boxed representation for the transformId() function
@OptIn(ExperimentalStdlibApi::class)
@JvmExposeBoxed
class DefaultTransformer : IdTransformer {
    override fun transformId(rawId: UInt): UInt = super.transformId(rawId)
}
```

To learn how inheritance works in Kotlin and how to call superclass implementations using the `super` keyword, see Inheritance.

Using Java records in Kotlin

Records are classes in Java for storing immutable data. Records carry a fixed set of values – the *records components*. They have a concise syntax in Java and save you from having to write boilerplate code:

```
// Java
public record Person (String name, int age) {}
```

The compiler automatically generates a final class inherited from `java.lang.Record` with the following members:

- a private final field for each record component
- a public constructor with parameters for all fields
- a set of methods to implement structural equality: `equals()`, `hashCode()`, `toString()`
- a public method for reading each record component

Records are very similar to Kotlin data classes.

Using Java records from Kotlin code

You can use record classes with components that are declared in Java the same way you would use classes with properties in Kotlin. To access the record component, just use its name like you do for Kotlin properties:

```
val newPerson = Person("Kotlin", 10)
val firstName = newPerson.name
```

Declare records in Kotlin

Kotlin supports record declaration only for data classes, and the data class must meet the requirements.

To declare a record class in Kotlin, use the `@JvmRecord` annotation:

Applying `@JvmRecord` to an existing class is not a binary compatible change. It alters the naming convention of the class property accessors.

```
@JvmRecord
data class Person(val name: String, val age: Int)
```

This JVM-specific annotation enables generating:

- the record components corresponding to the class properties in the class file
- the property accessor methods named according to the Java record naming convention

The data class provides `equals()`, `hashCode()`, and `toString()` method implementations.

Requirements

To declare a data class with the `@JvmRecord` annotation, it must meet the following requirements:

- The class must be in a module that targets JVM 16 bytecode (or 15 if the `-Xjvm-enable-preview` compiler option is enabled).
- The class cannot explicitly inherit any other class (including `Any`) because all JVM records implicitly inherit `java.lang.Record`. However, the class can implement interfaces.
- The class cannot declare any properties with backing fields – except those initialized from the corresponding primary constructor parameters.
- The class cannot declare any mutable properties with backing fields.
- The class cannot be local.
- The primary constructor of the class must be as visible as the class itself.

Enable JVM records

JVM records require the `16` target version or higher of the generated JVM bytecode.

To specify it explicitly, use the `jvmTarget` compiler option in Gradle or Maven.

Experimental

Annotate record components in Kotlin

In Java, annotations on a record component are automatically propagated to the backing field, getter, setter, and constructor parameter. You can replicate this behavior in Kotlin by using the `all` use-site target.

For example:

```
@JvmRecord
data class Person(val name: String, @all:Positive val age: Int)
```

When you use `@JvmRecord` with `@all:`, Kotlin:

- Propagates the annotation to the property, backing field, constructor parameter, getter, and setter.
- Applies the annotation to the record component as well, if the annotation supports Java's `RECORD_COMPONENT`.

Make annotations work with record components

To make an annotation available for both Kotlin properties **and** Java record components, add the following meta-annotations to your annotation declaration:

- For Kotlin: `@Target`
- For Java record components: `@java.lang.annotation.Target`

For example:

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.PROPERTY)
@java.lang.annotation.Target(ElementType.CLASS, ElementType.RECORD_COMPONENT)
annotation class ExampleClass
```

You can now apply `@ExampleClass` to Kotlin classes and properties, as well as Java classes and record components.

Further discussion

See this language proposal for JVM records for further technical details and discussion.

Strings in Java and Kotlin

This guide contains examples of how to perform typical tasks with strings in Java and Kotlin. It will help you migrate from Java to Kotlin and write your code in the authentically Kotlin way.

Concatenate strings

In Java, you can do this in the following way:

```
// Java
String name = "Joe";
System.out.println("Hello, " + name);
System.out.println("Your name is " + name.length() + " characters long");
```

In Kotlin, use the dollar sign (`$`) before the variable name to interpolate the value of this variable into your string:

```
fun main() {
    // Kotlin
    val name = "Joe"
    println("Hello, $name")
    println("Your name is ${name.length} characters long")
}
```

You can interpolate the value of a complicated expression by surrounding it with curly braces, like in `${name.length}`. See string templates for more information.

Build a string

In Java, you can use the StringBuilder:

```java
// Java
StringBuilder countDown = new StringBuilder();
for (int i = 5; i > 0; i--) {
    countDown.append(i);
    countDown.append("\n");
}
System.out.println(countDown);
```

In Kotlin, use buildString() – an inline function that takes logic to construct a string as a lambda argument:

```kotlin
fun main() {
       // Kotlin
       val countDown = buildString {
           for (i in 5 downTo 1) {
               append(i)
               appendLine()
           }
       }
       println(countDown)
}
```

Under the hood, the `buildString` uses the same `StringBuilder` class as in Java, and you access it via an implicit `this` inside the lambda.

Learn more about lambda coding conventions.

Create a string from collection items

In Java, you use the Stream API to filter, map, and then collect the items:

```java
// Java
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
String invertedOddNumbers = numbers
        .stream()
        .filter(it -> it % 2 != 0)
        .map(it -> -it)
        .map(Object::toString)
        .collect(Collectors.joining("; "));
System.out.println(invertedOddNumbers);
```

In Kotlin, use the joinToString() function, which Kotlin defines for every List:

```kotlin
fun main() {
    // Kotlin
    val numbers = listOf(1, 2, 3, 4, 5, 6)
    val invertedOddNumbers = numbers
        .filter { it % 2 != 0 }
        .joinToString(separator = ";") {"${-it}"}
    println(invertedOddNumbers)
}
```

> In Java, if you want spaces between your delimiters and following items, you need to add a space to the delimiter explicitly.

Learn more about joinToString() usage.

Set default value if the string is blank

In Java, you can use the ternary operator:

```
// Java
public void defaultValueIfStringIsBlank() {
    String nameValue = getName();
    String name = nameValue.isBlank() ? "John Doe" : nameValue;
    System.out.println(name);
}

public String getName() {
    Random rand = new Random();
    return rand.nextBoolean() ? "" : "David";
}
```

Kotlin provides the inline function ifBlank() that accepts the default value as an argument:

```
// Kotlin
import kotlin.random.Random

fun main() {
    val name = getName().ifBlank { "John Doe" }
    println(name)
}

fun getName(): String =
    if (Random.nextBoolean()) "" else "David"
```

Replace characters at the beginning and end of a string

In Java, you can use the replaceAll() function. The `replaceAll()` function in this case accepts regular expressions `^##` and `##$`, which define strings starting and ending with `##` respectively:

```
// Java
String input = "##place##holder##";
String result = input.replaceAll("^##|##$", "");
System.out.println(result);
```

In Kotlin, use the removeSurrounding() function with the string delimiter `##`:

```
fun main() {
    // Kotlin
    val input = "##place##holder##"
    val result = input.removeSurrounding("##")
    println(result)
}
```

Replace occurrences

In Java, you can use the Pattern and the Matcher classes, for example, to obfuscate some data:

```java
// Java
String input = "login: Pokemon5, password: 1q2w3e4r5t";
Pattern pattern = Pattern.compile("\\w*\\d+\\w*");
Matcher matcher = pattern.matcher(input);
String replacementResult = matcher.replaceAll(it -> "xxx");
System.out.println("Initial input: '" + input + "'");
System.out.println("Anonymized input: '" + replacementResult + "'");
```

In Kotlin, you use the Regex class that simplifies working with regular expressions. Additionally, use multiline strings to simplify a regex pattern by reducing the count of backslashes:

```kotlin
fun main() {
    // Kotlin
    val regex = Regex("""\w*\d+\w*""") // multiline string
    val input = "login: Pokemon5, password: 1q2w3e4r5t"
    val replacementResult = regex.replace(input, replacement = "xxx")
    println("Initial input: '$input'")
    println("Anonymized input: '$replacementResult'")
}
```

Split a string

In Java, to split a string with the period character (`.`), you need to use shielding (`\\`). This happens because the split() function of the `String` class accepts a regular expression as an argument:

```java
// Java
System.out.println(Arrays.toString("Sometimes.text.should.be.split".split("\\.")));
```

In Kotlin, use the Kotlin function split(), which accepts varargs of delimiters as input parameters:

```kotlin
fun main() {
    // Kotlin
    println("Sometimes.text.should.be.split".split("."))
}
```

If you need to split with a regular expression, use the overloaded `split()` version that accepts the `Regex` as a parameter.

Take a substring

In Java, you can use the substring() function, which accepts an inclusive beginning index of a character to start taking the substring from. To take a substring after this character, you need to increment the index:

```java
// Java
String input = "What is the answer to the Ultimate Question of Life, the Universe, and Everything?
42";
String answer = input.substring(input.indexOf("?") + 1);
System.out.println(answer);
```

In Kotlin, you use the substringAfter() function and don't need to calculate the index of the character you want to take a substring after:

```kotlin
fun main() {
    // Kotlin
    val input = "What is the answer to the Ultimate Question of Life, the Universe, and Everything?
42"
    val answer = input.substringAfter("?")
    println(answer)
}
```

Additionally, you can take a substring after the last occurrence of a character:

```kotlin
fun main() {
    // Kotlin
    val input = "To be, or not to be, that is the question."
    val question = input.substringAfterLast(",")
    println(question)
}
```

Use multiline strings

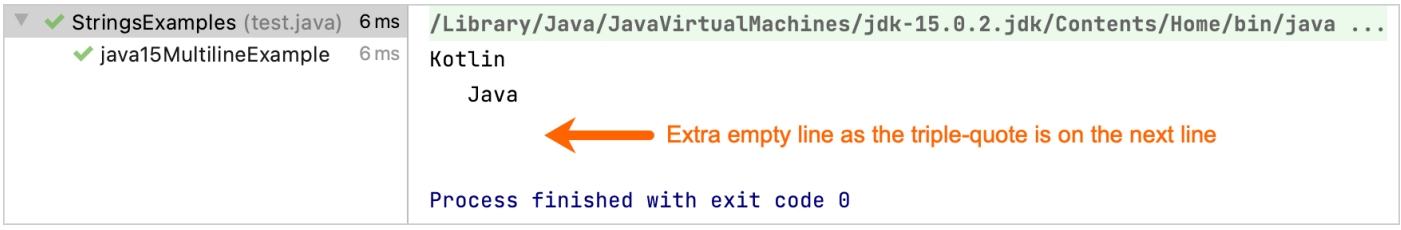
Before Java 15, there were several ways to create a multiline string. For example, using the join() function of the `String` class:

```java
// Java
String lineSeparator = System.getProperty("line.separator");
String result = String.join(lineSeparator,
       "Kotlin",
       "Java");
System.out.println(result);
```

In Java 15, text blocks appeared. There is one thing to keep in mind: if you print a multiline string and the triple-quote is on the next line, there will be an extra empty line:

```java
// Java
String result = """
    Kotlin
       Java
    """;
System.out.println(result);
```

The output:



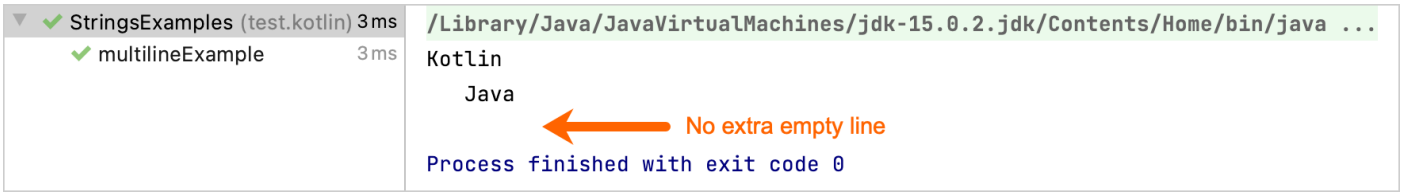
Java 15 multiline output

If you put the triple-quote on the same line as the last word, this difference in behavior disappears.

In Kotlin, you can format your line with the quotes on the new line, and there will be no extra empty line in the output. The left-most character of any line identifies the beginning of the line. The difference with Java is that Java automatically trims indents, and in Kotlin you should do it explicitly:

```kotlin
fun main() {
    // Kotlin
    val result = """
        Kotlin
           Java
    """.trimIndent()
    println(result)
}
```

The output:



Kotlin multiline output

To have an extra empty line, you should add this empty line to your multiline string explicitly.

In Kotlin, you can also use the trimMargin() function to customize the indents:

```kotlin
// Kotlin
fun main() {
    val result = """
       #  Kotlin
       #  Java
   """.trimMargin("#")
    println(result)
}
```

Learn more about multiline strings.

What's next?

- Look through other Kotlin idioms.
- Learn how to convert existing Java code to Kotlin with the Java to Kotlin converter.

If you have a favorite idiom, we invite you to share it by sending a pull request.

Collections in Java and Kotlin

Collections are groups of a variable number of items (possibly zero) that are significant to the problem being solved and are commonly operated on. This guide explains and compares collection concepts and operations in Java and Kotlin. It will help you migrate from Java to Kotlin and write your code in the authentically Kotlin way.

The first part of this guide contains a quick glossary of operations on the same collections in Java and Kotlin. It is divided into operations that are the same and operations that exist only in Kotlin. The second part of the guide, starting from Mutability, explains some of the differences by looking at specific cases.

For an introduction to collections, see the Collections overview or watch this video by Sebastian Aigner, Kotlin Developer Advocate.

All of the examples below use Java and Kotlin standard library APIs only.

Operations that are the same in Java and Kotlin

In Kotlin, there are many operations on collections that look exactly the same as their counterparts in Java.

Operations on lists, sets, queues, and deques

Description	Common operations	More Kotlin alternatives
Add an element or elements	`add()`, `addAll()`	Use the `plusAssign` (`+=`) operator: `collection += element`, `collection += anotherCollection`.
Check whether a collection contains an element or elements	`contains()`, `containsAll()`	Use the `in` keyword to call `contains()` in the operator form: `element in collection`.
Check whether a collection is empty	`isEmpty()`	Use `isNotEmpty()` to check whether a collection is not empty.
Remove under a certain condition	`removeIf()`	
Leave only selected elements	`retainAll()`	
Remove all elements from a collection	`clear()`	
Get a stream from a collection	`stream()`	Kotlin has its own way to process streams: sequences and methods like `map()` and `filter()`.
Get an iterator from a collection	`iterator()`	

Operations on maps

Description	Common operations	More Kotlin alternatives
Add an element or elements	`put()`, `putAll()`, `putIfAbsent()`	In Kotlin, the assignment `map[key] = value` behaves the same as `put(key, value)`. Also, you may use the `plusAssign` (`+=`) operator: `map += Pair(key, value)` or `map += anotherMap`.

Description	Common operations	More Kotlin alternatives
Replace an element or elements	`put()`, `replace()`, `replaceAll()`	Use the indexing operator `map[key] = value` instead of `put()` and `replace()`.
Get an element	`get()`	Use the indexing operator to get an element: `map[index]`.
Check whether a map contains an element or elements	`containsKey()`, `containsValue()`	Use the `in` keyword to call `contains()` in the operator form: `element in map`.
Check whether a map is empty	`isEmpty()`	Use `isNotEmpty()` to check whether a map is not empty.
Remove an element	`remove(key)`, `remove(key, value)`	Use the `minusAssign` (`-=`) operator: `map -= key`.
Remove all elements from a map	`clear()`	
Get a stream from a map	`stream()` on entries, keys, or values	

Operations that exist only for lists

Description	Common operations	More Kotlin alternatives
Get an index of an element	`indexOf()`	
Get the last index of an element	`lastIndexOf()`	
Get an element	`get()`	Use the indexing operator to get an element: `list[index]`.
Take a sublist	`subList()`	
Replace an element or elements	`set()`, `replaceAll()`	Use the indexing operator instead of `set()`: `list[index] = value`.

Operations that differ a bit

Operations on any collection type

Description	Java	Kotlin
Get a collection's size	`size()`	`count()`, `size`
Get flat access to nested collection elements	`collectionOfCollections.forEach(flatCollection::addAll)` or `collectionOfCollections.stream().flatMap().collect()`	`flatten()` or `flatMap()`
Apply the given function to every element	`stream().map().collect()`	`map()`
Apply the provided operation to collection elements sequentially and return the accumulated result	`stream().reduce()`	`reduce()`, `fold()`
Group elements by a classifier and count them	`stream().collect(Collectors.groupingBy(classifier, counting()))`	`eachCount()`
Filter by a condition	`stream().filter().collect()`	`filter()`
Check whether collection elements satisfy a condition	`stream().noneMatch()`, `stream().anyMatch()`, `stream().allMatch()`	`none()`, `any()`, `all()`
Sort elements	`stream().sorted().collect()`	`sorted()`
Take the first N elements	`stream().limit(N).collect()`	`take(N)`
Take elements with a predicate	`stream().takeWhile().collect()`	`takeWhile()`
Skip the first N elements	`stream().skip(N).collect()`	`drop(N)`
Skip elements with a predicate	`stream().dropWhile().collect()`	`dropWhile()`
Build maps from collection elements and certain values associated with them	`stream().collect(toMap(keyMapper, valueMapper))`	`associate()`

To perform all of the operations listed above on maps, you first need to get an `entrySet` of a map.

Operations on lists

Description	Java	Kotlin
Sort a list into natural order	`sort(null)`	`sort()`
Sort a list into descending order	`sort(comparator)`	`sortDescending()`
Remove an element from a list	`remove(index)`, `remove(element)`	`removeAt(index)`, `remove(element)` or `collection -= element`
Fill all elements of a list with a certain value	`Collections.fill()`	`fill()`
Get unique elements from a list	`stream().distinct().toList()`	`distinct()`

Operations that don't exist in Java's standard library

- `zip()`, `unzip()` – transform a collection.
- `aggregate()` – group by a condition.
- `takeLast()`, `takeLastWhile()`, `dropLast()`, `dropLastWhile()` – take or drop elements by a predicate.
- `slice()`, `chunked()`, `windowed()` – retrieve collection parts.
- Plus (`+`) and minus (`-`) operators – add or remove elements.

If you want to take a deep dive into `zip()`, `chunked()`, `windowed()`, and some other operations, watch this video by Sebastian Aigner about advanced collection operations in Kotlin:

Watch video online.

Mutability

In Java, there are mutable collections:

```
// Java
// This list is mutable!
public List<Customer> getCustomers() { ... }
```

Partially mutable ones:

```
// Java
List<String> numbers = Arrays.asList("one", "two", "three", "four");
numbers.add("five"); // Fails in runtime with `UnsupportedOperationException`
```

And immutable ones:

```java
// Java
List<String> numbers = new LinkedList<>();
// This list is immutable!
List<String> immutableCollection = Collections.unmodifiableList(numbers);
immutableCollection.add("five"); // Fails in runtime with `UnsupportedOperationException`
```

If you write the last two pieces of code in IntelliJ IDEA, the IDE will warn you that you're trying to modify an immutable object. This code will compile and fail in runtime with `UnsupportedOperationException`. You can't tell whether a collection is mutable by looking at its type.

Unlike in Java, in Kotlin you explicitly declare mutable or read-only collections depending on your needs. If you try to modify a read-only collection, the code won't compile:

```kotlin
// Kotlin
val numbers = mutableListOf("one", "two", "three", "four")
numbers.add("five")            // This is OK
val immutableNumbers = listOf("one", "two")
//immutableNumbers.add("five") // Compilation error - Unresolved reference: add
```

Read more about immutability on the Kotlin coding conventions page.

Covariance

In Java, you can't pass a collection with a descendant type to a function that takes a collection of the ancestor type. For example, if `Rectangle` extends `Shape`, you can't pass a collection of `Rectangle` elements to a function that takes a collection of `Shape` elements. To make the code compilable, use the `? extends Shape` type so the function can take collections with any inheritors of `Shape`:

```java
// Java
class Shape {}

class Rectangle extends Shape {}

public void doSthWithShapes(List<? extends Shape> shapes) {
/* If using just List<Shape>, the code won't compile when calling
this function with the List<Rectangle> as the argument as below */
}

public void main() {
    var rectangles = List.of(new Rectangle(), new Rectangle());
    doSthWithShapes(rectangles);
}
```

In Kotlin, read-only collection types are covariant. This means that if a `Rectangle` class inherits from the `Shape` class, you can use the type `List<Rectangle>` anywhere the `List<Shape>` type is required. In other words, the collection types have the same subtyping relationship as the element types. Maps are covariant on the value type, but not on the key type. Mutable collections aren't covariant – this would lead to runtime failures.

```kotlin
// Kotlin
open class Shape(val name: String)

class Rectangle(private val rectangleName: String) : Shape(rectangleName)

fun doSthWithShapes(shapes: List<Shape>) {
    println("The shapes are: ${shapes.joinToString { it.name }}")
}

fun main() {
    val rectangles = listOf(Rectangle("rhombus"), Rectangle("parallelepiped"))
    doSthWithShapes(rectangles)
}
```

Read more about collection types here.

Ranges and progressions

In Kotlin, you can create intervals using ranges. For example, `Version(1, 11)..Version(1, 30)` includes all of the versions from `1.11` to `1.30`. You can check that your version is in the range by using the `in` operator: `Version(0, 9) in versionRange`.

In Java, you need to manually check whether a `Version` fits both bounds:

```java
// Java
class Version implements Comparable<Version> {

    int major;
    int minor;

    Version(int major, int minor) {
        this.major = major;
        this.minor = minor;
    }

    @Override
    public int compareTo(Version o) {
        if (this.major != o.major) {
            return this.major - o.major;
        }
        return this.minor - o.minor;
    }
}

public void compareVersions() {
    var minVersion = new Version(1, 11);
    var maxVersion = new Version(1, 31);

   System.out.println(
           versionIsInRange(new Version(0, 9), minVersion, maxVersion));
   System.out.println(
           versionIsInRange(new Version(1, 20), minVersion, maxVersion));
}

public Boolean versionIsInRange(Version versionToCheck, Version minVersion,
                                Version maxVersion) {
    return versionToCheck.compareTo(minVersion) >= 0
            && versionToCheck.compareTo(maxVersion) <= 0;
}
```

In Kotlin, you operate with a range as a whole object. You don't need to create two variables and compare a `Version` with them:

```
// Kotlin
class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int {
        if (this.major != other.major) {
            return this.major - other.major
        }
        return this.minor - other.minor
    }
}

fun main() {
    val versionRange = Version(1, 11)..Version(1, 30)

    println(Version(0, 9) in versionRange)
    println(Version(1, 20) in versionRange)
}
```

As soon as you need to exclude one of the bounds, like to check whether a version is greater than or equal to (`>=`) the minimum version and less than (`<`) the maximum version, these inclusive ranges won't help.

Comparison by several criteria

In Java, to compare objects by several criteria, you may use the `comparing()` and `thenComparingX()` functions from the `Comparator` interface. For example, to compare people by their name and age:

```
class Person implements Comparable<Person> {
    String name;
    int age;

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return this.name + " " + age;
    }
}

public void comparePersons() {
    var persons = List.of(new Person("Jack", 35), new Person("David", 30),
            new Person("Jack", 25));
    System.out.println(persons.stream().sorted(Comparator
            .comparing(Person::getName)
            .thenComparingInt(Person::getAge)).collect(toList()));
}
```

In Kotlin, you just enumerate which fields you want to compare:

```
data class Person(
    val name: String,
    val age: Int
)

fun main() {
    val persons = listOf(Person("Jack", 35), Person("David", 30),
        Person("Jack", 25))
    println(persons.sortedWith(compareBy(Person::name, Person::age)))
}
```

Sequences

In Java, you can generate a sequence of numbers this way:

```
// Java
int sum = IntStream.iterate(1, e -> e + 3)
    .limit(10).sum();
System.out.println(sum); // Prints 145
```

In Kotlin, use *sequences*. Multistep processing of sequences is executed lazily when possible – actual computing happens only when the result of the whole processing chain is requested.

```
fun main() {
    // Kotlin
    val sum = generateSequence(1) {
        it + 3
    }.take(10).sum()
    println(sum) // Prints 145
}
```

Sequences may reduce the number of steps that are needed to perform some filtering operations. See the sequence processing example, which shows the difference between `Iterable` and `Sequence`.

Removal of elements from a list

In Java, the `remove()` function accepts an index of an element to remove.

When removing an integer element, use the `Integer.valueOf()` function as the argument for the `remove()` function:

```
// Java
public void remove() {
    var numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(2);
    numbers.add(3);
    numbers.add(1);
    numbers.remove(1); // This removes by index
    System.out.println(numbers); // [1, 3, 1]
    numbers.remove(Integer.valueOf(1));
    System.out.println(numbers); // [3, 1]
}
```

In Kotlin, there are two types of element removal: by index with `removeAt()` and by value with `remove()`.

```
fun main() {
    // Kotlin
    val numbers = mutableListOf(1, 2, 3, 1)
    numbers.removeAt(0)
    println(numbers) // [2, 3, 1]
    numbers.remove(1)
    println(numbers) // [2, 3]
}
```

Traverse a map

In Java, you can traverse a map via `forEach`:

```
// Java
numbers.forEach((k,v) -> System.out.println("Key = " + k + ", Value = " + v));
```

In Kotlin, use a `for` loop or a `forEach`, similar to Java's `forEach`, to traverse a map:

```
// Kotlin
for ((k, v) in numbers) {
    println("Key = $k, Value = $v")
}
// Or
numbers.forEach { (k, v) -> println("Key = $k, Value = $v") }
```

Get the first and the last items of a possibly empty collection

In Java, you can safely get the first and the last items by checking the size of the collection and using indices:

```
// Java
var list = new ArrayList<>();
// ...
if (list.size() > 0) {
    System.out.println(list.get(0));
    System.out.println(list.get(list.size() - 1));
}
```

In JDK 21 and later, you can also use the `getFirst()` and `getLast()` methods available on all `SequencedCollection` implementations. This includes all `List` implementations as well as other collections like `LinkedHashSet`. For example, with `ArrayList`:

```
// Java
var list = new ArrayList<>();
// ...
if (!list.isEmpty()) {
    System.out.println(list.getFirst());
    System.out.println(list.getLast());
}
```

In Kotlin, there are the special functions `firstOrNull()` and `lastOrNull()`. Using the `Elvis operator`, you can perform further actions right away depending on the result of a function. For example, `firstOrNull()`:

```kotlin
// Kotlin
val emails = listOf<String>() // Might be empty
val theOldestEmail = emails.firstOrNull() ?: ""
val theFreshestEmail = emails.lastOrNull() ?: ""
```

Create a set from a list

In Java, to create a `Set` from a `List`, you can use the `Set.copyOf` function:

```java
// Java
public void listToSet() {
    var sourceList = List.of(1, 2, 3, 1);
    var copySet = Set.copyOf(sourceList);
    System.out.println(copySet);
}
```

In Kotlin, use the function `toSet()`:

```kotlin
fun main() {
    // Kotlin
    val sourceList = listOf(1, 2, 3, 1)
    val copySet = sourceList.toSet()
    println(copySet)
}
```

Group elements

In Java, you can group elements with the Collectors function `groupingBy()`:

```java
// Java
public void analyzeLogs() {
    var requests = List.of(
        new Request("https://kotlinlang.org/docs/home.html", 200),
        new Request("https://kotlinlang.org/docs/home.html", 400),
        new Request("https://kotlinlang.org/docs/comparison-to-java.html", 200)
    );
    var urlsAndRequests = requests.stream().collect(
            Collectors.groupingBy(Request::getUrl));
    System.out.println(urlsAndRequests);
}
```

In Kotlin, use the function `groupBy()`:

```kotlin
data class Request(
    val url: String,
    val responseCode: Int
)

fun main() {
    // Kotlin
    val requests = listOf(
        Request("https://kotlinlang.org/docs/home.html", 200),
        Request("https://kotlinlang.org/docs/home.html", 400),
        Request("https://kotlinlang.org/docs/comparison-to-java.html", 200)
    )
    println(requests.groupBy(Request::url))
}
```

Filter elements

In Java, to filter elements from a collection, you need to use the Stream API. The Stream API has `intermediate` and `terminal` operations. `filter()` is an intermediate operation, which returns a stream. To receive a collection as the output, you need to use a terminal operation, like `collect()`. For example, to leave only those pairs whose keys end with `1` and whose values are greater than `10`:

```java
// Java
public void filterEndsWith() {
    var numbers = Map.of("key1", 1, "key2", 2, "key3", 3, "key11", 11);
    var filteredNumbers = numbers.entrySet().stream()
        .filter(entry -> entry.getKey().endsWith("1") && entry.getValue() > 10)
        .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
    System.out.println(filteredNumbers);
}
```

In Kotlin, filtering is built into collections, and `filter()` returns the same collection type that was filtered. So, all you need to write is the `filter()` and its predicate:

```kotlin
fun main() {
    // Kotlin
    val numbers = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredNumbers = numbers.filter { (key, value) -> key.endsWith("1") && value > 10 }
    println(filteredNumbers)
}
```

Learn more about filtering maps here.

Filter elements by type

In Java, to filter elements by type and perform actions on them, you need to check their types with the `instanceof` operator and then do the type cast:

```java
// Java
public void objectIsInstance() {
    var numbers = new ArrayList<>();
    numbers.add(null);
    numbers.add(1);
    numbers.add("two");
    numbers.add(3.0);
    numbers.add("four");
    System.out.println("All String elements in upper case:");
    numbers.stream().filter(it -> it instanceof String)
        .forEach( it -> System.out.println(((String) it).toUpperCase()));
}
```

In Kotlin, you just call `filterIsInstance<NEEDED_TYPE>()` on your collection, and the type cast is done by Smart casts:

```kotlin
// Kotlin
fun main() {
    // Kotlin
    val numbers = listOf(null, 1, "two", 3.0, "four")
    println("All String elements in upper case:")
    numbers.filterIsInstance<String>().forEach {
        println(it.uppercase())
    }
}
```

Test predicates

Some tasks require you to check whether all, none, or any elements satisfy a condition. In Java, you can do all of these checks via the Stream API functions `allMatch()`, `noneMatch()`, and `anyMatch()`:

```java
// Java
public void testPredicates() {
    var numbers = List.of("one", "two", "three", "four");
    System.out.println(numbers.stream().noneMatch(it -> it.endsWith("e"))); // false
    System.out.println(numbers.stream().anyMatch(it -> it.endsWith("e"))); // true
    System.out.println(numbers.stream().allMatch(it -> it.endsWith("e"))); // false
}
```

In Kotlin, the extension functions `none()`, `any()`, and `all()` are available for every Iterable object:

```kotlin
fun main() {
// Kotlin
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.none { it.endsWith("e") })
    println(numbers.any { it.endsWith("e") })
    println(numbers.all { it.endsWith("e") })
}
```

Learn more about test predicates.

Collection transformation operations

Zip elements

In Java, you can make pairs from elements with the same positions in two collections by iterating simultaneously over them:

```
// Java
public void zip() {
    var colors = List.of("red", "brown");
    var animals = List.of("fox", "bear", "wolf");

    for (int i = 0; i < Math.min(colors.size(), animals.size()); i++) {
        String animal = animals.get(i);
        System.out.println("The " + animal.substring(0, 1).toUpperCase()
               + animal.substring(1) + " is " + colors.get(i));
    }
}
```

If you want to do something more complex than just printing pairs of elements into the output, you can use Records. In the example above, the record would be `record AnimalDescription(String animal, String color) {}`.

In Kotlin, use the `zip()` function to do the same thing:

```
fun main() {
    // Kotlin
    val colors = listOf("red", "brown")
    val animals = listOf("fox", "bear", "wolf")

    println(colors.zip(animals) { color, animal ->
        "The ${animal.replaceFirstChar { it.uppercase() }} is $color" })
}
```

`zip()` returns the List of Pair objects.

> If collections have different sizes, the result of `zip()` is the smaller size. The last elements of the larger collection are not included in the result.

Associate elements

In Java, you can use the Stream API to associate elements with characteristics:

```
// Java
public void associate() {
    var numbers = List.of("one", "two", "three", "four");
    var wordAndLength = numbers.stream()
        .collect(toMap(number -> number, String::length));
    System.out.println(wordAndLength);
}
```

In Kotlin, use the `associate()` function:

```
fun main() {
    // Kotlin
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
}
```

What's next?

- Visit Kotlin Koans – complete exercises to learn Kotlin syntax. Each exercise is created as a failing unit test and your job is to make it pass.
- Look through other Kotlin idioms.
- Learn how to convert existing Java code to Kotlin with the Java to Kotlin converter.
- Discover collections in Kotlin.

If you have a favorite idiom, we invite you to share it by sending a pull request.

Nullability in Java and Kotlin

Nullability is the ability of a variable to hold a `null` value. When a variable contains `null`, an attempt to dereference the variable leads to a `NullPointerException`. There are many ways to write code in order to minimize the probability of receiving null pointer exceptions.

This guide covers differences between Java's and Kotlin's approaches to handling possibly nullable variables. It will help you migrate from Java to Kotlin and write your code in authentic Kotlin style.

The first part of this guide covers the most important difference – support for nullable types in Kotlin and how Kotlin processes types from Java code. The second part, starting from Checking the result of a function call, examines several specific cases to explain certain differences.

Learn more about null safety in Kotlin.

Support for nullable types

The most important difference between Kotlin's and Java's type systems is Kotlin's explicit support for nullable types. It is a way to indicate which variables can possibly hold a `null` value. If a variable can be `null`, it's not safe to call a method on the variable because this can cause a `NullPointerException`. Kotlin prohibits such calls at compile time and thereby prevents lots of possible exceptions. At runtime, objects of nullable types and objects of non-nullable types are treated the same: A nullable type isn't a wrapper for a non-nullable type. All checks are performed at compile time. That means there's almost no runtime overhead for working with nullable types in Kotlin.

We say "almost" because, even though intrinsic checks *are* generated, their overhead is minimal.

In Java, if you don't write null checks, methods may throw a `NullPointerException`:

```java
// Java
int stringLength(String a) {
    return a.length();
}

void main() {
    stringLength(null); // Throws a `NullPointerException`
}
```

This call will have the following output:

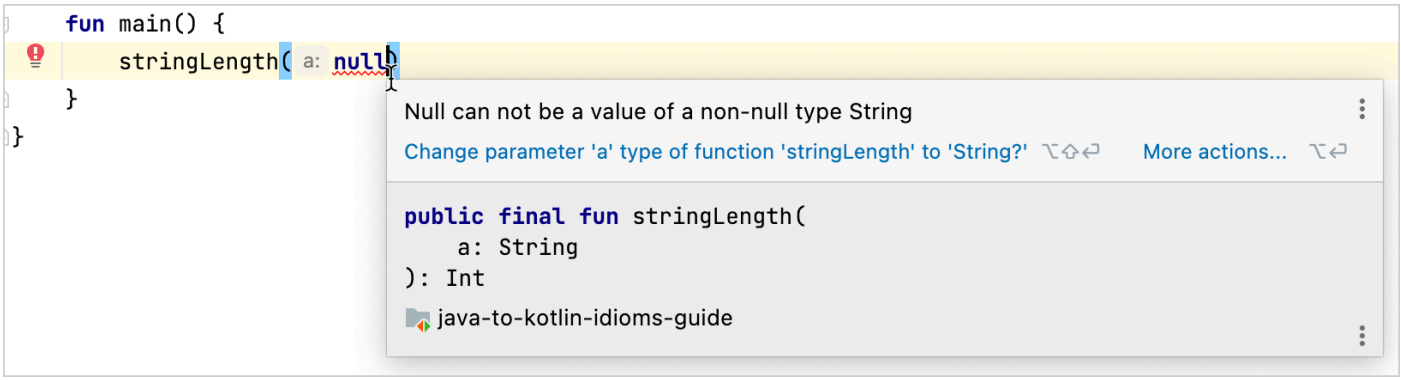
```
java.lang.NullPointerException: Cannot invoke "String.length()" because "a" is null
    at test.java.Nullability.stringLength(Nullability.java:8)
    at test.java.Nullability.main(Nullability.java:12)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)
```

In Kotlin, all regular types are non-nullable by default unless you explicitly mark them as nullable. If you don't expect `a` to be `null`, declare the `stringLength()` function as follows:

```kotlin
// Kotlin
fun stringLength(a: String) = a.length
```

The parameter `a` has the `String` type, which in Kotlin means it must always contain a `String` instance and it cannot contain `null`. Nullable types in Kotlin are marked with a question mark `?`, for example, `String?`. The situation with a `NullPointerException` at runtime is impossible if `a` is `String` because the compiler enforces the rule that all arguments of `stringLength()` not be `null`.

An attempt to pass a `null` value to the `stringLength(a: String)` function will result in a compile-time error, "Null can not be a value of a non-null type String":



Passing null to a non-nullable function error

If you want to use this function with any arguments, including `null`, use a question mark after the argument type `String?` and check inside the function body to ensure that the value of the argument is not `null`:

```kotlin
// Kotlin
fun stringLength(a: String?): Int = if (a != null) a.length else 0
```

After the check is passed successfully, the compiler treats the variable as if it were of the non-nullable type `String` in the scope where the compiler performs the check.

If you don't perform this check, the code will fail to compile with the following message: "Only safe (?.) or non-nullable asserted (!!.) calls are allowed on a nullable receiver of type String?".

You can write the same shorter – use the safe-call operator ?. (If-not-null shorthand), which allows you to combine a null check and a method call into a single operation:

```
// Kotlin
fun stringLength(a: String?): Int = a?.length ?: 0
```

Platform types

In Java, you can use annotations showing whether a variable can or cannot be `null`. Such annotations aren't part of the standard library, but you can add them separately. For example, you can use JetBrains annotations `@Nullable` and `@NotNull` (from the `org.jetbrains.annotations` package), annotations from JSpecify (`org.jspecify.annotations`), or from Eclipse (`org.eclipse.jdt.annotation`). Kotlin can recognize such annotations when you're calling Java code from Kotlin code and will treat types according to their annotations.

If your Java code doesn't have these annotations, then Kotlin will treat Java types as *platform types*. But since Kotlin doesn't have nullability information for such types, its compiler will allow all operations on them. You will need to decide whether to perform null checks, because:

- Just as in Java, you'll get a `NullPointerException` if you try to perform an operation on `null`.
- The compiler won't highlight any redundant null checks, which it normally does when you perform a null-safe operation on a value of a non-nullable type.

Learn more about calling Java from Kotlin in regard to null-safety and platform types.

Support for definitely non-nullable types

In Kotlin, if you want to override a Java method that contains `@NotNull` as an argument, you need Kotlin's definitely non-nullable types.

For example, consider this `load()` method in Java:

```
import org.jetbrains.annotations.*;

public interface Game<T> {
  public T save(T x) {}
  @NotNull
  public T load(@NotNull T x) {}
}
```

To override the `load()` method in Kotlin successfully, you need `T1` to be declared as definitely non-nullable (`T1 & Any`):

```
interface ArcadeGame<T1> : Game<T1> {
  override fun save(x: T1): T1
  // T1 is definitely non-nullable
  override fun load(x: T1 & Any): T1 & Any
}
```

Learn more about generic types that are definitely non-nullable.

Checking the result of a function call

One of the most common situations where you need to check for `null` is when you obtain a result from a function call.

In the following example, there are two classes, `Order` and `Customer`. `Order` has a reference to an instance of `Customer`. The `findOrder()` function returns an instance of the `Order` class, or `null` if it can't find the order. The objective is to process the customer instance of the retrieved order.

Here are the classes in Java:

```
//Java
record Order (Customer customer) {}

record Customer (String name) {}
```

In Java, call the function and do an if-not-null check on the result to proceed with the dereferencing of the required property:

```
// Java
Order order = findOrder();

if (order ≠ null) {
    processCustomer(order.getCustomer());
}
```

Converting the Java code above to Kotlin code directly results in the following:

```
// Kotlin
data class Order(val customer: Customer)

data class Customer(val name: String)

val order = findOrder()

// Direct conversion
if (order ≠ null){
    processCustomer(order.customer)
}
```

Use the safe-call operator `?.` (If-not-null shorthand) in combination with any of the scope functions from the standard library. The `let` function is usually used for this:

```
// Kotlin
val order = findOrder()

order?.let {
    processCustomer(it.customer)
}
```

Here is a shorter version of the same:

```
// Kotlin
findOrder()?.customer?.let(::processCustomer)
```

Default values instead of null

Checking for `null` is often used in combination with setting the default value in case the null check is successful.

The Java code with a null check:

```
// Java
Order order = findOrder();
if (order == null) {
    order = new Order(new Customer("Antonio"))
}
```

To express the same in Kotlin, use the Elvis operator (If-not-null-else shorthand):

```
// Kotlin
val order = findOrder() ?: Order(Customer("Antonio"))
```

Functions returning a value or null

In Java, you need to be careful when working with list elements. You should always check whether an element exists at an index before you attempt to use the element:

```
// Java
var numbers = new ArrayList<Integer>();
numbers.add(1);
numbers.add(2);

System.out.println(numbers.get(0));
//numbers.get(5) // Exception!
```

The Kotlin standard library often provides functions whose names indicate whether they can possibly return a `null` value. This is especially common in the collections API:

```
fun main() {
    // Kotlin
    // The same code as in Java:
    val numbers = listOf(1, 2)

    println(numbers[0])  // Can throw IndexOutOfBoundsException if the collection is empty
    //numbers.get(5)     // Exception!

    // More abilities:
    println(numbers.firstOrNull())
    println(numbers.getOrNull(5)) // null
}
```

Aggregate operations

When you need to get the biggest element or `null` if there are no elements, in Java you would use the Stream API:

```java
// Java
var numbers = new ArrayList<Integer>();
var max = numbers.stream().max(Comparator.naturalOrder()).orElse(null);
System.out.println("Max: " + max);
```

In Kotlin, use aggregate operations:

```kotlin
// Kotlin
val numbers = listOf<Int>()
println("Max: ${numbers.maxOrNull()}")
```

Learn more about collections in Java and Kotlin.

Casting types safely

When you need to safely cast a type, in Java you would use the `instanceof` operator and then check how well it worked:

```java
// Java
int getStringLength(Object y) {
    return y instanceof String x ? x.length() : -1;
}

void main() {
    System.out.println(getStringLength(1)); // Prints `-1`
}
```

To avoid exceptions in Kotlin, use the safe cast operator `as?`, which returns `null` on failure:

```kotlin
// Kotlin
fun main() {
    println(getStringLength(1)) // Prints `-1`
}

fun getStringLength(y: Any): Int {
    val x: String? = y as? String // null
    return x?.length ?: -1 // Returns -1 because `x` is null
}
```

In the Java example above, the function `getStringLength()` returns a result of the primitive type `int`. To make it return `null`, you can use the *boxed* type `Integer`. However, it's more resource-efficient to make such functions return a negative value and then check the value – you would do the check anyway, but no additional boxing is performed this way.

When migrating Java code to Kotlin, you may want to initially use the regular cast operator `as` with a nullable type to preserve the original semantics of your code. However, we recommend adapting your code to use the safe cast operator `as?` for a safer and more idiomatic approach. For example, if you have the following Java code:

```java
public class UserProfile {
    Object data;

    public static String getUsername(UserProfile profile) {
        if (profile == null) {
            return null;
        }
        return (String) profile.data;
    }
}
```

Migrating this directly with the `as` operator gives you:

```kotlin
class UserProfile(var data: Any? = null)

fun getUsername(profile: UserProfile?): String? {
    if (profile == null) {
        return null
    }
    return profile.data as String?
}
```

Here, `profile.data` is cast to a nullable string using `as String?`.

We recommend going one step further and using `as? String` to safely cast the value. This approach returns `null` on failure instead of throwing a `ClassCastException`:

```kotlin
class UserProfile(var data: Any? = null)

fun getUsername(profile: UserProfile?): String? =
  profile?.data as? String
```

This version replaces the `if` expression with the safe call operator `?.`, which safely accesses the data property before attempting the cast.

What's next?

- Browse other Kotlin idioms.
- Learn how to convert existing Java code to Kotlin with the Java-to-Kotlin (J2K) converter.
- Check out other migration guides:
  - Strings in Java and Kotlin
  - Collections in Java and Kotlin

If you have a favorite idiom, feel free to share it with us by sending a pull request!

Standard input

Java Scanner is a slow tool. Use it only when you need the specific functionalities it offers. Otherwise, it's generally preferable to use Kotlin's `readln()` function to read standard input.

To read from the standard input, Java provides the `Scanner` class. Kotlin offers two main ways to read from the standard input: the `Scanner` class, similar to Java, and the `readln()` function.

Read from the standard input with Java Scanner

In Java, the standard input is typically accessed through the `System.in` object. You need to import the `Scanner` class, create an object, and use methods like `.nextLine()` and `.nextInt()` to read different data types:

```
//Java implementation
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Reads a single line of input. For example: Hi there!
        System.out.print("Enter a line: ");
        String line = scanner.nextLine();
        System.out.println("You entered: " + line);
        // You entered: Hi there!

        // Reads an integer. For example: 08081990
        System.out.print("Enter an integer: ");
        int number = scanner.nextInt();
        System.out.println("You entered: " + number);
        // You entered: 08081990

        scanner.close();
    }
}
```

Use Java Scanner in Kotlin

Due to Kotlin's interoperability with Java libraries, you can access Java Scanner from Kotlin code out of the box.

To use Java Scanner in Kotlin, you need to import the `Scanner` class and initialize it by passing a `System.in` object that represents the standard input stream and dictates how to read the data. You can use the available reading methods for reading values different from strings, such as `.nextLine()`, `.next()`, and `.nextInt()`:

```
// Imports Java Scanner
import java.util.Scanner

fun main() {
    // Initializes the Scanner
    val scanner = Scanner(System.`in`)

    // Reads a whole string line. For example: "Hello, Kotlin"
    val line = scanner.nextLine()
    print(line)
    // Hello, Kotlin

    // Reads a string. For example: "Hello"
    val string = scanner.next()
    print(string)
    // Hello

    // Reads a number. For example: 123
    val num = scanner.nextInt()
    print(num)
    // 123
}
```

Other useful methods for reading input with Java Scanner are `.hasNext()`, `.useDelimiter()`, and `.close()`:

- The `.hasNext()` method checks if there's more data available in the input. It returns the boolean value `true` if there are remaining elements to iterate and `false` if no more elements are left in the input.
- The `.useDelimiter()` method sets the delimiter for reading input elements. The delimiter is whitespaces by default, but you can specify other characters. For example, `.useDelimiter(",")` reads the input elements separated by commas.
- The `.close()` method closes the input stream associated with the Scanner, preventing further use of the Scanner for reading input.

Always use the `.close()` method when you're finished using Java Scanner. Closing the Java Scanner releases the resources it consumes and ensures proper program behavior.

Read from the standard input with readln()

In Kotlin, apart from the Java Scanner, you have the `readln()` function. It is the most straightforward way to read input. This function reads a line of text from the standard input and returns it as a string:

```
// Reads a string. For example: Charlotte
val name = readln()

// Reads a string and converts it into an integer. For example: 43
val age = readln().toInt()

println("Hello, $name! You are $age years old.")
// Hello, Charlotte! You are 43 years old.
```

For more information, see Read standard input.

Browser and DOM API

The `kotlinx-browser` library lets you access browser-specific functionality. It includes common top-level objects such as `document` and `window` and, where possible, provides typesafe wrappers for their functionality.

As a fallback, the `dynamic` type provides access to functions that do not map well into the Kotlin type system.

To use the browser and DOM API, add the `kotlinx-browser` library as a dependency to your project's `build.gradle(.kts)` file:

Kotlin

```
repositories {
    mavenCentral()
}

kotlin {
    js {
        browser()
    }

    sourceSets {
        commonMain {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-browser:0.5.0")
            }
        }
    }
}
```

Groovy

```
repositories {
    mavenCentral()
}

kotlin {
    js {
        browser()
    }

    sourceSets {
        commonMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-browser:0.5.0'
            }
        }
    }
}
```

Interaction with the DOM

For interaction with the Document Object Model (DOM), you can use the variable `document`. For example, you can set the background color of our website through this object:

```
document.bgColor = "FFAA12"
```

The `document` object also provides you a way to retrieve a specific element by ID, name, class name, tag name and so on. All returned elements have the `Element?` type. To access their properties, you need to cast them to their appropriate type. For example, assume that you have an HTML page with an email `<input>` field:

```
<body>
    <input type="text" name="email" id="email"/>

    <script type="text/javascript" src="tutorial.js"></script>
</body>
```

Note that your script is included at the bottom of the `body` tag. This ensures that the DOM is fully available before the script is loaded.

With this setup, you can access elements of the DOM. To access the properties of the `input` field, invoke `getElementById` and cast it to `HTMLInputElement`. You can then safely access its properties, such as `value`:

```
val email = document.getElementById("email") as HTMLInputElement
email.value = "hadi@jetbrains.com"
```

Much like you reference this `input` element, you can access other elements on the page, casting them to the appropriate types.

What's next?

To see how to create and structure elements in the DOM in a concise way, check out the Typesafe HTML DSL.

Use JavaScript code from Kotlin

Kotlin was first designed for easy interoperation with the Java platform: it sees Java classes as Kotlin classes, and Java sees Kotlin classes as Java classes.

However, JavaScript is a dynamically typed language, which means it does not check types at compile time. You can freely talk to JavaScript from Kotlin via dynamic types. If you want to use the full power of the Kotlin type system, you can create external declarations for JavaScript libraries which will be understood by the Kotlin compiler and the surrounding tooling.

Inline JavaScript

You can inline JavaScript code into your Kotlin code using the `js()` function:

```
fun jsTypeOf(o: Any): String {
    return js("typeof o")
}
```

JavaScript code inlining has full support for ES2015 features, including:

- `const` and `let` variable declarations
- ES classes
- Generators

- Lambdas (arrow functions)
- Spread and rest operators
- Template strings

Because the parameter of `js` is parsed at compile time and translated to JavaScript code "as-is", it must be a string constant. So, the following code is incorrect:

```
fun jsTypeOf(o: Any): String {
    return js(getTypeof() + " o") // Error: Argument must be a string constant
    // The compiler cannot evaluate string concatenations
}

fun getTypeof() = "typeof"
```

Instead, for example to inline the rest operator, use a string constant:

```
fun runSumExample() {
    val sum = js("(...nums) => nums.reduce((a, b) => a + b, 0)")
    println(sum(1, 2, 3, 4))
}
```

Invoking `js()` returns a result of `dynamic` type, which doesn't provide type safety at compile time.

`external` modifier

To tell Kotlin that a certain declaration is written in pure JavaScript, you should mark it with the `external` modifier. When the compiler sees such a declaration, it assumes that the implementation for the corresponding class, function or property is provided externally (by the developer or via an npm dependency), and therefore does not try to generate any JavaScript code from the declaration. This is also why `external` declarations can't have a body. For example:

```
external fun alert(message: Any?): Unit

external class Node {
    val firstChild: Node

    fun append(child: Node): Node

    fun removeChild(child: Node): Node

    // etc
}

external val window: Window
```

Note that the `external` modifier is inherited by nested declarations. This is why in the example `Node` class, there is no `external` modifier before member functions and properties.

The `external` modifier is only allowed on package-level declarations. You can't declare an `external` member of a non-`external` class.

Declare (static) members of a class

In JavaScript, you can define members either on a prototype or a class itself:

```
function MyClass() { ... }
MyClass.sharedMember = function() { /* implementation */ };
MyClass.prototype.ownMember = function() { /* implementation */ };
```

There is no such syntax in Kotlin. However, in Kotlin we have `companion` objects. Kotlin treats companion objects of `external` classes in a special way: instead of expecting an object, it assumes members of companion objects to be members of the class itself. `MyClass` from the example above can be described as follows:

```
external class MyClass {
    companion object {
        fun sharedMember()
    }

    fun ownMember()
}
```

Declare parameters with default values

If you are writing an external declaration for a JavaScript function which has a parameter with a default value, use `definedExternally`. This delegates the generation of the default values to the JavaScript function itself:

```
external fun myFunWithOptionalArgs(
    x: Int,
    y: String = definedExternally,
    z: String = definedExternally
)
```

With this external declaration, you can call `myFunWithOptionalArgs` with one required argument and two optional arguments, where the default values are calculated by the JavaScript implementation of `myFunWithOptionalArgs`.

Extend JavaScript classes

You can easily extend JavaScript classes as if they were Kotlin classes. Just define an `external open` class and extend it by a non-`external` class. For example:

```
open external class Foo {
    open fun run()
    fun stop()
}

class Bar : Foo() {
    override fun run() {
        window.alert("Running!")
    }

    fun restart() {
        window.alert("Restarting")
    }
}
```

There are some limitations:

- When a function of an external base class is overloaded by signature, you can't override it in a derived class.
- You can't override a function including parameters with default values.
- Non-external classes can't be extended by external classes.

external interfaces

JavaScript does not have the concept of interfaces. When a function expects its parameter to support two methods `foo` and `bar`, you would just pass in an object that actually has these methods.

You can use interfaces to express this concept in statically typed Kotlin:

```
external interface HasFooAndBar {
    fun foo()

    fun bar()
}

external fun myFunction(p: HasFooAndBar)
```

A typical use case for external interfaces is to describe settings objects. For example:

```
external interface JQueryAjaxSettings {
    var async: Boolean

    var cache: Boolean

    var complete: (JQueryXHR, String) -> Unit

    // etc
}

fun JQueryAjaxSettings(): JQueryAjaxSettings = js("{}")

external class JQuery {
    companion object {
        fun get(settings: JQueryAjaxSettings): JQueryXHR
    }
}

fun sendQuery() {
    JQuery.get(JQueryAjaxSettings().apply {
        complete = { (xhr, data) ->
            window.alert("Request complete")
        }
    })
}
```

External interfaces have some restrictions:

- They can't be used on the right-hand side of `is` checks.
- They can't be passed as reified type arguments.
- They can't be used in class literal expressions (such as `I::class`).
- `as` casts to external interfaces always succeed. Casting to external interfaces produces the "Unchecked cast to external interface" compile time warning. The warning can be suppressed with the `@Suppress("UNCHECKED_CAST_TO_EXTERNAL_INTERFACE")` annotation.

IntelliJ IDEA can also automatically generate the `@Suppress` annotation. Open the intentions menu via the light bulb icon or Alt-Enter, and click the small arrow next to the "Unchecked cast to external interface" inspection. Here, you can select the suppression scope, and your IDE will add the annotation to your file accordingly.

Casts

In addition to the "unsafe" cast operator `as`, which throws a `ClassCastException` in case a cast is not possible, Kotlin/JS also provides `unsafeCast<T>()`. When using `unsafeCast`, *no type checking is done at all* during runtime. For example, consider the following two methods:

```
fun usingUnsafeCast(s: Any) = s.unsafeCast<String>()
fun usingAsOperator(s: Any) = s as String
```

They will be compiled accordingly:

```
function usingUnsafeCast(s) {
    return s;
}

function usingAsOperator(s) {
    var tmp$;
    return typeof (tmp$ = s) === 'string' ? tmp$ : throwCCE();
}
```

Equality

Kotlin/JS has particular semantics for equality checks compared to other platforms.

In Kotlin/JS, the Kotlin referential equality operator (`===`) always translates to the JavaScript strict equality operator (`===`).

The JavaScript `===` operator checks not only that two values are equal but also that the types of these two values are equal:

```
fun main() {
   val name = "kotlin"
   val value1 = name.substring(0, 1)
   val value2 = name.substring(0, 1)

   println(value1 === value2)
   // Prints 'true' on Kotlin/JS
   // Prints 'false' on other platforms
}
```

Also, in Kotlin/JS, the `Byte`, `Short`, `Int`, `Float`, and `Double` numeric types are all represented with the `Number` JavaScript type in runtime. Therefore, the values of these five types are indistinguishable:

```
fun main() {
   println(1.0 as Any === 1 as Any)
   // Prints 'true' on Kotlin/JS
   // Prints 'false' on other platforms
}
```

For more information about equality in Kotlin, see the Equality documentation.

Dynamic type

The dynamic type is not supported in code targeting the JVM.

Being a statically typed language, Kotlin still has to interoperate with untyped or loosely typed environments, such as the JavaScript ecosystem. To facilitate these use cases, the `dynamic` type is available in the language:

```
val dyn: dynamic = ...
```

The `dynamic` type basically turns off Kotlin's type checker:

- A value of the `dynamic` type can be assigned to any variable or passed anywhere as a parameter.
- Any value can be assigned to a variable of the `dynamic` type or passed to a function that takes `dynamic` as a parameter.
- `null` -checks are disabled for the `dynamic` type values.

The most peculiar feature of `dynamic` is that we are allowed to call **any** property or function with any parameters on a `dynamic` variable:

```
dyn.whatever(1, "foo", dyn) // 'whatever' is not defined anywhere
dyn.whatever(*arrayOf(1, 2, 3))
```

On the JavaScript platform this code will be compiled "as is": `dyn.whatever(1)` in Kotlin becomes `dyn.whatever(1)` in the generated JavaScript code.

When calling functions written in Kotlin on values of `dynamic` type, keep in mind the name mangling performed by the Kotlin to JavaScript compiler. You may need to use the @JsName annotation to assign well-defined names to the functions that you need to call.

A dynamic call always returns `dynamic` as a result, so you can chain such calls freely:

```
dyn.foo().bar.baz()
```

When you pass a lambda to a dynamic call, all of its parameters by default have the type `dynamic`:

```
dyn.foo {
    x → x.bar() // x is dynamic
}
```

Expressions using values of `dynamic` type are translated to JavaScript "as is", and do not use the Kotlin operator conventions. The following operators are supported:

- binary: `+`, `-`, `*`, `/`, `%`, `>`, `<`, `>=`, `<=`, `==`, `!=`, `===`, `!==`, `&&`, `||`
- unary
  - prefix: `-`, `+`, `!`
  - prefix and postfix: `++`, `--`
- assignments: `+=`, `-=`, `*=`, `/=`, `%=`
- indexed access:
  - read: `d[a]`, more than one argument is an error

- write: `d[a1] = a2`, more than one argument in `[]` is an error

`in`, `!in` and `..` operations with values of type `dynamic` are forbidden.

For a more technical description, see the spec document.

Use dependencies from npm

In Kotlin/JS projects, all dependencies can be managed through the Gradle plugin. This includes Kotlin/Multiplatform libraries such as `kotlinx.coroutines`, `kotlinx.serialization`, or `ktor-client`.

For depending on JavaScript packages from npm, the Gradle DSL exposes an `npm` function that lets you specify packages you want to import from npm. Let's consider the import of an NPM package called `is-sorted`.

The corresponding part in the Gradle build file looks as follows:

```
dependencies {
    // ...
    implementation(npm("is-sorted", "1.0.5"))
}
```

Because JavaScript modules are usually dynamically typed and Kotlin is a statically typed language, you need to provide a kind of adapter. In Kotlin, such adapters are called *external declarations*. For the `is-sorted` package which offers only one function, this declaration is small to write. Inside the source folder, create a new file called `is-sorted.kt`, and fill it with these contents:

```
@JsModule("is-sorted")
@JsNonModule
external fun <T> sorted(a: Array<T>): Boolean
```

Please note that if you're using CommonJS as a target, the `@JsModule` and `@JsNonModule` annotations need to be adjusted accordingly.

This JavaScript function can now be used just like a regular Kotlin function. Because we provided type information in the header file (as opposed to simply defining parameter and return type to be `dynamic`), proper compiler support and type-checking is also available.

```
console.log("Hello, Kotlin/JS!")
console.log(sorted(arrayOf(1,2,3)))
console.log(sorted(arrayOf(3,1,2)))
```

Running these three lines either in the browser or Node.js, the output shows that the call to `sorted` was properly mapped to the function exported by the `is-sorted` package:

```
Hello, Kotlin/JS!
true
false
```

Because the JavaScript ecosystem has multiple ways of exposing functions in a package (for example, through named or default exports), other npm packages might need a slightly altered structure for their external declarations.

To learn more about how to write declarations, please refer to Calling JavaScript from Kotlin.

Use Kotlin code from JavaScript

Depending on the selected JavaScript Module system, the Kotlin/JS compiler generates different output. But in general, the Kotlin compiler generates normal JavaScript classes, functions, and properties, which you can freely use from JavaScript code. There are some subtle things you should remember, though.

Isolating declarations in a separate JavaScript object in plain mode

If you have explicitly set your module kind to be `plain`, Kotlin creates an object that contains all Kotlin declarations from the current module. This is done to prevent spoiling the global object. This means that for a module `myModule`, all declarations are available to JavaScript via the `myModule` object. For example:

```
fun foo() = "Hello"
```

This function can be called from JavaScript like this:

```
alert(myModule.foo());
```

Calling the function directly like this is not applicable when you compile your Kotlin module to JavaScript modules like UMD (the default setting for both `browser` and `nodejs` targets), ESM, CommonJS, or AMD. In these cases, your declarations are exposed according to the chosen JavaScript module system. For example, when using UMD, ESM, or CommonJS, your call site would look like this:

```
alert(require('myModule').foo());
```

For more information about JavaScript module systems, see JavaScript Modules.

Package structure

For most of the module systems (CommonJS, Plain, and UMD), Kotlin exposes its package structure to JavaScript. Unless you define your declarations in the root package, you have to use fully qualified names in JavaScript. For example:

```
package my.qualified.packagename

fun foo() = "Hello"
```

For example, when using UMD or CommonJS, your call site could look like this:

```
alert(require('myModule').my.qualified.packagename.foo())
```

When using `plain` as a module system setting, the call site would be:

```
alert(myModule.my.qualified.packagename.foo());
```

When targeting ECMAScript Modules (ESM), package information is not preserved to improve the application bundle size and match the typical layout of ESM packages. In this case, the consumption of the Kotlin declarations with ES modules looks like this:

```
import { foo } from 'myModule';

alert(foo());
```

`@JsName` annotation

In some cases (for example, to support overloads), the Kotlin compiler mangles the names of generated functions and attributes in JavaScript code. To control the generated names, you can use the `@JsName` annotation:

```
// Module 'kjs'
class Person(val name: String) {
    fun hello() {
        println("Hello $name!")
    }

    @JsName("helloWithGreeting")
    fun hello(greeting: String) {
        println("$greeting $name!")
    }
}
```

Now you can use this class from JavaScript in the following way:

```
// If necessary, import 'kjs' according to chosen module system
var person = new kjs.Person("Dmitry");   // refers to module 'kjs'
person.hello();                          // prints "Hello Dmitry!"
person.helloWithGreeting("Servus");      // prints "Servus Dmitry!"
```

If we didn't specify the `@JsName` annotation, the name of the corresponding function would contain a suffix calculated from the function signature, for example `hello_61zpoe$`.

Note that there are some cases in which the Kotlin compiler does not apply mangling:

- `external` declarations are not mangled.
- Any overridden functions in non-`external` classes inheriting from `external` classes are not mangled.

The parameter of `@JsName` is required to be a constant string literal which is a valid identifier. The compiler will report an error on any attempt to pass a non-identifier string to `@JsName`. The following example produces a compile-time error:

```
@JsName("new C()")   // error here
external fun newC()
```

Experimental

`@JsExport` annotation

By applying the `@JsExport` annotation to a top-level declaration (like a class, interface, or function), you can make Kotlin declarations available from JavaScript or TypeScript. The annotation exports all nested declarations with the name given in Kotlin.

For example, here's how you can export a Kotlin interface with a nested class and a named companion object:

```kotlin
@JsExport
interface Identity {
     class Metadata(val tag: String)

    companion object Registry {
        val defaultTag = "GUEST"
    }
}
```

Currently, the `@JsExport` annotation is the only way to make your functions visible from Kotlin.

The `@JsExport` annotation is also available:

- In common code in multiplatform projects. It only has an effect when compiling for the JavaScript target and allows you to also export Kotlin declarations that are not platform-specific.
- Together with the `@JsName` annotation to specify the names for the generated and exported functions. This helps to resolve ambiguities in exports (like overloads for functions with the same name).
- At the file level using `@file:JsExport`.

Export value classes

You can export Kotlin's inline value classes as regular TypeScript classes.

To export a value class, mark it with the `@JsExport` annotation on the Kotlin side:

```kotlin
// Kotlin
@JsExport
@JvmInline
value class Email(val address: String) {
    init { require(address.contains("@")) { "Invalid email" } }
}

@JsExport
class AuthService {
    suspend fun login(email: Email): String = ...
}
```

On the TypeScript side, it looks like a regular class:

```typescript
// TypeScript
import { AuthService, Email } from "..."
const auth = new AuthService();

console.log(await auth.login(new Email("jane@example.com")));
// "Welcome, jane@example.com!"
console.log(await auth.login(new Email("not-an-email")));
// "Invalid email"
```

Export suspending lambdas

You can export Kotlin's suspending lambda expressions as JavaScript async functions:

1. To enable this feature, add the following compiler option to your `build.gradle.kts` file:

```
kotlin {
    js {
        compilations.all {
            compileTaskProvider.configure {
                compilerOptions {
                    freeCompilerArgs.add("-Xsuspend-lambda-exporting")
                }
            }
        }
    }
}
```

2. Mark the relevant Kotlin declarations with the `@JsExport` annotation:

```
// Kotlin
@JsExport
class TaskRunner {
    suspend fun runTask(task: suspend () -> String): String {
        return task()
    }
}
```

3. On the TypeScript side, the `suspend` lambda will be mapped to a regular `async` function:

```
// TypeScript
import { TaskRunner } from "..."

const runner = new TaskRunner();
const result = await runner.runTask(async () => "done");
console.log(result); // "done"
```

`@JsNoRuntime` annotation

You can export Kotlin interfaces to JavaScript/TypeScript with the `@JsNoRuntime` annotation. It allows for direct mapping to regular TypeScript interfaces.

To export a Kotlin interface, for example, from a Kotlin Multiplatform project:

1. Annotate the Kotlin interface with `@JsNoRuntime` in common code:

```
// commonMain
import kotlin.js.JsNoRuntime

@JsNoRuntime
expect interface DataProcessor {
    fun process(data: String): Int
}
```

2. Provide the actual implementation with `@JsNoRuntime` in your JS-specific source code:

```
// jsMain
import kotlin.js.JsNoRuntime

@JsNoRuntime
actual interface DataProcessor {
    actual fun process(data: String): Int
}
```

3. On the TypeScript side, the interface will be mapped to a regular TypeScript interface:

```
// Generated .d.ts
export interface DataProcessor {
    process(data: string): number;
}
```

For Kotlin Multiplatform projects, the general rules are:

- Both `expect` and `actual` interface declarations must be annotated with `@JsNoRuntime`. The only exception is `external` implementations in platform-specific code on the `actual` side that require no annotation.
- Using `external` interface declarations in common code on the `expect` side is prohibited. Instead, use regular interfaces annotated with `@JsNoRuntime`.

Exporting Kotlin interfaces with `@JsNoRuntime` has some restrictions. The annotation isn't allowed with:

- `external` interfaces as they already behave as if they have `@JsNoRuntime` by default. Adding it results in a compiler warning.
- `is` and `as` type checks.
- Class references that use the `::class` syntax.
- Interfaces that are passed as reified type argument.

Experimental

`@JsStatic`

The `@JsStatic` annotation instructs the compiler to generate additional static methods for the target declaration. This helps you use static members from your Kotlin code directly in JavaScript.

You can apply the `@JsStatic` annotation to functions defined in named objects, as well as in companion objects declared inside classes and interfaces. If you use this annotation, the compiler will generate both a static method of the object and an instance method in the object itself. For example:

```
// Kotlin
class C {
    companion object {
        @JsStatic
        fun callStatic() {}
        fun callNonStatic() {}
    }
}
```

Now, the `callStatic()` function is static in JavaScript while the `callNonStatic()` function is not:

```
// JavaScript
C.callStatic();              // Works, accessing the static function
C.callNonStatic();           // Error, not a static function in the generated JavaScript
C.Companion.callStatic();    // Instance method remains
C.Companion.callNonStatic(); // The only way it works
```

It's also possible to apply the `@JsStatic` annotation to a property of an object or a companion object, making its getter and setter methods static members in that object or the class containing the companion object.

This feature is Experimental. Share your feedback in our issue tracker, YouTrack.

Experimental

Use `BigInt` type to represent Kotlin's `Long` type

Kotlin/JS uses JavaScript's built-in `BigInt` type to represent Kotlin `Long` values when compiling to modern JavaScript (ES2020).

To enable support for the `BigInt` type, you need to add the following compiler option to your `build.gradle(.kts)` file:

```
// build.gradle.kts
kotlin {
    js {
        ...
        compilerOptions {
            freeCompilerArgs.add("-Xes-long-as-bigint")
        }
    }
}
```

This feature is Experimental. Share your feedback in our issue tracker, YouTrack.

Use `Long` in exported declarations

Since Kotlin's `Long` type can compile to JavaScript's `BigInt` type, Kotlin/JS supports exporting `Long` values to JavaScript.

To enable this feature:

1. Allow exporting `Long` in Kotlin/JS. Add the following compiler option to the `freeCompilerArgs` attribute in your `build.gradle(.kts)` file:

   ```
   // build.gradle.kts
   kotlin {
      js {
          ...
          compilerOptions {
              freeCompilerArgs.add("-XXLanguage:+JsAllowLongInExportedDeclarations")
          }
      }
   }
   ```

2. Enable the `BigInt` type. See how to enable it in Use `BigInt` type to represent Kotlin's `Long` type.

Experimental

Use `BigInt64Array` type to represent Kotlin's `LongArray` type

Kotlin/JS can use JavaScript's built-in `BigInt64Array` type to represent Kotlin's `LongArray` values when compiling to JavaScript.

To enable support for the `BigInt64Array` type, add the following compiler option to your `build.gradle(.kts)` file:

```
// build.gradle.kts
kotlin {
    js {
        ...
        compilerOptions {
            freeCompilerArgs.add("-Xes-long-as-bigint")
        }
    }
}
```

This feature is Experimental. Share your feedback in our issue tracker, YouTrack.

Kotlin types in JavaScript

See how Kotlin types are mapped to JavaScript ones:

Kotlin	JavaScript	Comments
`Byte`, `Short`, `Int`, `Float`, `Double`	`Number`	
`Char`	`Number`	The number represents the character's code.
`Long`	`BigInt`	Needs the `-Xes-long-as-bigint` compiler option configured.
`Boolean`	`Boolean`	
`String`	`String`	
`Array`	`Array`	
`ByteArray`	`Int8Array`	
`ShortArray`	`Int16Array`	
`IntArray`	`Int32Array`	
`CharArray`	`UInt16Array`	Carries the property `$type$ == "CharArray"`.
`FloatArray`	`Float32Array`	
`DoubleArray`	`Float64Array`	
`LongArray`	`BigInt64Array`	

Kotlin	JavaScript	Comments
`BooleanArray`	`Int8Array`	Carries the property `$type$ == "BooleanArray"`.
`List`, `MutableList`	`KtList`, `KtMutableList`	Exposes an `Array` via `KtList.asJsReadonlyArrayView` or `KtMutableList.asJsArrayView`.
`Map`, `MutableMap`	`KtMap`, `KtMutableMap`	Exposes an ES2015 `Map` via `KtMap.asJsReadonlyMapView` or `KtMutableMap.asJsMapView`.
`Set`, `MutableSet`	`KtSet`, `KtMutableSet`	Exposes an ES2015 `Set` via `KtSet.asJsReadonlySetView` or `KtMutableSet.asJsSetView`.
`Unit`	Undefined	Exportable when used as return type, but not when used as parameter type.
`Any`	`Object`	
`Throwable`	`Error`	
`enum class Type`	`Type`	Enum entries are exposed as static class properties (`Type.ENTRY`).
Nullable `Type?`	`Type | null | undefined`	
All other Kotlin types, except for those marked with `@JsExport`	Not supported	Includes Kotlin's unsigned integer types.

Additionally, it is important to know that:

- Kotlin preserves overflow semantics for `kotlin.Int`, `kotlin.Byte`, `kotlin.Short`, `kotlin.Char` and `kotlin.Long`.
- Kotlin cannot distinguish between numeric types at runtime (except for `kotlin.Long`), so the following code works:

```
fun f() {
    val x: Int = 23
    val y: Any = x
    println(y as Float)
}
```

- Kotlin preserves lazy object initialization in JavaScript.

JavaScript modules

You can compile your Kotlin projects to JavaScript modules for various popular module systems. We currently support the following configurations for JavaScript modules:

- ES Modules, the standard way to declare a module in JavaScript (with the `import/export` JavaScript syntax). It's used by default if the `target` is set to `es2015`.
- Unified Module Definitions (UMD), which is compatible with both *AMD* and *CommonJS*. UMD modules are also able to be executed without being imported or when no module system is present. This is the default option for the `browser` and `nodejs` targets.
- Asynchronous Module Definitions (AMD), which is in particular used by the RequireJS library.
- CommonJS, widely used by Node.js/npm (`require` function and `module.exports` object).
- Plain. Don't compile for any module system. You can access a module by its name in the global scope.

Browser targets

If you intend to run your code in a web browser environment and want to use a module system other than UMD, you can specify the desired module type in the `webpackTask` configuration block. For example, to switch to CommonJS, use:

```
kotlin {
    js {
        browser {
            webpackTask {
                output.libraryTarget = "commonjs2"
            }
        }
        binaries.executable()
    }
}
```

Webpack provides two different flavors of CommonJS, `commonjs` and `commonjs2`, which affect the way your declarations are made available. In most cases, you probably want `commonjs2`, which adds the `module.exports` syntax to the generated library. Alternatively, you can also opt for the `commonjs` option, which adheres strictly to the CommonJS specification. To learn more about the difference between `commonjs` and `commonjs2`, see the Webpack repository.

JavaScript libraries and Node.js files

If you are creating a library for use in JavaScript or Node.js environments, and want to use a different module system, the instructions are slightly different.

Choose the target module system

To select the target module system, set the `moduleKind` compiler option in the Gradle build script:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.targets.js.ir.KotlinJsIrLink> {
    compilerOptions.moduleKind.set(org.jetbrains.kotlin.gradle.dsl.JsModuleKind.MODULE_COMMONJS)
}
```

Groovy

```
compileKotlinJs.compilerOptions.moduleKind =
org.jetbrains.kotlin.gradle.dsl.JsModuleKind.MODULE_COMMONJS
```

The available values are: `umd` (default), `es`, `commonjs`, `amd`, `plain`.

This is different from adjusting `webpackTask.output.libraryTarget`. The library target changes the output *generated by webpack* (after your code has already been compiled). `compilerOptions.moduleKind` changes the output generated *by the Kotlin compiler*.

In the Kotlin Gradle DSL, there is also a shortcut for setting the CommonJS and ESM module kinds:

```
kotlin {
    js {
        useCommonJs()
        // OR
        useEsModules()
        // ...
    }
}
```

@JsModule annotation

To tell Kotlin that an `external` class, package, function or property is a JavaScript module, you can use `@JsModule` annotation. Consider you have the following CommonJS module called "hello":

```
module.exports.sayHello = function (name) { alert("Hello, " + name); }
```

You should declare it like this in Kotlin:

```
@JsModule("hello")
external fun sayHello(name: String)
```

Apply @JsModule to packages

Some JavaScript libraries export packages (namespaces) instead of functions and classes. In terms of JavaScript, it's an *object* that has *members* that are classes, functions, and properties. Importing these packages as Kotlin objects often looks unnatural. The compiler can map imported JavaScript packages to Kotlin packages, using the following notation:

```
@file:JsModule("extModule")

package ext.jspackage.name

external fun foo()

external class C
```

Where the corresponding JavaScript module is declared like this:

```
module.exports = {
  foo: { /* some code here */ },
  C: { /* some code here */ }
}
```

Files marked with `@file:JsModule` annotation can't declare non-external members. The example below produces a compile-time error:

```
@file:JsModule("extModule")

package ext.jspackage.name

external fun foo()

fun bar() = "!" + foo() + "!" // error here
```

Import deeper package hierarchies

In the previous example, the JavaScript module exports a single package. However, some JavaScript libraries export multiple packages from within a module. This case is also supported by Kotlin, though you have to declare a new `.kt` file for each package you import.

For example, let's make the example a bit more complicated:

```
module.exports = {
  mylib: {
    pkg1: {
      foo: function () { /* some code here */ },
      bar: function () { /* some code here */ }
    },
    pkg2: {
      baz: function () { /* some code here */ }
    }
  }
}
```

To import this module in Kotlin, you have to write two Kotlin source files:

```
@file:JsModule("extModule")
@file:JsQualifier("mylib.pkg1")

package extlib.pkg1

external fun foo()

external fun bar()
```

and

```
@file:JsModule("extModule")
@file:JsQualifier("mylib.pkg2")

package extlib.pkg2

external fun baz()
```

@JsNonModule annotation

When a declaration is marked as `@JsModule`, you can't use it from Kotlin code when you don't compile it to a JavaScript module. Usually, developers distribute their libraries both as JavaScript modules and downloadable `.js` files that you can copy to your project's static resources and include via a `<script>` tag. To tell Kotlin that it's okay to use a `@JsModule` declaration from a non-module environment, add the `@JsNonModule` annotation. For example, consider the following JavaScript code:

```
function topLevelSayHello (name) { alert("Hello, " + name); }

if (module && module.exports) {
  module.exports = topLevelSayHello;
}
```

You could describe it from Kotlin as follows:

```
@JsModule("hello")
@JsNonModule
@JsName("topLevelSayHello")
external fun sayHello(name: String)
```

Module system used by the Kotlin Standard Library

Kotlin is distributed with the Kotlin/JS standard library as a single file, which is itself compiled as an UMD module, so you can use it with any module system described above. For most use cases of Kotlin/JS, it is recommended to use a Gradle dependency on `kotlin-stdlib-js`, which is also available on NPM as the `kotlin` package.

Kotlin/JS reflection

Kotlin/JS provides a limited support for the Kotlin reflection API. The only supported parts of the API are:

- Class references (`::class`)
- `KType` and `typeof()`
- `KClass` and `createInstance()`

Class references

The `::class` syntax returns a reference to the class of an instance, or the class corresponding to the given type. In Kotlin/JS, the value of a `::class` expression is a stripped-down KClass implementation that supports only:

- simpleName and isInstance() members.
- cast() and safeCast() extension functions.

In addition to that, you can use KClass.js to access the JsClass instance corresponding to the class. The `JsClass` instance itself is a reference to the constructor function. This can be used to interoperate with JS functions that expect a reference to a constructor.

KType and typeOf()

The `typeof()` function constructs an instance of `KType` for a given type. The `KType` API is fully supported in Kotlin/JS except for Java-specific parts.

KClass and createInstance()

The `createInstance()` function from the KClass interface creates a new instance of the specified class, which is useful for getting the runtime reference to a Kotlin class.

Example

Here is an example of the reflection usage in Kotlin/JS.

```
open class Shape
class Rectangle : Shape()

inline fun <reified T> accessReifiedTypeArg() =
    println(typeOf<T>().toString())

fun main() {
    val s = Shape()
    val r = Rectangle()

    println(r::class.simpleName) // Prints "Rectangle"
    println(Shape::class.simpleName) // Prints "Shape"
    println(Shape::class.js.name) // Prints "Shape"

    println(Shape::class.isInstance(r)) // Prints "true"
    println(Rectangle::class.isInstance(s)) // Prints "false"
    val rShape = Shape::class.cast(r) // Casts a Rectangle "r" to Shape

    accessReifiedTypeArg<Rectangle>() // Accesses the type via typeOf(). Prints "Rectangle"
}
```

Typesafe HTML DSL

The kotlinx.html library provides the ability to generate DOM elements using statically typed HTML builders (and besides JavaScript, it is even available on the JVM target!) To use the library, include the corresponding repository and dependency to our `build.gradle.kts` file:

```
repositories {
    // ...
    mavenCentral()
}

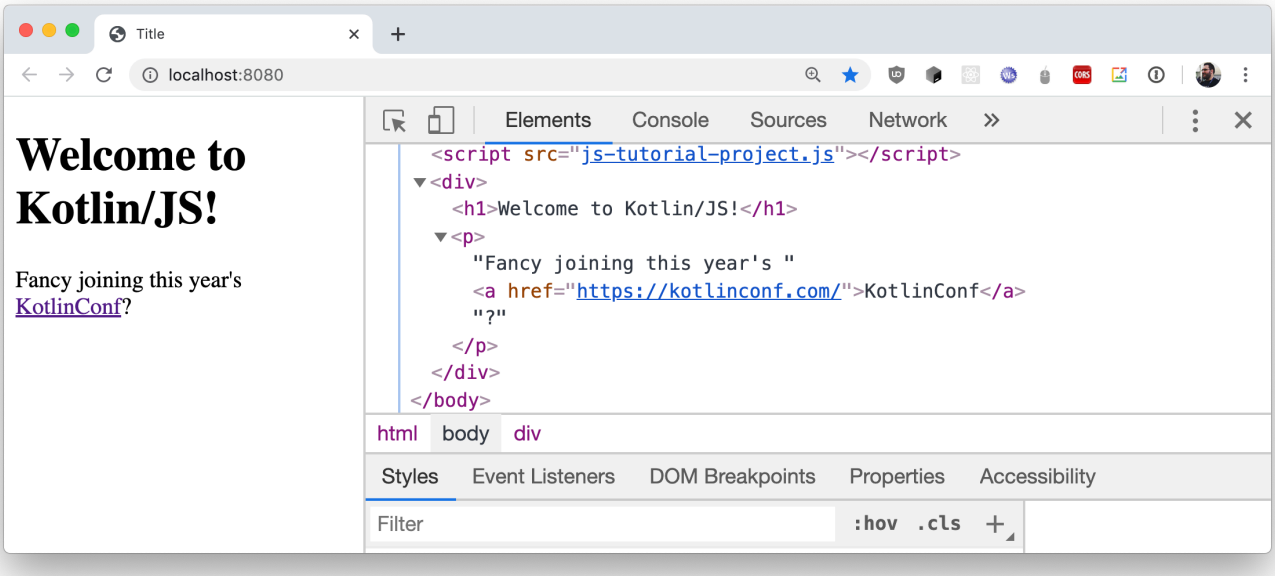
dependencies {
    implementation(kotlin("stdlib-js"))
    implementation("org.jetbrains.kotlinx:kotlinx-html-js:0.8.0")
    // ...
}
```

Once the dependency is included, you can access the different interfaces provided to generate the DOM. To render a headline, some text, and a link, the following snippet would be sufficient, for example:

```
import kotlinx.browser.*
import kotlinx.html.*
import kotlinx.html.dom.*

fun main() {
    document.body!!.append.div {
        h1 {
            +"Welcome to Kotlin/JS!"
        }
        p {
            +"Fancy joining this year's "
            a("https://kotlinconf.com/") {
                +"KotlinConf"
            }
            +"?"
        }
    }
}
```

When running this example in the browser, the DOM will be assembled in a straightforward way. This is easily confirmed by checking the Elements of the website using the developer tools of our browser:



Rendering a website from kotlinx.html

To learn more about the `kotlinx.html` library, check out the GitHub Wiki, where you can find more information about how to create elements without adding them to the DOM, binding to events like `onClick`, and examples on how to apply CSS classes to your HTML elements, to name just a few.

Interoperability with Swift/Objective-C

> The Objective-C libraries import is in Beta. All Kotlin declarations generated by the cinterop tool from Objective-C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.

Kotlin/Native provides indirect interoperability with Swift through Objective-C. This document covers how you can use Kotlin declarations in Swift/Objective-C code and Objective-C declarations in Kotlin code.

Some other resources you might find useful:

- The Kotlin-Swift interopedia, a collection of examples on how to use Kotlin declarations in Swift code.
- The Integration with Swift/Objective-C ARC section, covering the details of integration between Kotlin's tracing GC and Objective-C's ARC.

Importing Swift/Objective-C libraries to Kotlin

Objective-C frameworks and libraries can be used in Kotlin code if properly imported to the build (system frameworks are imported by default). For more details, see:

- Create and configure a library definition file
- Configure compilation for native libraries

A Swift library can be used in Kotlin code if its API is exported to Objective-C with `@objc`. Pure Swift modules are not yet supported.

Using Kotlin in Swift/Objective-C

Kotlin modules can be used in Swift/Objective-C code if compiled into a framework:

- See Build final native binaries to see how to declare binaries.
- Check out the Kotlin Multiplatform sample project for an example.

Experimental

Hide Kotlin declarations from Objective-C and Swift

To make your Kotlin code more Swift/Objective-C-friendly, use the `@HiddenFromObjC` annotation to hide a Kotlin declaration from Objective-C and Swift. It disables the function or property export to Objective-C.

Alternatively, you can mark Kotlin declarations with the `internal` modifier to restrict their visibility in the compilation module. Use `@HiddenFromObjC` if you want to hide the Kotlin declaration from Objective-C and Swift while keeping it visible to other Kotlin modules.

See an example in the Kotlin-Swift interopedia.

Experimental

Use refining in Swift

`@ShouldRefineInSwift` helps to replace a Kotlin declaration with a wrapper written in Swift. The annotation marks a function or property as `swift_private` in the generated Objective-C API. Such declarations get the `__` prefix, which makes them invisible from Swift.

You can still use these declarations in your Swift code to create a Swift-friendly API, but they won't be suggested in the Xcode autocomplete.

- For more information on refining Objective-C declarations in Swift, see the official Apple documentation.
- For an example on how to use the `@ShouldRefineInSwift` annotation, see the Kotlin-Swift interopedia.

Experimental

Change declaration names

To avoid renaming Kotlin declarations, use the `@ObjCName` annotation. It instructs the Kotlin compiler to use the custom Objective-C and Swift name for the annotated class, interface, or another Kotlin entity:

```
@ObjCName(swiftName = "MySwiftArray")
class MyKotlinArray {
    @ObjCName("index")
    fun indexOf(@ObjCName("of") element: String): Int = TODO()
}

// Usage with the ObjCName annotations
let array = MySwiftArray()
let index = array.index(of: "element")
```

See another example in the Kotlin-Swift interopedia.

Provide documentation with KDoc comments

Documentation is essential for understanding any API. Providing documentation for the shared Kotlin API allows you to communicate with its users on matters of usage, dos and don'ts, and so on.

When generating Objective-C headers, KDoc comments from Kotlin code are translated into corresponding Objective-C comments. For example, the following Kotlin code with KDoc:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit integer.
 */
fun printSum(a: Int, b: Int) = println(a.toLong() + b)
```

Will produce an Objective-C header with the corresponding comment:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit integer.
 */
+ (void)printSumA:(int32_t)a b:(int32_t)b __attribute__((swift_name("printSum(a:b:)")));
```

KDoc comments are embedded into klibs and extracted from klibs into the produced Apple frameworks. As a result, comments on classes and methods appear during autocompletion, for example in Xcode. If you go to the definition of functions in the `.h` file, you'll see comments on `@param`, `@return`, and similar tags.

Known limitations:

- Dependency documentation is not exported unless it is compiled with the `-Xexport-kdoc` option. Libraries compiled with this compiler option might be incompatible with other compiler versions.
- KDoc comments are mostly exported as is, but many KDoc block tags, such as `@property`, are not supported.

If necessary, you can disable the export of KDoc comments from klibs to the produced Apple frameworks in the `binaries {}` block of your Gradle build file:

```
// build.gradle.kts
import org.jetbrains.kotlin.gradle.ExperimentalKotlinGradlePluginApi

kotlin {
    iosArm64 {
        binaries {
            framework {
                baseName = "sdk"
                @OptIn(ExperimentalKotlinGradlePluginApi::class)
                exportKdoc.set(false)
            }
        }
    }
}
```

Mappings

The table below shows how Kotlin concepts are mapped to Swift/Objective-C and vice versa.

"->" and "<-" indicate that mapping only goes one way.

Kotlin	Swift	Objective-C	Notes
`class`	`class`	`@interface`	note
`interface`	`protocol`	`@protocol`	
`constructor` / `create`	Initializer	Initializer	note
Property	Property	Property	note 1, note 2
Method	Method	Method	note 1, note 2
`enum class`	`class`	`@interface`	note
`suspend` ->	`completionHandler:` / `async`	`completionHandler:`	note 1, note 2
`@Throws fun`	`throws`	`error:(NSError**)error`	note
Extension	Extension	Category member	note

Kotlin	Swift	Objective-C	Notes
`companion` member <-	Class method or property	Class method or property	
`null`	`nil`	`nil`	
`Singleton`	`shared` or `companion` property	`shared` or `companion` property	note
Primitive type	Primitive type / `NSNumber`		note
`Unit` return type	`Void`	`void`	
`String`	`String`	`NSString`	note
`String`	`NSMutableString`	`NSMutableString`	note
`List`	`Array`	`NSArray`	
`MutableList`	`NSMutableArray`	`NSMutableArray`	
`Set`	`Set`	`NSSet`	
`MutableSet`	`NSMutableSet`	`NSMutableSet`	note
`Map`	`Dictionary`	`NSDictionary`	
`MutableMap`	`NSMutableDictionary`	`NSMutableDictionary`	note
Function type	Function type	Block pointer type	note
Inline classes	Unsupported	Unsupported	note

Classes

Name translation

Objective-C classes are imported into Kotlin with their original names. Protocols are imported as interfaces with a `Protocol` name suffix, for example, `@protocol Foo` -> `interface FooProtocol`. These classes and interfaces are placed into a package specified in build configuration (`platform.*` packages for preconfigured system frameworks).

The names of Kotlin classes and interfaces are prefixed when imported to Objective-C. The prefix is derived from the framework name.

Objective-C does not support packages in a framework. If the Kotlin compiler finds Kotlin classes in the same framework which have the same name but different packages, it renames them. This algorithm is not stable yet and can change between Kotlin releases. To work around this, you can rename the conflicting Kotlin classes in the framework.

Strong linking

Whenever you use an Objective-C class in the Kotlin source, it's marked as a strongly linked symbol. The resulting build artifact mentions related symbols as strong external references.

This means that the app tries to link symbols during the launch dynamically, and if they are unavailable, the app crashes. The crash happens even if symbols were never used. Symbols might be unavailable on a particular device or OS version.

To work around this issue and avoid "Symbol not found" errors, use a Swift or Objective-C wrapper that checks if the class is actually available. See how this workaround was implemented in the Compose Multiplatform framework.

Initializers

A Swift/Objective-C initializer is imported to Kotlin as constructors or as factory methods named `create`. The latter happens with initializers declared in the Objective-C category or as a Swift extension, because Kotlin has no concept of extension constructors.

> Before importing Swift initializers to Kotlin, don't forget to annotate them with `@objc`.

Kotlin constructors are imported as initializers to Swift/Objective-C.

Setters

Writeable Objective-C properties overriding read-only properties of the superclass are represented as the `setFoo()` method for the property `foo`. The same goes for a protocol's read-only properties that are implemented as mutable.

Top-level functions and properties

Top-level Kotlin functions and properties are accessible as members of special classes. Each Kotlin file is translated into such a class, for example:

```
// MyLibraryUtils.kt
package my.library

fun foo() {}
```

You can then call the `foo()` function from Swift like this:

```
MyLibraryUtilsKt.foo()
```

See a collection of examples on accessing top-level Kotlin declarations in the Kotlin-Swift interopedia:

- Top-level functions
- Top-level read-only properties

- Top-level mutable properties

Method names translation

Generally, Swift argument labels and Objective-C selector pieces are mapped to Kotlin parameter names. These two concepts have different semantics, so sometimes Swift/Objective-C methods can be imported with a clashing Kotlin signature. In this case, the clashing methods can be called from Kotlin using named arguments, for example:

```
[player moveTo:LEFT byMeters:17]
[player moveTo:UP byInches:42]
```

In Kotlin, it's:

```
player.moveTo(LEFT, byMeters = 17)
player.moveTo(UP, byInches = 42)
```

Here's how the `kotlin.Any` functions are mapped to Swift/Objective-C:

Kotlin	Swift	Objective-C
`equals()`	`isEquals(_:)`	`isEquals:`
`hashCode()`	`hash`	`hash`
`toString()`	`description`	`description`

See an example with data classes in the Kotlin-Swift interopedia.

You can specify a more idiomatic name in Swift or Objective-C, instead of renaming the Kotlin declaration with the `@ObjCName` annotation.

Errors and exceptions

All Kotlin exceptions are unchecked, meaning that errors are caught at runtime. However, Swift has only checked errors that are handled at compile time. So, if Swift or Objective-C code calls a Kotlin method that throws an exception, the Kotlin method should be marked with the `@Throws` annotation, specifying a list of "expected" exception classes.

When compiling to the Swift/Objective-C framework, non-`suspend` functions that have or inherit the `@Throws` annotation are represented as `NSError*`-producing methods in Objective-C and as `throws` methods in Swift. Representations for `suspend` functions always have an `NSError*` / `Error` parameter in the completion handler.

When a Kotlin function called from Swift/Objective-C code throws an exception which is an instance of one of the classes specified with `@Throws` or their subclasses, the exception is propagated as an `NSError`. Other Kotlin exceptions reaching Swift/Objective-C are considered unhandled and cause program termination.

`suspend` functions without `@Throws` propagate only `CancellationException` (as `NSError`). Non-`suspend` functions without `@Throws` don't propagate Kotlin exceptions at all.

Note that the opposite reversed translation is not implemented yet: Swift/Objective-C error-throwing methods aren't imported to Kotlin as exception-throwing.

See an example in the Kotlin-Swift interopedia.

Enums

Kotlin enums are imported into Objective-C as `@interface` and into Swift as `class`. These data structures have properties corresponding to each enum value. Consider this Kotlin code:

```
// Kotlin
enum class Colors {
    RED, GREEN, BLUE
}
```

You can access the properties of this enum class from Swift as follows:

```
// Swift
Colors.red
Colors.green
Colors.blue
```

To use variables of a Kotlin enum in a Swift `switch` statement, provide a default statement to prevent a compilation error:

```
switch color {
    case .red: print("It's red")
    case .green: print("It's green")
    case .blue: print("It's blue")
    default: fatalError("No such color")
}
```

See another example in the Kotlin-Swift interopedia.

Experimental

Suspending functions

Kotlin's suspending functions (`suspend`) are presented in the generated Objective-C headers as functions with callbacks, or completion handlers in Swift/Objective-C terminology.

Starting from Swift 5.5, Kotlin's `suspend` functions are also available for calling from Swift as `async` functions without using the completion handlers. Currently, this functionality is highly experimental and has certain limitations. See this YouTrack issue for details.

- Learn more about the `async` / `await` mechanism in the Swift documentation.
- See an example and recommendations on third-party libraries that implement the same functionality in the Kotlin-Swift interopedia.

Extensions and category members

Members of Objective-C categories and Swift extensions are generally imported to Kotlin as extensions. That's why these declarations can't be overridden in Kotlin, and the extension initializers aren't available as Kotlin constructors.

Currently, there are two exceptions. Starting with Kotlin 1.8.20, category members that are declared in the same headers as the NSView class (from the AppKit framework) or UIView classes (from the UIKit framework) are imported as members of these classes. This means that you can override methods that subclass from NSView or UIView.

Kotlin extensions to "regular" Kotlin classes are imported to Swift and Objective-C as extensions and category members, respectively. Kotlin extensions to other types are treated as top-level declarations with an additional receiver parameter. These types include:

- Kotlin `String` type
- Kotlin collection types and subtypes
- Kotlin `interface` types
- Kotlin primitive types
- Kotlin `inline` classes
- Kotlin `Any` type
- Kotlin function types and subtypes
- Objective-C classes and protocols

See a collection of examples in the Kotlin-Swift interopedia.

Kotlin singletons

Kotlin singleton (made with an `object` declaration, including `companion object`) is imported to Swift/Objective-C as a class with a single instance.

The instance is available through the `shared` and `companion` properties.

For the following Kotlin code:

```
object MyObject {
    val x = "Some value"
}

class MyClass {
    companion object {
        val x = "Some value"
    }
}
```

Access these objects as follows:

```
MyObject.shared
MyObject.shared.x
MyClass.companion
MyClass.Companion.shared
```

Access objects through `[MySingleton mySingleton]` in Objective-C and `MySingleton()` in Swift has been deprecated.

See more examples in the Kotlin-Swift interopedia:

- How to access Kotlin objects using `shared`
- How to access members of Kotlin companion objects from Swift.

Primitive types

Kotlin primitive type boxes are mapped to special Swift/Objective-C classes. For example, the `kotlin.Int` box is represented as the `KotlinInt` class instance in Swift (or the `${prefix}Int` instance in Objective-C, where `prefix` is the framework's name prefix). These classes are derived from `NSNumber`, so the instances are proper `NSNumber` s supporting all corresponding operations.

The `NSNumber` type is not automatically translated to Kotlin primitive types when used as a Swift/Objective-C parameter type or return value. The reason is that the `NSNumber` type doesn't provide enough information about a wrapped primitive value type, for example, `NSNumber` is statically not known to be `Byte`, `Boolean`, or `Double`. So Kotlin primitive values should be cast to and from `NSNumber` manually.

Strings

When a Kotlin `String` is passed to Swift, it's first exported as an Objective-C object, and then the Swift compiler copies it one more time for a Swift conversion. This results in additional runtime overhead.

To avoid that, access Kotlin strings in Swift directly as an Objective-C `NSString` instead. See the conversion example.

NSMutableString

`NSMutableString` Objective-C class is not available from Kotlin. All instances of `NSMutableString` are copied when passed to Kotlin.

Collections

Kotlin -> Objective-C -> Swift

When a Kotlin collection is passed to Swift, it's first converted to an Objective-C equivalent, and then the Swift compiler copies the entire collection and converts it into a Swift-native collection as described in the mappings table.

This last conversion leads to performance costs. To prevent this, when using Kotlin collections in Swift, explicitly cast them to their Objective-C counterparts: `NSDictionary`, `NSArray`, or `NSSet`.

See the conversion example

For example, the following Kotlin declaration:

```
val map: Map<String, String>
```

In Swift, it looks like this:

```
map[key]?.count ?? 0
```

Here, the `map` is implicitly converted to Swift's `Dictionary`, and its string values are mapped to Swift's `String`. This results in a performance cost.

To avoid the conversion, explicitly cast `map` to Objective-C's `NSDictionary` and access values as `NSString` instead:

```
let nsMap: NSDictionary = map as NSDictionary
(nsMap[key] as? NSString)?.length ?? 0
```

This ensures that the Swift compiler doesn't perform an additional conversion step.

Swift -> Objective-C -> Kotlin

Swift/Objective-C collections are mapped to Kotlin as described in the mappings table, except for `NSMutableSet` and `NSMutableDictionary`.

`NSMutableSet` isn't converted to a Kotlin's `MutableSet`. To pass an object to Kotlin `MutableSet`, explicitly create this kind of Kotlin collection. To do this, use, for example, the `mutableSetOf()` function in Kotlin or the `KotlinMutableSet` class in Swift and `${prefix}MutableSet` in Objective-C (`prefix` is the framework names prefix). The same is true for `MutableMap`.

See an example in the Kotlin-Swift interopedia.

Function types

Kotlin function-typed objects (for example, lambdas) are converted to closures in Swift and blocks in Objective-C. See an example of a Kotlin function with a lambda in the Kotlin-Swift interopedia.

However, there is a difference in how types of parameters and return values are mapped when translating a function and a function type. In the latter case, primitive types are mapped to their boxed representation. Kotlin `Unit` return value is represented as a corresponding `Unit` singleton in Swift/Objective-C. The value of this singleton can be retrieved the same way as for any other Kotlin `object`. See singletons in the table above.

Consider the following Kotlin function:

```
fun foo(block: (Int) -> Unit) { ... }
```

It's represented in Swift as follows:

```
func foo(block: (KotlinInt) -> KotlinUnit)
```

And you can call it like this:

```
foo {
    bar($0 as! Int32)
    return KotlinUnit()
}
```

Experimental

Explicit parameter names in Objective-C block types

You can add explicit parameter names to Kotlin's function types for exported Objective-C headers. Xcode's autocompletion then suggests these names when calling Objective-C functions in the Objective-C block. This helps to avoid Clang warnings in generated blocks.

To enable explicit parameter names, add the following binary option to your `gradle.properties` file:

```
kotlin.native.binary.objcExportBlockExplicitParameterNames=true
```

For example, for the following Kotlin code:

```
// Kotlin:
fun greetUser(block: (name: String) -> Unit) = block("John")
```

Kotlin forwards the parameter names from Kotlin function types to Objective-C block types, allowing Xcode to use them in suggestions:

```
// Objective-C:
greetUserBlock:^(NSString *name) {
    // ...
};
```

This option affects Objective-C interop only. It applies when calling generated Objective-C code from Objective-C in Xcode, and generally doesn't affect calls from Swift.

Generics

Objective-C supports "lightweight generics" defined in classes, with a relatively limited feature set. Swift can import generics defined on classes to help provide additional type information to the compiler.

Generic feature support for Objective-C and Swift differ from Kotlin, so the translation will inevitably lose some information, but the features supported retain meaningful information.

For specific examples on how to use Kotlin generics in Swift, see the Kotlin-Swift interopedia.

Limitations

Objective-C generics do not support all features of either Kotlin or Swift, so there will be some information lost in the translation.

Generics can only be defined on classes, not on interfaces (protocols in Objective-C and Swift) or functions.

Nullability

Kotlin and Swift both define nullability as part of the type specification, while Objective-C defines nullability on methods and properties of a type. So, the following Kotlin code:

```
class Sample<T>() {
    fun myVal(): T
}
```

In Swift, it looks like this:

```
class Sample<T>() {
    fun myVal(): T?
}
```

To support a potentially nullable type, the Objective-C header needs to define `myVal` with a nullable return value.

To mitigate this, when defining your generic classes, provide a non-nullable type constraint if the generic type should *never* be null:

```
class Sample<T : Any>() {
    fun myVal(): T
}
```

That will force the Objective-C header to mark `myVal` as non-nullable.

Variance

Objective-C allows generics to be declared covariant or contravariant. Swift has no support for variance. Generic classes coming from Objective-C can be force-cast as needed.

```
data class SomeData(val num: Int = 42) : BaseData()
class GenVarOut<out T : Any>(val arg: T)
```

```
let variOut = GenVarOut<SomeData>(arg: sd)
let variOutAny : GenVarOut<BaseData> = variOut as! GenVarOut<BaseData>
```

Constraints

In Kotlin, you can provide upper bounds for a generic type. Objective-C also supports this, but that support is unavailable in more complex cases and is currently not supported in the Kotlin - Objective-C interop. The exception here being a non-nullable upper bound will make Objective-C methods/properties non-nullable.

To disable

To have the framework header written without generics, add the following compiler option in your build file:

```
binaries.framework {
    freeCompilerArgs += "-Xno-objc-generics"
}
```

Forward declarations

To import forward declarations, use the `objcnames.classes` and `objcnames.protocols` packages. For example, to import a `objcprotocolName` forward declaration declared in an Objective-C library with a `library.package`, use a special forward declaration package: `import objcnames.protocols.objcprotocolName`.

Consider two objcinterop libraries: one that uses `objcnames.protocols.ForwardDeclaredProtocolProtocol` and another with an actual implementation in another package:

```
// First objcinterop library
#import <Foundation/Foundation.h>

@protocol ForwardDeclaredProtocol;

NSString* consumeProtocol(id<ForwardDeclaredProtocol> s) {
    return [NSString stringWithUTF8String:"Protocol"];
}
```

```
// Second objcinterop library
// Header:
#import <Foundation/Foundation.h>
@protocol ForwardDeclaredProtocol
@end
// Implementation:
@interface ForwardDeclaredProtocolImpl : NSObject <ForwardDeclaredProtocol>
@end

id<ForwardDeclaredProtocol> produceProtocol() {
    return [ForwardDeclaredProtocolImpl new];
}
```

To transfer objects between the two libraries, use an explicit `as` cast in your Kotlin code:

```
// Kotlin code:
fun test() {
    consumeProtocol(produceProtocol() as objcnames.protocols.ForwardDeclaredProtocolProtocol)
}
```

> You can only cast to `objcnames.protocols.ForwardDeclaredProtocolProtocol` from the corresponding real class. Otherwise, you'll get an error.

Casting between mapped types

When writing Kotlin code, an object may need to be converted from a Kotlin type to the equivalent Swift/Objective-C type or vice versa. In this case, you can use the `as` cast, for example:

```
@file:Suppress("CAST_NEVER_SUCCEEDS")
import platform.Foundation.*

val nsNumber = 42 as NSNumber
val nsArray = listOf(1, 2, 3) as NSArray
val nsString = "Hello" as NSString
val string = nsString as String
```

IDEs might incorrectly emit "This cast can never succeed" warnings. In such cases, use the `@Suppress("CAST_NEVER_SUCCEEDS")` annotation.

Subclassing

Subclassing Kotlin classes and interfaces from Swift/Objective-C

Kotlin classes and interfaces can be subclassed by Swift/Objective-C classes and protocols.

Subclassing Swift/Objective-C classes and protocols from Kotlin

Swift/Objective-C classes and protocols can be subclassed with a Kotlin `final` class. Non-`final` Kotlin classes inheriting Swift/Objective-C types aren't supported yet, so it is not possible to declare a complex class hierarchy inheriting Swift/Objective-C types.

Normal methods can be overridden using the `override` Kotlin keyword. In this case, the overriding method must have the same parameter names as the overridden one.

Sometimes it is required to override initializers, for example, when subclassing `UIViewController`. Initializers imported as Kotlin constructors can be overridden by Kotlin constructors marked with the `@OverrideInit` annotation:

```
class ViewController : UIViewController {
    @OverrideInit constructor(coder: NSCoder) : super(coder)

    ...
}
```

The overriding constructor must have the same parameter names and types as the overridden one.

To override different methods with clashing Kotlin signatures, you can add the `@ObjCSignatureOverride` annotation to the class. The annotation instructs the Kotlin compiler to ignore conflicting overloads, in case several functions with the same argument types, but different argument names, are inherited from the Objective-C class.

By default, the Kotlin/Native compiler doesn't allow calling a non-designated Objective-C initializer as a `super()` constructor. This behavior can be inconvenient if the designated initializers aren't marked properly in the Objective-C library. To disable these compiler checks, add the `disableDesignatedInitializerChecks = true` to the library's `.def` file.

C features

See Interoperability with C for an example case where the library uses some plain C features, such as unsafe pointers, structs, and so on.

Unsupported

Some features of the Kotlin programming language are not yet mapped into the respective features of Objective-C or Swift. Currently, the following features are not properly exposed in generated framework headers:

- Inline classes (arguments are mapped as either underlying primitive type or `id`)
- Custom classes implementing standard Kotlin collection interfaces (`List`, `Map`, `Set`) and other special classes
- Kotlin subclasses of Objective-C classes

Kotlin/Native as an Apple framework – tutorial

> The Objective-C libraries import is in Beta. All Kotlin declarations generated by the cinterop tool from Objective-C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.

Kotlin/Native provides bidirectional interoperability with Swift/Objective-C. You can both use Objective-C frameworks and libraries in Kotlin code, and Kotlin modules in Swift/Objective-C code.

Kotlin/Native comes with a set of pre-imported system frameworks; it's also possible to import an existing framework and use it from Kotlin. In this tutorial, you'll learn how to create your own framework and use Kotlin/Native code from Swift/Objective-C applications on macOS and iOS.

In this tutorial, you will:

- Create a Kotlin library and compile it to a framework
- Examine the generated Swift/Objective-C API code
- Use the framework from Objective-C
- Use the framework from Swift

You can use the command line to generate a Kotlin framework, either directly or with a script file (such as `.sh` or `.bat` file). However, this approach doesn't scale well for larger projects that have hundreds of files and libraries. Using a build system simplifies the process by downloading and caching the Kotlin/Native compiler binaries and libraries with transitive dependencies, as well as by running the compiler and tests. Kotlin/Native can use the Gradle build system through the Kotlin Multiplatform plugin.



If you use a Mac and want to create and run applications for iOS or other Apple targets, you also need to install the Xcode Command Line Tools, launch it, and accept the license terms first.

Create a Kotlin library



See the Get started with Kotlin/Native tutorial for detailed first steps and instructions on how to create a new Kotlin/Native project and open it in IntelliJ IDEA.

The Kotlin/Native compiler can produce a framework for macOS and iOS from the Kotlin code. The created framework contains all declarations and binaries needed to use it with Swift/Objective-C.

Let's first create a Kotlin library:

1. In the `src/nativeMain/kotlin` directory, create the `lib.kt` file with the library contents:

```kotlin
package example

object Object {
    val field = "A"
}

interface Interface {
    fun iMember() {}
}

class Clazz : Interface {
    fun member(p: Int): ULong? = 42UL
}

fun forIntegers(b: Byte, s: UShort, i: Int, l: ULong?) { }
fun forFloats(f: Float, d: Double?) { }

fun strings(str: String?) : String {
    return "That is '$str' from C"
}

fun acceptFun(f: (String) -> String?) = f("Kotlin/Native rocks!")
fun supplyFun() : (String) -> String? = { "$it is cool!" }
```

2. Update your `build.gradle(.kts)` Gradle build file with the following:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget

plugins {
    kotlin("multiplatform") version "2.4.20"
}

repositories {
    mavenCentral()
}

kotlin {
    iosArm64()
    // macosArm64()
    // iosSimulatorArm64()

    targets.withType<KotlinNativeTarget>().configureEach {
        binaries {
            framework {
                baseName = "Demo"
            }
        }
    }
}

tasks.wrapper {
    gradleVersion = "9.7.0"
    distributionType = Wrapper.DistributionType.ALL
}
```

Groovy

```
import org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget

plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.4.20'
}

repositories {
    mavenCentral()
}

kotlin {
    iosArm64()
    // macosArm64()
    // iosSimulatorArm64()

    targets.withType(KotlinNativeTarget).configureEach {
        binaries {
            framework {
                baseName = "Demo"
            }
        }
    }
}

wrapper {
    gradleVersion = "9.7.0"
    distributionType = "ALL"
}
```

The `binaries {}` block configures the project to generate a dynamic or shared library.

Kotlin/Native supports the `iosArm64` and `iosSimulatorArm64` targets for iOS, as well as the `macosArm64` target for macOS. So, you can replace `iosArm64()` with the respective Gradle function for your target platform:

Target/device	Gradle function
macOS ARM64	`macosArm64()`
iOS ARM64	`iosArm64()`
iOS Simulator (ARM64)	`iosSimulatorArm64()`

For information on other supported Apple targets, see Kotlin/Native target support.

3. To build the framework, run the `linkDebugFramework<YourTargetName>` Gradle task in your IDE or use the console command in your terminal, for example:

```
./gradlew linkDebugFrameworkIosArm64
```

The build generates the framework into the `build/bin/<yourTargetName>/debugFramework` directory.

You can also use the general `link<YourTargetName>` Gradle task to generate both `debug` and `release` variants of the framework.

Generated framework headers

Each framework variant contains a header file. The headers don't depend on the target platform. Header files contain definitions for your Kotlin code and a few Kotlin-wide declarations. Let's see what's inside.

Kotlin/Native runtime declarations

In the `build/bin/<yourTargetName>/debugFramework/Demo.framework/Headers` directory, open the `Demo.h` header file. Take a look at Kotlin runtime declarations:

```
NS_ASSUME_NONNULL_BEGIN
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wunknown-warning-option"
#pragma clang diagnostic ignored "-Wincompatible-property-type"
#pragma clang diagnostic ignored "-Wnullability"

#pragma push_macro("_Nullable_result")
#if !__has_feature(nullability_nullable_result)
#undef _Nullable_result
#define _Nullable_result _Nullable
#endif

__attribute__((swift_name("KotlinBase")))
@interface DemoBase : NSObject
- (instancetype)init __attribute__((unavailable));
+ (instancetype)new __attribute__((unavailable));
+ (void)initialize __attribute__((objc_requires_super));
@end

@interface DemoBase (DemoBaseCopying) <NSCopying>
@end

__attribute__((swift_name("KotlinMutableSet")))
@interface DemoMutableSet<ObjectType> : NSMutableSet<ObjectType>
@end

__attribute__((swift_name("KotlinMutableDictionary")))
@interface DemoMutableDictionary<KeyType, ObjectType> : NSMutableDictionary<KeyType, ObjectType>
@end

@interface NSError (NSErrorDemoKotlinException)
@property (readonly) id _Nullable kotlinException;
@end
```

Kotlin classes have a `KotlinBase` base class in Swift/Objective-C that extends the `NSObject` class there. There are also wrappers for collections and exceptions. Most of the collection types are mapped to similar collection types in Swift/Objective-C:

Kotlin	Swift	Objective-C
List	Array	NSArray

Kotlin	Swift	Objective-C
MutableList	NSMutableArray	NSMutableArray
Set	Set	NSSet
MutableSet	NSMutableSet	NSMutableSet
Map	Dictionary	NSDictionary
MutableMap	NSMutableDictionary	NSMutableDictionary

Kotlin numbers and NSNumber

The next part of the `Demo.h` file contains type mappings between Kotlin/Native number types and `NSNumber`. The base class is called `DemoNumber` in Objective-C and `KotlinNumber` in Swift. It extends `NSNumber`.

For each Kotlin number type, there is a corresponding predefined child class:

Kotlin	Swift	Objective-C	Simple type
`-`	`KotlinNumber`	`<Package>Number`	`-`
`Byte`	`KotlinByte`	`<Package>Byte`	`char`
`UByte`	`KotlinUByte`	`<Package>UByte`	`unsigned char`
`Short`	`KotlinShort`	`<Package>Short`	`short`
`UShort`	`KotlinUShort`	`<Package>UShort`	`unsigned short`
`Int`	`KotlinInt`	`<Package>Int`	`int`
`UInt`	`KotlinUInt`	`<Package>UInt`	`unsigned int`
`Long`	`KotlinLong`	`<Package>Long`	`long long`
`ULong`	`KotlinULong`	`<Package>ULong`	`unsigned long long`
`Float`	`KotlinFloat`	`<Package>Float`	`float`
`Double`	`KotlinDouble`	`<Package>Double`	`double`
`Boolean`	`KotlinBoolean`	`<Package>Boolean`	`BOOL/Bool`

Every number type has a class method to create a new instance from the corresponding simple type. Also, there is an instance method to extract a simple value back. Schematically, all such declarations look like that:

```
__attribute__((swift_name("Kotlin__TYPE__")))
@interface Demo__TYPE__ : DemoNumber
- (instancetype)initWith__TYPE__:(__CTYPE__)value;
+ (instancetype)numberWith__TYPE__:(__CTYPE__)value;
@end;
```

Here, `__TYPE__` is one of the simple type names, and `__CTYPE__` is the corresponding Objective-C type, for example, `initWithChar(char)`.

These types are used to map boxed Kotlin number types to Swift/Objective-C. In Swift, you can call the constructor to create an instance, for example, `KotlinLong(value: 42)`.

Classes and objects from Kotlin

Let's see how `class` and `object` are mapped to Swift/Objective-C. The generated `Demo.h` file contains the exact definitions for `Class`, `Interface`, and `Object`:

```
__attribute__((swift_name("Interface")))
@protocol DemoInterface
@required
- (void)iMember __attribute__((swift_name("iMember()")));
@end

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Clazz")))
@interface DemoClazz : DemoBase <DemoInterface>
- (instancetype)init __attribute__((swift_name("init()")))
__attribute__((objc_designated_initializer));
+ (instancetype)new __attribute__((availability(swift, unavailable, message="use object initializers
instead")));
- (DemoULong * _Nullable)memberP:(int32_t)p __attribute__((swift_name("member(p:)")));
@end

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Object")))
@interface DemoObject : DemoBase
+ (instancetype)alloc __attribute__((unavailable));
+ (instancetype)allocWithZone:(struct _NSZone *)zone __attribute__((unavailable));
+ (instancetype)object __attribute__((swift_name("init()")));
@property (class, readonly, getter=shared) DemoObject *shared __attribute__((swift_name("shared")));
@property (readonly) NSString *field __attribute__((swift_name("field")));
@end
```

Objective-C attributes in this code help use the framework from both Swift and Objective-C languages. `DemoInterface`, `DemoClazz`, and `DemoObject` are created for `Interface`, `Clazz`, and `Object`, respectively.

The `Interface` is turned into `@protocol`, while both a `class` and an `object` are represented as `@interface`. The `Demo` prefix comes from the framework name. The nullable return type `ULong?` is turned into `DemoULong` in Objective-C.

Global declarations from Kotlin

All global functions from Kotlin are turned into `DemoLibKt` in Objective-C and into `LibKt` in Swift, where `Demo` is the framework name set by the `-output` parameter of `kotlinc-native`:

```
__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("LibKt")))
@interface DemoLibKt : DemoBase
+ (NSString * _Nullable)acceptFunF:(NSString * _Nullable (^)(NSString *))f
__attribute__((swift_name("acceptFun(f:)")));
+ (void)forFloatsF:(float)f d:(DemoDouble * _Nullable)d
__attribute__((swift_name("forFloats(f:d:)")));
+ (void)forIntegersB:(int8_t)b s:(uint16_t)s i:(int32_t)i l:(DemoULong * _Nullable)l
__attribute__((swift_name("forIntegers(b:s:i:l:)")));
+ (NSString *)stringsStr:(NSString * _Nullable)str __attribute__((swift_name("strings(str:)")));
+ (NSString * _Nullable (^)(NSString *))supplyFun __attribute__((swift_name("supplyFun()")));
@end
```

Kotlin `String` and Objective-C `NSString*` are mapped transparently. Similarly, `Unit` type from Kotlin is mapped to `void`. The primitive types are mapped directly. Non-nullable primitive types are mapped transparently. Nullable primitive types are mapped to `Kotlin<TYPE>*` types, as shown in the table. Both higher-order functions `acceptFunF` and `supplyFun` are included and accept Objective-C blocks.

You can find more information about type mapping in Interoperability with Swift/Objective-C.

Garbage collection and reference counting

Swift and Objective-C use automatic reference counting (ARC). Kotlin/Native has its own garbage collector, which is also integrated with Swift/Objective-C ARC.

Unused Kotlin objects are automatically removed. You don't need to take additional steps to control the lifetime of Kotlin/Native instances from Swift or Objective-C.

Use code from Objective-C

Let's call the framework from Objective-C. In the framework directory, create the `main.m` file with the following code:

```
#import <Foundation/Foundation.h>
#import <Demo/Demo.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        [DemoObject.shared field];

        DemoClazz* clazz = [[ DemoClazz alloc] init];
        [clazz memberP:42];

        [DemoLibKt forIntegersB:1 s:1 i:3 l:[DemoULong numberWithUnsignedLongLong:4]];
        [DemoLibKt forIntegersB:1 s:1 i:3 l:nil];

        [DemoLibKt forFloatsF:2.71 d:[DemoDouble numberWithDouble:2.71]];
        [DemoLibKt forFloatsF:2.71 d:nil];

        NSString* ret = [DemoLibKt acceptFunF:^NSString * _Nullable(NSString * it) {
            return [it stringByAppendingString:@" Kotlin is fun"];
        }];

        NSLog(@"%@", ret);
        return 0;
    }
}
```

Here, you call Kotlin classes directly from Objective-C code. A Kotlin object uses the `<object name>.shared` class property, which allows you to get the object's only instance and call object methods on it.

The widespread pattern is used to create an instance of the `Clazz` class. You call the `[[ DemoClazz alloc] init]` on Objective-C. You can also use `[DemoClazz new]` for constructors without parameters.

Global declarations from the Kotlin sources are scoped under the `DemoLibKt` class in Objective-C. All Kotlin functions are turned into class methods of that class.

The `strings` function is turned into `DemoLibKt.stringsStr` function in Objective-C, so you can pass `NSString` directly to it. The return value is visible as `NSString` too.

Use code from Swift

The framework you generated has helper attributes to make it easier to use with Swift. Let's convert the previous Objective-C example into Swift.

In the framework directory, create the `main.swift` file with the following code:

```
import Foundation
import Demo

let kotlinObject = Object.shared

let field = Object.shared.field

let clazz = Clazz()
clazz.member(p: 42)

LibKt.forIntegers(b: 1, s: 2, i: 3, l: 4)
LibKt.forFloats(f: 2.71, d: nil)

let ret = LibKt.acceptFun { "\($0) Kotlin is fun" }
if (ret ≠ nil) {
    print(ret!)
}
```

There are some small differences between the original Kotlin code and its Swift version. In Kotlin, any object declaration has only one instance. The `Object.shared` syntax is used to access this single instance.

Kotlin function and property names are translated as is. Kotlin's `String` is turned into Swift's `String`. Swift hides `NSNumber*` boxing too. You can also pass a Swift closure to Kotlin and call a Kotlin lambda function from Swift.

You can find more information about type mapping in Interoperability with Swift/Objective-C.

Connect the framework to your iOS project

Now you can connect the generated framework to your iOS project as a dependency. There are multiple ways to set it up and automate the process, choose the method that suits you best:

Choose iOS integration method

What's next

- Learn more about interoperability with Objective-C

- See how interoperability with C is implemented in Kotlin
- Check out the Kotlin/Native as a dynamic library tutorial

Interoperability with Swift using Swift export

Kotlin's interoperability with Swift through Swift export is currently in Alpha. Swift export allows you to export Kotlin sources directly and call Kotlin code from Swift idiomatically, eliminating the need for Objective-C headers.

Swift export makes multiplatform development for Apple targets more streamlined. For example, if you have a Kotlin module with top-level functions, Swift export enables clean, module-specific imports, removing the confusing Objective-C underscores and mangled names.

Current Swift export features are:

- **Multi-module support**. Each Kotlin module is exported as a separate Swift module, simplifying function calls.
- **Package support**. Kotlin packages are explicitly preserved during export, avoiding naming conflicts in the generated Swift code.
- **Type aliases**. Kotlin type aliases are exported and preserved in Swift, improving readability.
- **Enhanced nullability for primitives**. Unlike Objective-C interop, which required boxing types like `Int?` into wrapper classes like `KotlinInt` to preserve nullability, Swift export converts nullability information directly.
- **Overloads**. You can call Kotlin's overloaded functions in Swift without ambiguity.
- **Flattened package structure**. You can translate Kotlin packages into Swift enums, removing the package prefix from generated Swift code.
- **Module name customization**. You can customize the resulting Swift module names in the Gradle configuration of your Kotlin project.
- **Concurrency support**. You can seamlessly call suspending Kotlin code from Swift and export `kotlinx.coroutines` flows as Swift's `AsyncSequence` out of the box.

Enable Swift export

Swift export is currently in Alpha and still incomplete, so breaking changes are expected. To try it out, configure the build file in your Kotlin project and set up Xcode to integrate Swift export.

Configure Kotlin project

You can use the following build file in your project as a starting point for setting up Swift export:

```kotlin
// build.gradle.kts
kotlin {

    iosArm64()
    iosSimulatorArm64()

    swiftExport {
        // Set the root module name
        moduleName = "Shared"

        // Set the collapse rule
        // Removes package prefix from generated Swift code
        flattenPackage = "com.example.sandbox"

        // Configure external modules export
        export(project(":subproject")) {
            // Set the name for the exported module
            moduleName = "Subproject"
            // Set the collapse rule for the exported dependency
            flattenPackage = "com.subproject.library"
        }

        // Provide compiler arguments to link tasks
        configure {
            freeCompilerArgs.add("-Xexpect-actual-classes")
        }
    }
}
```

The Kotlin compiler automatically generates all the necessary files (including `swiftmodule` files, static `.a` library, a header file, and a `modulemap` file) and copies them into the app's build directory, which you can access from Xcode.

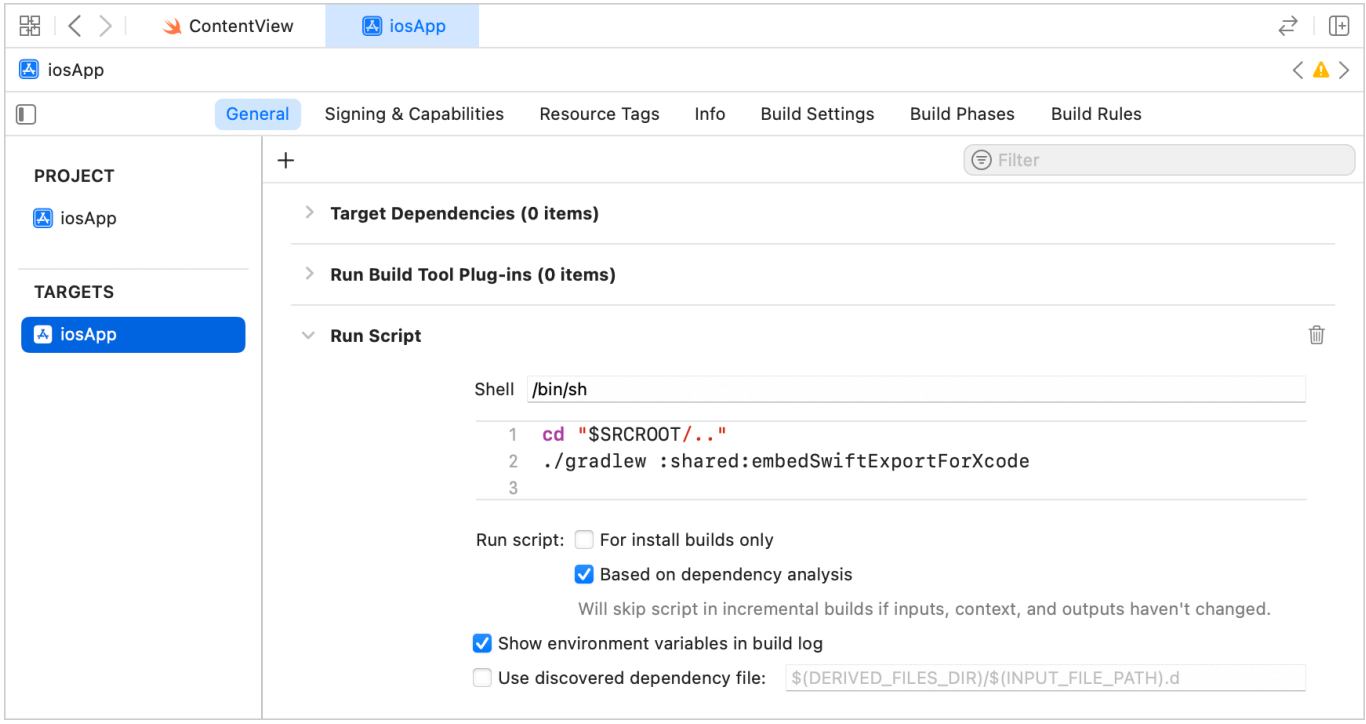
> You can also clone our public sample with Swift export already set up.

Configure Xcode project

To configure Xcode to integrate Swift export into your project:

1. In Xcode, open the project settings.
2. On the **Build Phases** tab, locate the **Run Script** phase with the `embedAndSignAppleFrameworkForXcode` task.
3. Replace the script with the `embedSwiftExportForXcode` task in the run script phase:

```
./gradlew :<Shared module name>:embedSwiftExportForXcode
```



Add the Swift export script

4. Build the project. The build generates Swift modules in the output directory.

Current limitations

Swift export currently works only in projects that use direct integration to connect the iOS framework to the Xcode project. This is a standard configuration for Kotlin Multiplatform projects created with the Kotlin Multiplatform plugin in IntelliJ IDEA or through the web wizard.

Other known issues:

- Types that inherit from `List`, `Set`, or `Map` are ignored during the export (KT-80416).
- Inheritors of `List`, `Set`, or `Map` cannot be instantiated on the Swift side (KT-80417).
- When exported to Swift, Kotlin generic type parameters are type-erased to their upper bounds.
- No IDE migration tips or automation are available.
- When using declarations that require opt-in, you must add an explicit `optIn` compiler option at the *module level* to your Gradle build file. For example, for the `kotlinx.datetime` library:

```
swiftExport {
    moduleName = "Shared"

    export("org.jetbrains.kotlinx:kotlinx-datetime:0.8.0") {
        moduleName = "KotlinDateTime"
        flattenPackage = "kotlinx.datetime"
    }
}

// Add a separate opt-in block at the module level
compilerOptions {
    optIn.add("kotlin.time.ExperimentalTime")
}
```

Mappings

The table below shows how Kotlin concepts are mapped to Swift.

Kotlin	Swift
`class`	`class`
`object`	`class` with `shared` property
`enum class`	`enum`
`sealed` classes and interfaces	`enum`
`typealias`	`typealias`
Function	Function
`suspend fun`	`async`
`kotlinx.coroutines` flows	`AsyncSequence`
Property	Property
Constructor	Initializer
Package	Nested enum
`Boolean`	`Bool`
`Char`	`Unicode.UTF16.CodeUnit`
`Byte`	`Int8`
`Short`	`Int16`
`Int`	`Int32`
`Long`	`Int64`
`UByte`	`UInt8`
`UShort`	`UInt16`

Kotlin	Swift
`UInt`	`UInt32`
`ULong`	`UInt64`
`Float`	`Float`
`Double`	`Double`
`Any`	`KotlinBase` class
`Unit`	`Void`
`Nothing`	`Never`

Declarations

Classes

Swift export supports only final classes that directly inherit from `Any`, like `class Foo()`. They are translated to Swift classes that inherit from a special `KotlinBase` class:

```
// Kotlin
class MyClass {
    val property: Int = 0

    fun method() {}
}
```

```
// Swift
public class MyClass : KotlinRuntime.KotlinBase {
    public var property: Swift.Int32 {
        get {
            // ...
        }
    }
    public override init() {
        // ...
    }
    public func method() -> Swift.Void {
        // ...
    }
}
```

Objects

Objects are translated to Swift classes with a private `init` and static `shared` accessor:

```
// Kotlin
object O
```

```
// Swift
public class O : KotlinRuntime.KotlinBase {
    public static var shared: O {
        get {
            // ...
        }
    }
    private override init() {
        // ...
    }
}
```

Type aliases

Kotlin type aliases are exported as is:

```
// Kotlin
typealias MyInt = Int
```

```
// Swift
public typealias MyInt = Swift.Int32
```

Enums

Kotlin `enum class` declarations are exported as regular native Swift `enum` types:

```
// Kotlin
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}

val color = Color.RED
```

```
// Swift
public enum Color: Swift.CaseIterable, Swift.LosslessStringConvertible, Swift.RawRepresentable {
    case RED, GREEN, BLUE

    public var rgb: Swift.Int32 { get }
}
```

Sealed classes and interfaces

Sealed hierarchies defined in Kotlin are mapped to Swift enums, enabling exhaustive `switch` statements.

Swift export generates a `.sealedType()` method on each sealed type. This method returns a Swift enum whose cases match the direct subclasses of the sealed hierarchy. You can nest these calls to match deeper levels of the hierarchy.

For example, declare a sealed interface with a class hierarchy in Kotlin:

```kotlin
// Kotlin
sealed interface Shape

class Circle : Shape {
    override fun toString(): String = "Circle"
}

class Rectangle : Shape {
    override fun toString(): String = "Rectangle"
}

fun createCircle(): Shape = Circle()
```

On the Swift side, you can use an exhaustive `switch` without a `default` case:

```swift
// Swift
let shape = createCircle()

let name = switch shape.sealedType() {
    case let .circle(type): "It's a \(type.value)"
    case let .rectangle(type): "It's a \(type.value)"
}
// name == "It's a Circle"
```

Because the `switch` is exhaustive, the compiler warns you if a new subclass is added to the sealed hierarchy, so you can handle it immediately instead of relying on a `default` case for `switch`.

Functions

Swift export supports simple top-level functions and methods:

```kotlin
// Kotlin
fun foo(a: Short, b: Bar) {}

fun baz(): Long = 0
```

```swift
// Swift
public func foo(a: Swift.Int16, b: Bar) -> Swift.Void {
    // ...
}

public func baz() -> Swift.Int64 {
    // ...
}
```

For Kotlin's extension functions, the receiver parameter becomes an ordinary Swift parameter in the first position:

```kotlin
// Kotlin
fun Int.foo(): Unit = TODO()
```

```swift
// Swift
func foo(_ receiver: Int32) {}
```

Kotlin's functions with `vararg` are mapped to Swift's variadic function parameters:

```kotlin
// Kotlin
fun log(vararg messages: String)
```

```kotlin
// Swift
public func log(messages: Swift.String...)
```

- Support for functions with the `operator` modifier is currently limited.
- Generic types are generally not supported.

Properties

Kotlin properties are translated to Swift properties:

```kotlin
// Kotlin
val a: Int = 0

var b: Short = 15

const val c: Int = 0
```

```kotlin
// Swift
public var a: Swift.Int32 {
    get {
        // ...
    }
}
public var b: Swift.Int16 {
    get {
        // ...
    }
    set {
        // ...
    }
}
public var c: Swift.Int32 {
    get {
        // ...
    }
}
```

Constructors

Constructors are translated to Swift initializers:

```kotlin
// Kotlin
class Foo(val prop: Int)
```

```
// Swift
public class Foo : KotlinRuntime.KotlinBase {
    public init(
        prop: Swift.Int32
    ) {
        // ...
    }
}
```

Types

kotlin.Nothing

The Kotlin `Nothing` type is translated to the `Never` type:

```
// Kotlin
fun foo(): Nothing = TODO()

fun baz(input: Nothing) {}
```

```
// Swift
public func foo() -> Swift.Never {
    // ...
}

public func baz(input: Swift.Never) -> Void {
    // ...
}
```

Classifier types

Swift export currently supports only final classes that directly inherit from `Any`.

Packages

Kotlin packages are translated to nested Swift enums to avoid name collisions:

```
// Kotlin
// bar.kt file in foo.bar package
fun callMeMaybe() {}
```

```
// Kotlin
// baz.kt file in foo.baz package
fun callMeMaybe() {}
```

```swift
// Swift
public extension foo.bar {
    public func callMeMaybe() {}
}

public extension foo.baz {
    public func callMeMaybe() {}
}

public enum foo {
    public enum bar {}

    public enum baz {}
}
```

Concurrency

Suspending functions

You can call suspending Kotlin code from Swift. Kotlin suspending functions and suspending functional types are exported as Swift's `async` counterparts:

```kotlin
// Kotlin
suspend fun hello(): String {
    delay(1000)
    return "Hello Swift! This is Kotlin."
}
```

```swift
// Swift
let msg = try await hello()
```

Flows

You can also export `kotlinx.coroutines` flows as Swift's `AsyncSequence` :

```kotlin
// Kotlin
// Preserves the String type when exporting Flow
fun flowOfStrings(): Flow<String> = flowOf("hello", "any", "world")
```

```swift
// Swift
var actual: [String] = []

// Infers the String type from Kotlin
for try await element in flowOfStrings().asAsyncSequence() {
    actual.append(element)
}
```

Coroutine dispatchers

By default, when you call a Kotlin suspending function from Swift or use the `asAsyncSequence` function, Kotlin creates a coroutine context that uses the `Dispatchers.Default` dispatcher and executes the exported code there.

To run the exported code on a different dispatcher, use the `withContext()` function to switch the coroutine context in Kotlin. For example:

```
suspend fun runOnMain(): Int = withContext(Dispatchers.Main) {
    delay(10L)
    42
}
```

Cross-language inheritance

Swift export supports cross-language inheritance. A common use case for this feature is the reverse import pattern, where you define a contract in Kotlin and provide platform-specific implementations on the Swift side. This is especially useful when you need to use pure Swift libraries that can't be directly imported into Kotlin.

To implement the pattern, you need to declare a Kotlin interface and a Kotlin superclass that the Swift implementation can inherit from. You then implement the interface in Swift and pass the Swift object to Kotlin functions that accept that interface. For example, for the CryptoKit library:

1. On the Kotlin side, declare an interface, a function that accepts it, and an `open` base class:

   ```
   // Kotlin
   interface CryptoProvider {
       fun hashMD5(input: String): String
   }

   fun processHash(provider: CryptoProvider, input: String): String = provider.hashMD5(input)

   open class SwiftBase
   ```

2. On the Swift side, inherit from the exported `SwiftBase` class, implement the interface using a pure Swift library, and pass the object back to Kotlin:

   ```
   // Swift
   import CryptoKit

   final class IosCryptoProvider: SwiftBase, CryptoProvider {
       func hashMD5(input: String) -> String {
           guard let data = input.data(using: .utf8) else { return "failed" }
           return Insecure.MD5.hash(data: data).description
       }
   }

   let provider = IosCryptoProvider()

   // Calls the Kotlin function, which calls hashMD5() back in Swift
   print(processHash(provider: provider, input: "Hello, world!"))
   ```

When Kotlin receives the Swift object, it treats it like an implementation of a regular Kotlin interface, calling the Swift code directly.

Evolution of Swift export

We're planning to expand and gradually stabilize Swift export in future Kotlin releases, improving interoperability between Kotlin and Swift. You can leave your feedback:

- In Kotlin Slack – get an invite and join the #swift-export channel.

- Report issues in YouTrack.

Interoperability with C

> The C libraries import is in Beta. All Kotlin declarations generated by the cinterop tool from C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.

This document covers general aspects of Kotlin's interoperability with C. Kotlin/Native comes with a cinterop tool, which you can use to quickly generate everything you need to interact with an external C library.

The tool analyzes C headers and produces a straightforward mapping of C types, functions, and strings into Kotlin. The generated stubs then can be imported into an IDE to enable code completion and navigation.

> Kotlin also provides interoperability with Objective-C. Objective-C libraries are imported through the cinterop tool as well. For more details, see Swift/Objective-C interop.

Setting up your project

Here's a general workflow when working with a project that needs to consume a C library:

1. Create and configure a definition file. It describes what the cinterop tool should include into Kotlin bindings.
2. Configure your Gradle build file to include cinterop in the build process.
3. Compile and run the project to produce the final executable.

> For a hands-on experience, complete the Create an app using C interop tutorial.

In many cases, there's no need to configure custom interoperability with a C library. Instead, you can use APIs available on the platform standardized bindings called platform libraries. For example, POSIX on Linux/macOS platforms, Win32 on the Windows platform, or Apple frameworks on macOS/iOS are available this way.

Bindings

Basic interop types

All the supported C types have corresponding representations in Kotlin:

- Signed, unsigned integral, and floating point types are mapped to their Kotlin counterpart with the same width.
- Pointers and arrays are mapped to `CPointer<T>?`.
- Enums can be mapped to either Kotlin enum or integral values, depending on heuristics and the definition file settings.
- Structs and unions are mapped to types having fields available via the dot notation, i.e. `someStructInstance.field1`.

- `typedef` are represented as `typealias`.

Also, any C type has the Kotlin type representing the lvalue of this type, i.e., the value located in memory rather than a simple immutable self-contained value. Think C++ references as a similar concept. For structs (and `typedef`s to structs), this representation is the main one and has the same name as the struct itself. For Kotlin enums, it's named `${type}.Var`; for `CPointer<T>`, it's `CPointerVar<T>`; and for most other types, it's `${type}Var`.

For types that have both representations, the one with the lvalue has a mutable `.value` property for accessing the value.

Pointer types

The type argument `T` of `CPointer<T>` must be one of the lvalue types described above. For example, the C type `struct S*` is mapped to `CPointer<S>`, `int8_t*` is mapped to `CPointer<int_8tVar>`, and `char**` is mapped to `CPointer<CPointerVar<ByteVar>>`.

C null pointer is represented as Kotlin's `null`, and the pointer type `CPointer<T>` is not nullable, but the `CPointer<T>?` is. The values of this type support all the Kotlin operations related to handling `null`, for example, `?:`, `?.`, `!!`, and so on:

```
val path = getenv("PATH")?.toKString() ?: ""
```

Since the arrays are also mapped to `CPointer<T>`, it supports the `[]` operator for accessing values by index:

```
import kotlinx.cinterop.*

@OptIn(ExperimentalForeignApi::class)
fun shift(ptr: CPointer<ByteVar>, length: Int) {
    for (index in 0 .. length - 2) {
        ptr[index] = ptr[index + 1]
    }
}
```

The `.pointed` property for `CPointer<T>` returns the lvalue of type `T`, pointed by this pointer. The reverse operation is `.ptr`, it takes the lvalue and returns the pointer to it.

`void*` is mapped to `COpaquePointer` – the special pointer type which is the supertype for any other pointer type. So if the C function takes `void*`, the Kotlin binding accepts any `CPointer`.

Casting a pointer (including `COpaquePointer`) can be done with `.reinterpret<T>`, for example:

```
import kotlinx.cinterop.*

@OptIn(ExperimentalForeignApi::class)
val intPtr = bytePtr.reinterpret<IntVar>()
```

Or:

```
import kotlinx.cinterop.*

@OptIn(ExperimentalForeignApi::class)
val intPtr: CPointer<IntVar> = bytePtr.reinterpret()
```

As is with C, these `.reinterpret` casts are unsafe and can potentially lead to subtle memory problems in the application.

Also, there are unsafe casts between `CPointer<T>?` and `Long` available, provided by the `.toLong()` and `.toCPointer<T>()` extension methods:

```
val longValue = ptr.toLong()
val originalPtr = longValue.toCPointer<T>()
```

> If the type of the result is known from the context, you can omit the type argument thanks to the type inference.

Memory allocation

The native memory can be allocated using the `NativePlacement` interface, for example:

```
@file:OptIn(ExperimentalForeignApi::class)
import kotlinx.cinterop.*

val placement: NativePlacement = // See below for placement examples
val byteVar = placement.alloc<ByteVar>()
val bytePtr = placement.allocArray<ByteVar>(5)
```

The most logical placement is in the object `nativeHeap`. It corresponds to allocating native memory with `malloc` and provides an additional `.free()` operation to free allocated memory:

```
@file:OptIn(ExperimentalForeignApi::class)
import kotlinx.cinterop.*

fun main() {
    val size: Long = 0
    val buffer = nativeHeap.allocArray<ByteVar>(size)
    nativeHeap.free(buffer)
}
```

`nativeHeap` requires memory to be freed manually. However, it's often useful to allocate memory with a lifetime bound to the lexical scope. It's helpful if such memory is freed automatically.

To address this, you can use `memScoped { }`. Inside the braces, the temporary placement is available as an implicit receiver, so it's possible to allocate native memory with alloc and allocArray, and the allocated memory will be automatically freed after leaving the scope.

For example, a C function returning values through pointer parameters can be used like:

```
@file:OptIn(ExperimentalForeignApi::class)
import kotlinx.cinterop.*
import platform.posix.*

val fileSize = memScoped {
    val statBuf = alloc<stat>()
    val error = stat("/", statBuf.ptr)
    statBuf.st_size
}
```

Pass pointers to bindings

Although C pointers are mapped to the `CPointer<T> type`, C function pointer-typed parameters are mapped to `CValuesRef<T>.` When passing a `CPointer<T>` as a value of such a parameter, it's passed to the C function as

is. However, a sequence of values can be passed instead of a pointer. In this case, the sequence is passed "by value", i.e., the C function receives the pointer to the temporary copy of that sequence, which is valid only until the function returns.

The `CValuesRef<T>` representation of pointer parameters is designed to support C array literals without explicit native memory allocation. To construct the immutable self-contained sequence of C values, the following methods are provided:

- `${type}Array.toCValues()`, where `type` is the Kotlin primitive type
- `Array<CPointer<T>?>.toCValues()`, `List<CPointer<T>?>.toCValues()`
- `cValuesOf(vararg elements: ${type})`, where `type` is a primitive or pointer

For example:

```
// C:
void foo(int* elements, int count);
...
int elements[] = {1, 2, 3};
foo(elements, 3);
```

```
// Kotlin:

foo(cValuesOf(1, 2, 3), 3)
```

Strings

Unlike other pointers, the parameters of type `const char*` are represented as a Kotlin `String`. So it's possible to pass any Kotlin string to a binding expecting a C string.

There are also some tools available to convert between Kotlin and C strings manually:

- `fun CPointer<ByteVar>.toKString(): String`
- `val String.cstr: CValuesRef<ByteVar>`.

To get the pointer, `.cstr` should be allocated in native memory, for example:

```
val cString = kotlinString.cstr.getPointer(nativeHeap)
```

In all cases, the C string is supposed to be encoded as UTF-8.

To skip automatic conversion and ensure raw pointers are used in the bindings, add the `noStringConversion` property to the `.def` file:

```
noStringConversion = LoadCursorA LoadCursorW
```

This way, any value of type `CPointer<ByteVar>` can be passed as an argument of `const char*` type. If a Kotlin string should be passed, code like this could be used:

```
import kotlinx.cinterop.*

@OptIn(kotlinx.cinterop.ExperimentalForeignApi::class)
memScoped {
    LoadCursorA(null, "cursor.bmp".cstr.ptr)  // for ASCII or UTF-8 version
    LoadCursorW(null, "cursor.bmp".wcstr.ptr) // for UTF-16 version
}
```

Scope-local pointers

It's possible to create a scope-stable pointer of C representation for the `CValues<T>` instance using the `CValues<T>.ptr` extension property, available under `memScoped {}`. It allows using APIs that require C pointers with a lifetime bound to a certain `MemScope`. For example:

```
import kotlinx.cinterop.*

@OptIn(kotlinx.cinterop.ExperimentalForeignApi::class)
memScoped {
    items = arrayOfNulls<CPointer<ITEM>?>(6)
    arrayOf("one", "two").forEachIndexed { index, value -> items[index] = value.cstr.ptr }
    menu = new_menu("Menu".cstr.ptr, items.toCValues().ptr)
    // ...
}
```

In this example, all values passed to the C API `new_menu()` have a lifetime of the innermost `memScope` it belongs to. Once the control flow leaves the `memScoped` scope, C pointers become invalid.

Pass and receive structs by value

When a C function takes or returns a struct/union `T` by value, the corresponding argument type or return type is represented as `CValue<T>`.

`CValue<T>` is an opaque type, so the structure fields cannot be accessed with the appropriate Kotlin properties. This can be fine if an API uses structures as opaque handles. However, if field access is required, the following conversion methods are available:

- `fun T.readValue(): CValue<T>` converts (the lvalue) `T` to a `CValue<T>`. So to construct the `CValue<T>`, `T` can be allocated, filled, and then converted to `CValue<T>`.
- `CValue<T>.useContents(block: T.() -> R): R` temporarily stores the `CValue<T>` in memory, and then runs the passed lambda with this placed value `T` as receiver. So to read a single field, you can use the following code:

  ```
  val fieldValue = structValue.useContents { field }
  ```

- `fun cValue(initialize: T.() -> Unit): CValue<T>` applies the provided `initialize` function to allocate `T` in memory and converts the result into a `CValue<T>`.
- `fun CValue<T>.copy(modify: T.() -> Unit): CValue<T>` creates a modified copy of an existing `CValue<T>`. The original value is placed in memory, altered using the `modify()` function, and then converted back into a new `CValue<T>`.
- `fun CValues<T>.placeTo(scope: AutofreeScope): CPointer<T>` places the `CValues<T>` into an `AutofreeScope`, returning a pointer to the allocated memory. The allocated memory is automatically freed when the `AutofreeScope` is disposed.

Callbacks

To convert a Kotlin function to a pointer to a C function, you can use `staticCFunction(::kotlinFunction)`. It's also possible to provide a lambda instead of a function reference. The function or lambda must not capture any values.

Pass user data to callbacks

Often C APIs allow passing some user data to callbacks. Such data is usually provided by the user when configuring the callback. It's passed to some C function (or written to the struct) as `void*`, for example. However, references to Kotlin objects can't be directly passed to C. So they require wrapping before configuring the callback and then unwrapping in the callback itself, to safely swim from Kotlin to Kotlin through the C world. Such wrapping is possible with the `StableRef` class.

To wrap the reference:

```
import kotlinx.cinterop.*

@OptIn(ExperimentalForeignApi::class)
val stableRef = StableRef.create(kotlinReference)
val voidPtr = stableRef.asCPointer()
```

Here, the `voidPtr` is a `COpaquePointer` and can be passed to the C function.

To unwrap the reference:

```
@OptIn(ExperimentalForeignApi::class)
val stableRef = voidPtr.asStableRef<KotlinClass>()
val kotlinReference = stableRef.get()
```

Here, `kotlinReference` is the original wrapped reference.

The created `StableRef` eventually be manually disposed using the `.dispose()` method to prevent memory leaks:

```
stableRef.dispose()
```

After that it becomes invalid, so `voidPtr` can't be unwrapped anymore.

Macros

Every C macro that expands to a constant is represented as a Kotlin property.

Macros without parameters are supported in cases when the compiler can infer the type:

```
int foo(int);
#define FOO foo(42)
```

In this case, `FOO` is available in Kotlin.

To support other macros, you can expose them manually by wrapping them with supported declarations. For example, function-like macro `FOO` can be exposed as a function `foo()` by adding custom declaration to the library:

```
headers = library/base.h

---

static inline int foo(int arg) {
    return FOO(arg);
}
```

Portability

Sometimes the C libraries have function parameters or struct fields of a platform-dependent type, for example, `long` or `size_t`. Kotlin itself doesn't provide either implicit integer casts or C-style integer casts (for example, `(size_t) intValue`), so to make writing portable code in such cases easier, the `convert` method is provided:

```
fun ${type1}.convert<${type2}>(): ${type2}
```

Here, each of `type1` and `type2` must be an integral type, either signed or unsigned.

`.convert<${type}>` has the same semantics as one of the `.toByte`, `.toShort`, `.toInt`, `.toLong`, `.toUByte`, `.toUShort`, `.toUInt` or `.toULong` methods, depending on `type`.

An example of using `convert`:

```
import kotlinx.cinterop.*
import platform.posix.*

@OptIn(ExperimentalForeignApi::class)
fun zeroMemory(buffer: COpaquePointer, size: Int) {
    memset(buffer, 0, size.convert<size_t>())
}
```

Also, the type parameter can be inferred automatically and so may be omitted in some cases.

Object pinning

Kotlin objects could be pinned, i.e. their position in memory is guaranteed to be stable until they are unpinned, and pointers to such objects' inner data could be passed to C functions.

There's a couple of approaches you can take:

- Use the `.usePinned()` extension function that pins an object, executes a block, and unpins it on normal and exception paths:

```
import kotlinx.cinterop.*
import platform.posix.*

@OptIn(ExperimentalForeignApi::class)
fun readData(fd: Int) {
    val buffer = ByteArray(1024)
    buffer.usePinned { pinned ->
        while (true) {
            val length = recv(fd, pinned.addressOf(0), buffer.size.convert(), 0).toInt()
            if (length <= 0) {
                break
            }
            // Now `buffer` has raw data obtained from the `recv()` call.
        }
    }
}
```

Here, `pinned` is an object of a special type `Pinned<T>`. It provides useful extensions like `.addressOf()`, which allows getting the address of a pinned array body.

- Use the `.refTo()` extension function that has similar functionality under the hood but, in certain cases, may help you reduce boilerplate code:

```
import kotlinx.cinterop.*
import platform.posix.*

@OptIn(ExperimentalForeignApi::class)
fun readData(fd: Int) {
    val buffer = ByteArray(1024)
    while (true) {
        val length = recv(fd, buffer.refTo(0), buffer.size.convert(), 0).toInt()

        if (length <= 0) {
            break
        }
        // Now `buffer` has raw data obtained from the `recv()` call.
    }
}
```

Here, `buffer.refTo(0)` has the `CValuesRef` type that pins the array before entering the `recv()` function, passes the address of its zeroth element to the function, and unpins the array after exiting.

Forward declarations

To import forward declarations, use the `cnames` package. For example, to import a `cstructName` forward declaration declared in a C library with a `library.package`, use a special forward declaration package: `import cnames.structs.cstructName`.

Consider two cinterop libraries: one that has a forward declaration of a struct and another with an actual implementation in another package:

```c
// First C library
#include <stdio.h>

struct ForwardDeclaredStruct;

void consumeStruct(struct ForwardDeclaredStruct* s) {
    printf("Struct consumed\n");
}
```

```c
// Second C library
// Header:
#include <stdlib.h>

struct ForwardDeclaredStruct {
    int data;
};

// Implementation:
struct ForwardDeclaredStruct* produceStruct() {
    struct ForwardDeclaredStruct* s = malloc(sizeof(struct ForwardDeclaredStruct));
    s->data = 42;
    return s;
}
```

To transfer objects between the two libraries, use an explicit `as` cast in your Kotlin code:

```
// Kotlin code:
fun test() {
    consumeStruct(produceStruct() as CPointer<cnames.structs.ForwardDeclaredStruct>)
}
```

What's next

Learn how types, functions, and strings are mapped between Kotlin and C by completing the following tutorials:

1. Mapping primitive data types from C
2. Mapping struct and union types from C
3. Mapping function pointers from C
4. Mapping strings from C

Start

Mapping primitive data types from C – tutorial

> The C libraries import is in Beta. All Kotlin declarations generated by the cinterop tool from C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.

Let's explore which C data types are visible in Kotlin/Native and vice versa and examine advanced C interop-related use cases of Kotlin/Native and multiplatform Gradle builds.

In this tutorial, you'll:

- Learn about data types in the C language
- Create a C Library that uses those types in exports
- Inspect generated Kotlin APIs from a C library

You can use the command line to generate a Kotlin library, either directly or with a script file (such as `.sh` or `.bat` file). However, this approach doesn't scale well for larger projects that have hundreds of files and libraries. Using a build system simplifies the process by downloading and caching the Kotlin/Native compiler binaries and libraries with transitive dependencies, as well as by running the compiler and tests. Kotlin/Native can use the Gradle build system through the Kotlin Multiplatform plugin.

Types in C language

The C programming language has the following data types:

- Basic types: `char, int, float, double` with modifiers `signed, unsigned, short, long`
- Structures, unions, arrays
- Pointers
- Function pointers

There are also more specific types:

- Boolean type (from C99)
- `size_t` and `ptrdiff_t` (also `ssize_t`)
- Fixed width integer types, such as `int32_t` or `uint64_t` (from C99)

There are also the following type qualifiers in the C language: `const` , `volatile` , `restrict` , `atomic` .

Let's see which C data types are visible in Kotlin.

Create a C library

In this tutorial, you won't create a `lib.c` source file, which is only necessary if you want to compile and run your C library. For this setup, you'll only need a `.h` header file that is required for running the cinterop tool.

The cinterop tool generates a Kotlin/Native library (a `.klib` file) for each set of `.h` files. The generated library helps bridge calls from Kotlin/Native to C. It includes Kotlin declarations that correspond to the definitions from the `.h` files.

To create a C library:

1. Create an empty folder for your future project.
2. Inside, create a `lib.h` file with the following content to see how C functions are mapped into Kotlin:

```
#ifndef LIB2_H_INCLUDED
#define LIB2_H_INCLUDED

void ints(char c, short d, int e, long f);
void uints(unsigned char c, unsigned short d, unsigned int e, unsigned long f);
void doubles(float a, double b);

#endif
```

The file doesn't have the `extern "C"` block, which is not needed for this example but may be necessary if you use C++ and overloaded functions. See this Stackoverflow thread for more details.

3. Create the `lib.def` definition file with the following content:

```
headers = lib.h
```

4. It can be helpful to include macros or other C definitions in the code generated by the cinterop tool. This way, method bodies are also compiled and fully included in the binary. With this feature, you can create a runnable example without needing a C compiler.

   To do that, add implementations to the C functions from the `lib.h` file to a new `interop.def` file after the `---` separator:

```
---

void ints(char c, short d, int e, long f) { }
void uints(unsigned char c, unsigned short d, unsigned int e, unsigned long f) { }
void doubles(float a, double b) { }
```

The `interop.def` file provides everything necessary to compile, run, or open the application in an IDE.

Create a Kotlin/Native project

> See the Get started with Kotlin/Native tutorial for detailed first steps and instructions on how to create a new Kotlin/Native project and open it in IntelliJ IDEA.

To create project files:

1. In your project folder, create a `build.gradle(.kts)` Gradle build file with the following content:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget

plugins {
    kotlin("multiplatform") version "2.4.20"
}

repositories {
    mavenCentral()
}

kotlin {
    macosArm64()    // macOS on Apple Silicon
    // linuxArm64() // Linux on ARM64 platforms
    // linuxX64()   // Linux on x86_64 platforms
    // mingwX64()   // Windows on x86_64 platforms

    targets.withType<KotlinNativeTarget>().configureEach {
        val main by compilations.getting
        val interop by main.cinterops.creating

        binaries {
            executable()
        }
    }
}

tasks.wrapper {
    gradleVersion = "9.7.0"
    distributionType = Wrapper.DistributionType.BIN
}
```

Groovy

```
import org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget

plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.4.20'
}

repositories {
    mavenCentral()
}

kotlin {
    macosArm64()    // Apple Silicon macOS
    // linuxArm64() // Linux on ARM64 platforms
    // linuxX64()   // Linux on x86_64 platforms
    // mingwX64()   // Windows

    targets.withType(KotlinNativeTarget).configureEach {
        compilations.main.cinterops {
            interop
        }

        binaries {
            executable()
        }
    }
}

wrapper {
    gradleVersion = '9.7.0'
    distributionType = 'BIN'
}
```

The project file configures the C interop as an additional build step. Check out the Multiplatform Gradle DSL reference to learn about different ways you can configure it.

2. Move your `interop.def`, `lib.h`, and `lib.def` files to the `src/nativeInterop/cinterop` directory.
3. Create a `src/nativeMain/kotlin` directory. This is where you should place all the source files, following Gradle's recommendations on using conventions instead of configurations.

   By default, all the symbols from C are imported to the `interop` package.
4. In `src/nativeMain/kotlin`, create a `hello.kt` stub file with the following content:

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi

@OptIn(ExperimentalForeignApi::class)
fun main() {
    println("Hello Kotlin/Native!")

    ints(/* fix me*/)
    uints(/* fix me*/)
    doubles(/* fix me*/)
}
```

You'll complete the code later as you learn how C primitive type declarations look from the Kotlin side.

Inspect generated Kotlin APIs for a C library

Let's see how C primitive types are mapped into Kotlin/Native and update the example project accordingly.

Use IntelliJ IDEA's Go to declaration command (Cmd + B/Ctrl + B) to navigate to the following generated API for C functions:

```
fun ints(c: kotlin.Byte, d: kotlin.Short, e: kotlin.Int, f: kotlin.Long)
fun uints(c: kotlin.UByte, d: kotlin.UShort, e: kotlin.UInt, f: kotlin.ULong)
fun doubles(a: kotlin.Float, b: kotlin.Double)
```

C types are mapped directly, except for the `char` type, which is mapped to `kotlin.Byte` as it's usually an 8-bit signed value:

C	Kotlin
char	kotlin.Byte
unsigned char	kotlin.UByte
short	kotlin.Short
unsigned short	kotlin.UShort
int	kotlin.Int
unsigned int	kotlin.UInt
long long	kotlin.Long
unsigned long long	kotlin.ULong
float	kotlin.Float
double	kotlin.Double

Update Kotlin code

Now that you've seen the C definitions, you can update your Kotlin code. The final code in the `hello.kt` file may look like this:

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi

@OptIn(ExperimentalForeignApi::class)
fun main() {
    println("Hello Kotlin/Native!")

    ints(1, 2, 3, 4)
    uints(5u, 6u, 7u, 8u)
    doubles(9.0f, 10.0)
}
```

To verify that everything works as expected, run the `runDebugExecutable<YourTargetName>` Gradle task in your IDE or use the console command in your terminal, in this example:

```
./gradlew runDebugExecutableMacosArm64
```

Next step

In the next part of the series, you'll learn how struct and union types are mapped between Kotlin and C:

- Next step

See also

Learn more in the Interoperability with C documentation that covers more advanced scenarios.

Mapping struct and union types from C – tutorial

> The C libraries import is in Beta. All Kotlin declarations generated by the cinterop tool from C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.

Let's explore which C struct and union declarations are visible from Kotlin and examine advanced C interop-related use cases of Kotlin/Native and multiplatform Gradle builds.

In the tutorial, you'll learn:

- How struct and union types are mapped
- How to use struct and union types from Kotlin

Mapping struct and union C types

To understand how Kotlin maps struct and union types, let's declare them in C and examine how they are represented in Kotlin.

In the previous tutorial, you've already created a C library with the necessary files. For this step, update the declarations in the `interop.def` file after the `---` separator:

```
---

typedef struct {
  int a;
  double b;
} MyStruct;

void struct_by_value(MyStruct s) {}
void struct_by_pointer(MyStruct* s) {}

typedef union {
  int a;
  MyStruct b;
  float c;
} MyUnion;

void union_by_value(MyUnion u) {}
void union_by_pointer(MyUnion* u) {}
```

The `interop.def` file provides everything necessary to compile, run, or open the application in an IDE.

Inspect generated Kotlin APIs for a C library

Let's see how C struct and union types are mapped into Kotlin/Native and update your project:

1. In `src/nativeMain/kotlin`, update your `hello.kt` file from the previous tutorial with the following content:

   ```
   import interop.*
   import kotlinx.cinterop.ExperimentalForeignApi

   @OptIn(ExperimentalForeignApi::class)
   fun main() {
       println("Hello Kotlin/Native!")

       struct_by_value(/* fix me*/)
       struct_by_pointer(/* fix me*/)
       union_by_value(/* fix me*/)
       union_by_pointer(/* fix me*/)
   }
   ```

2. To avoid compiler errors, add interoperability to the build process. For that, update your `build.gradle(.kts)` build file with the following content:

Kotlin

```
kotlin {
    macosArm64()    // macOS on Apple Silicon
    // linuxArm64() // Linux on ARM64 platforms
    // linuxX64()   // Linux on x86_64 platforms
    // mingwX64()   // on Windows

    targets.withType<KotlinNativeTarget>().configureEach {
        val main by compilations.getting
        val interop by main.cinterops.creating {
            definitionFile.set(project.file("src/nativeInterop/cinterop/interop.def"))
        }

        binaries {
            executable()
        }
    }
}
```

Groovy

```
kotlin {
    macosArm64()    // Apple Silicon macOS
    // linuxArm64() // Linux on ARM64 platforms
    // linuxX64()   // Linux on x86_64 platforms
    // mingwX64()   // Windows

    targets.withType(KotlinNativeTarget).configureEach {
        compilations.main.cinterops {
            interop {
                definitionFile = project.file('src/nativeInterop/cinterop/interop.def')
            }
        }

        binaries {
            executable()
        }
    }
}
```

3. Use IntelliJ IDEA's Go to declaration command (Cmd + B/Ctrl + B) to navigate to the following generated API for C functions, struct, and union:

```
fun struct_by_value(s: kotlinx.cinterop.CValue<interop.MyStruct>)
fun struct_by_pointer(s: kotlinx.cinterop.CValuesRef<interop.MyStruct>?)

fun union_by_value(u: kotlinx.cinterop.CValue<interop.MyUnion>)
fun union_by_pointer(u: kotlinx.cinterop.CValuesRef<interop.MyUnion>?)
```

Technically, there is no difference between struct and union types on the Kotlin side. The cinterop tool generates Kotlin types for both struct and union C declarations.

The generated API includes fully qualified package names for `CValue<T>` and `CValuesRef<T>`, reflecting their location in `kotlinx.cinterop`. `CValue<T>` represents a by-value structure parameter, while `CValuesRef<T>?` is used to pass a pointer to a structure or a union.

Use struct and union types from Kotlin

Using C struct and union types from Kotlin is straightforward thanks to the generated API. The only question is how to create new instances of these types.

Let's take a look at the generated functions that take `MyStruct` and `MyUnion` as parameters. By-value parameters are represented as `kotlinx.cinterop.CValue<T>`, while pointer-typed parameters use `kotlinx.cinterop.CValuesRef<T>?`.

Kotlin provides a convenient API for creating and working with these types. Let's explore how to use it in practice.

Create a CValue<T>

`CValue<T>` type is used to pass by-value parameters to a C function call. Use the `cValue` function to create a `CValue<T>` instance. The function requires a lambda function with a receiver to initialize the underlying C type in-place. The function is declared as follows:

```
fun <reified T : CStructVar> cValue(initialize: T.() → Unit): CValue<T>
```

Here's how to use `cValue` and pass by-value parameters:

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.cValue

@OptIn(ExperimentalForeignApi::class)
fun callValue() {

    val cStruct = cValue<MyStruct> {
        a = 42
        b = 3.14
    }
    struct_by_value(cStruct)

    val cUnion = cValue<MyUnion> {
        b.a = 5
        b.b = 2.7182
    }

    union_by_value(cUnion)
}
```

Create struct and union as CValuesRef<T>

The `CValuesRef<T>` type is used in Kotlin to pass a pointer-typed parameter of a C function. To allocate `MyStruct` and `MyUnion` in the native memory, use the following extension function on `kotlinx.cinterop.NativePlacement` type:

```
fun <reified T : kotlinx.cinterop.CVariable> alloc(): T
```

`NativePlacement` represents native memory with functions similar to `malloc` and `free`. There are several implementations of `NativePlacement`:

- The global implementation is `kotlinx.cinterop.nativeHeap`, but you must call `nativeHeap.free()` to release the memory after use.

- A safer alternative is `memScoped()`, which creates a short-lived memory scope where all allocations are automatically freed at the end of the block:

```
fun <R> memScoped(block: kotlinx.cinterop.MemScope.() → R): R
```

With `memScoped()`, your code for calling functions with pointers can look like this:

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.memScoped
import kotlinx.cinterop.alloc
import kotlinx.cinterop.ptr

@OptIn(ExperimentalForeignApi::class)
fun callRef() {
    memScoped {
        val cStruct = alloc<MyStruct>()
        cStruct.a = 42
        cStruct.b = 3.14

        struct_by_pointer(cStruct.ptr)

        val cUnion = alloc<MyUnion>()
        cUnion.b.a = 5
        cUnion.b.b = 2.7182

        union_by_pointer(cUnion.ptr)
    }
}
```

Here, the `ptr` extension property, which is available within the `memScoped {}` block, converts `MyStruct` and `MyUnion` instances into native pointers.

Since memory is managed inside the `memScoped {}` block, it's automatically freed at the end of the block. Avoid using pointers outside of this scope to prevent accessing deallocated memory. If you need longer-lived allocations (for example, for caching in a C library), consider using `Arena()` or `nativeHeap`.

Conversion between CValue<T> and CValuesRef<T>

Sometimes you need to pass a struct as a value in one function call and then pass the same struct as a reference in another.

To do this, you'll need a `NativePlacement`, but first, let's see how `CValue<T>` is turned into a pointer:

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.cValue
import kotlinx.cinterop.memScoped

@OptIn(ExperimentalForeignApi::class)
fun callMix_ref() {
    val cStruct = cValue<MyStruct> {
        a = 42
        b = 3.14
    }

    memScoped {
        struct_by_pointer(cStruct.ptr)
    }
}
```

Here again, the `ptr` extension property from `memScoped {}` turns `MyStruct` instances into native pointers. These pointers are only valid inside the `memScoped {}` block.

To turn a pointer back into a by-value variable, call the `.readValue()` extension function:

```
import interop.*
import kotlinx.cinterop.alloc
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.memScoped
import kotlinx.cinterop.readValue

@OptIn(ExperimentalForeignApi::class)
fun callMix_value() {
    memScoped {
        val cStruct = alloc<MyStruct>()
        cStruct.a = 42
        cStruct.b = 3.14

        struct_by_value(cStruct.readValue())
    }
}
```

Update Kotlin code

Now that you've learned how to use C declarations in Kotlin code, try to use them in your project. The final code in the `hello.kt` file may look like this:

```
import interop.*
import kotlinx.cinterop.alloc
import kotlinx.cinterop.cValue
import kotlinx.cinterop.memScoped
import kotlinx.cinterop.ptr
import kotlinx.cinterop.readValue
import kotlinx.cinterop.ExperimentalForeignApi

@OptIn(ExperimentalForeignApi::class)
fun main() {
    println("Hello Kotlin/Native!")

    val cUnion = cValue<MyUnion> {
        b.a = 5
        b.b = 2.7182
    }

    memScoped {
        union_by_value(cUnion)
        union_by_pointer(cUnion.ptr)
    }

    memScoped {
        val cStruct = alloc<MyStruct> {
            a = 42
            b = 3.14
        }

        struct_by_value(cStruct.readValue())
        struct_by_pointer(cStruct.ptr)
    }
}
```

To verify that everything works as expected, run the `runDebugExecutable<YourTargetName>` Gradle task in your IDE or use the console command in your terminal, in this example:

```
./gradlew runDebugExecutableMacosArm64
```

Next step

In the next part of the series, you'll learn how function pointers are mapped between Kotlin and C:

Previous step

Next step

See also

Learn more in the Interoperability with C documentation that covers more advanced scenarios.

Mapping function pointers from C – tutorial

> The C libraries import is in Beta. All Kotlin declarations generated by the cinterop tool from C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.

Let's explore which C function pointers are visible from Kotlin and examine advanced C interop-related use cases of Kotlin/Native and multiplatform Gradle builds.

In this tutorial, you'll:

- Learn how to pass Kotlin function as a C function pointer
- Use C function pointers from Kotlin

Mapping function pointer types from C

To understand the mapping between Kotlin and C, let's declare two functions: one that accepts a function pointer as a parameter and another that returns a function pointer.

In the first part of the series, you've already created a C library with the necessary files. For this step, update the declarations in the `interop.def` file after the `---` separator:

```
---

int myFun(int i) {
  return i+1;
}

typedef int  (*MyFun)(int);

void accept_fun(MyFun f) {
  f(42);
}

MyFun supply_fun() {
  return myFun;
}
```

The `interop.def` file provides everything necessary to compile, run, or open the application in an IDE.

Inspect generated Kotlin APIs for a C library

Let's see how C function pointers are mapped into Kotlin/Native and update your project:

1. In `src/nativeMain/kotlin`, update your `hello.kt` file from the previous tutorial with the following content:

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi

@OptIn(ExperimentalForeignApi::class)
fun main() {
    println("Hello Kotlin/Native!")

    accept_fun(/* fix me*/)
    val useMe = supply_fun()
}
```

2. Use IntelliJ IDEA's Go to declaration command (Cmd + B/Ctrl + B) to navigate to the following generated API for C functions:

```
fun myFun(i: kotlin.Int): kotlin.Int
fun accept_fun(f: kotlinx.cinterop.CPointer<kotlinx.cinterop.CFunction<(kotlin.Int) →
kotlin.Int>>? /* from: interop.MyFun? */)
fun supply_fun(): kotlinx.cinterop.CPointer<kotlinx.cinterop.CFunction<(kotlin.Int) →
kotlin.Int>>? /* from: interop.MyFun? */
```

As you can see, C function pointers are represented in Kotlin using `CPointer<CFunction<...>>`. The `accept_fun()` function takes an optional function pointer as a parameter, while `supply_fun()` returns a function pointer.

`CFunction<(Int) -> Int>` represents the function signature, and `CPointer<CFunction<...>>?` represents a nullable function pointer. There is an `.invoke()` operator extension function available for all `CPointer<CFunction<...>>` types, allowing you to call function pointers as if they were regular Kotlin functions.

Pass Kotlin function as a C function pointer

It's time to try using C functions from Kotlin code. Call the `accept_fun()` function and pass the C function pointer to a Kotlin lambda:

```
import interop.*
import kotlinx.cinterop.staticCFunction
import kotlinx.cinterop.ExperimentalForeignApi

@OptIn(ExperimentalForeignApi::class)
fun myFun() {
    accept_fun(staticCFunction<Int, Int> { it + 1 })
}
```

This call uses the `staticCFunction {}` helper function from Kotlin/Native to wrap a Kotlin lambda function into a C function pointer. It allows only unbound and non-capturing lambda functions. For example, it cannot capture a local variable from the function, only globally visible declarations.

Ensure that the function doesn't throw any exceptions. Throwing exceptions from a `staticCFunction {}` causes non-deterministic side effects.

Use the C function pointer from Kotlin

The next step is to invoke a C function pointer returned from the `supply_fun()` call:

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.invoke

@OptIn(ExperimentalForeignApi::class)
fun myFun2() {
    val functionFromC = supply_fun() ?: error("No function is returned")

    functionFromC(42)
}
```

Kotlin turns the function pointer return type into a nullable `CPointer<CFunction<>>` object. You need to first explicitly check for `null`, which is why the Elvis operator is used in the code above. The cinterop tool allows you to call a C function pointer as a regular Kotlin function call: `functionFromC(42)`.

Update Kotlin code

Now that you've seen all the definitions, try to use them in your project. The code in the `hello.kt` file may look like this:

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.invoke
import kotlinx.cinterop.staticCFunction

@OptIn(ExperimentalForeignApi::class)
fun main() {
    println("Hello Kotlin/Native!")

    val cFunctionPointer = staticCFunction<Int, Int> { it + 1 }
    accept_fun(cFunctionPointer)

    val funFromC = supply_fun() ?: error("No function is returned")
    funFromC(42)
}
```

To verify that everything works as expected, run the `runDebugExecutable<YourTargetName>` Gradle task in your IDE or use the console command in your terminal, in this example:

```
./gradlew runDebugExecutableMacosArm64
```

Next step

In the next part of the series, you'll learn how strings are mapped between Kotlin and C:

Previous step

Next step

See also

Learn more in the Interoperability with C documentation that covers more advanced scenarios.

Mapping strings from C – tutorial

> The C libraries import is in Beta. All Kotlin declarations generated by the cinterop tool from C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.

In the final part of the series, let's see how to deal with C strings in Kotlin/Native.

In this tutorial, you'll learn how to:

- Pass a Kotlin string to C
- Read a C string in Kotlin
- Receive C string bytes into a Kotlin string

Working with C strings

C doesn't have a dedicated string type. Method signatures or documentation can help you identify whether a given `char *` represents a C string in a particular context.

Strings in the C language are null-terminated, so a trailing zero character `\0` is added to the end of a byte sequence to mark the end of a string. Usually, UTF-8 encoded strings are used. The UTF-8 encoding uses variable-width characters and is backward-compatible with ASCII. Kotlin/Native uses UTF-8 character encoding by default.

To understand how strings are mapped between Kotlin and C, first create the library headers. In the first part of the series, you've already created a C library with the necessary files. For this step:

1. Update your `lib.h` file with the following function declarations that work with C strings:

   ```
   #ifndef LIB2_H_INCLUDED
   #define LIB2_H_INCLUDED

   void pass_string(char* str);
   char* return_string();
   int copy_string(char* str, int size);

   #endif
   ```

   This example shows common ways to pass or receive a string in the C language. Handle the return value of the `return_string()` function carefully. Ensure you use the correct `free()` function to release the returned `char*`.

2. Update the declarations in the `interop.def` file after the `---` separator:

```
---

void pass_string(char* str) {
}

char* return_string() {
  return "C string";
}

int copy_string(char* str, int size) {
    *str++ = 'C';
    *str++ = ' ';
    *str++ = 'K';
    *str++ = '/';
    *str++ = 'N';
    *str++ = 0;
    return 0;
}
```

The `interop.def` file provides everything necessary to compile, run, or open the application in an IDE.

Inspect generated Kotlin APIs for a C library

Let's see how C string declarations are mapped into Kotlin/Native:

1. In `src/nativeMain/kotlin`, update your `hello.kt` file from the previous tutorial with the following content:

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi

@OptIn(ExperimentalForeignApi::class)
fun main() {
    println("Hello Kotlin/Native!")

    pass_string(/*fix me*/)
    val useMe = return_string()
    val useMe2 = copy_string(/*fix me*/)
}
```

2. Use IntelliJ IDEA's Go to declaration command (Cmd + B/Ctrl + B) to navigate to the following generated API for C functions:

```
fun pass_string(str: kotlinx.cinterop.CValuesRef<kotlinx.cinterop.ByteVarOf<kotlin.Byte> /* from:
kotlinx.cinterop.ByteVar */>?)
fun return_string(): kotlinx.cinterop.CPointer<kotlinx.cinterop.ByteVarOf<kotlin.Byte> /* from:
kotlinx.cinterop.ByteVar */>?
fun copy_string(str: kotlinx.cinterop.CValuesRef<kotlinx.cinterop.ByteVarOf<kotlin.Byte> /* from:
kotlinx.cinterop.ByteVar */>?, size: kotlin.Int): kotlin.Int
```

These declarations are straightforward. In Kotlin, C `char *` pointers are mapped into `str: CValuesRef<ByteVarOf>?` for parameters and into `CPointer<ByteVarOf>?` for return types. Kotlin represents the `char` type as `kotlin.Byte`, as it's usually an 8-bit signed value.

In the generated Kotlin declarations, `str` is defined as `CValuesRef<ByteVarOf<Byte>>?`. Since this type is nullable, you can pass `null` as the argument value.

Pass Kotlin strings to C

Let's try to use the API from Kotlin. Call the `pass_string()` function first:

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.cstr

@OptIn(ExperimentalForeignApi::class)
fun passStringToC() {
    val str = "This is a Kotlin string"
    pass_string(str.cstr)
}
```

Passing a Kotlin string to C is straightforward, thanks to the `String.cstr` extension property. There is also the `String.wcstr` property for cases that involve UTF-16 characters.

Read C strings in Kotlin

Now take a returned `char *` from the `return_string()` function and turn it into a Kotlin string:

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.toKString

@OptIn(ExperimentalForeignApi::class)
fun passStringToC() {
    val stringFromC = return_string()?.toKString()

    println("Returned from C: $stringFromC")
}
```

Here, the `.toKString()` extension function converts a C string returned from the `return_string()` function into a Kotlin string.

Kotlin provides several extension functions for converting C `char *` strings into Kotlin strings, depending on the encoding:

```
fun CPointer<ByteVarOf<Byte>>.toKString(): String // Standard function for UTF-8 strings
fun CPointer<ByteVarOf<Byte>>.toKStringFromUtf8(): String // Explicitly converts UTF-8 strings
fun CPointer<ShortVarOf<Short>>.toKStringFromUtf16(): String // Converts UTF-16 encoded strings
fun CPointer<IntVarOf<Int>>.toKStringFromUtf32(): String // Converts UTF-32 encoded strings
```

Receive C string bytes from Kotlin

This time, use the `copy_string()` C function to write a C string to a given buffer. It takes two arguments: a pointer to the memory location where the string should be written and the allowed buffer size.

The function should also return something to indicate if it has succeeded or failed. Let's assume `0` means it succeeded, and the supplied buffer was big enough:

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.addressOf
import kotlinx.cinterop.usePinned

@OptIn(ExperimentalForeignApi::class)
fun sendString() {
    val buf = ByteArray(255)
    buf.usePinned { pinned ->
        if (copy_string(pinned.addressOf(0), buf.size - 1) != 0) {
            throw Error("Failed to read string from C")
        }
    }

    val copiedStringFromC = buf.decodeToString()
    println("Message from C: $copiedStringFromC")
}
```

Here, a native pointer is passed to the C function first. The `.usePinned()` extension function temporarily pins the native memory address of the byte array. The C function fills in the byte array with data. Another extension function, `ByteArray.decodeToString()`, turns the byte array into a Kotlin string, assuming UTF-8 encoding.

Update Kotlin code

Now that you've learned how to use C declarations in Kotlin code, try to use them in your project. The code in the final `hello.kt` file may look like this:

```
import interop.*
import kotlinx.cinterop.*

@OptIn(ExperimentalForeignApi::class)
fun main() {
    println("Hello Kotlin/Native!")

    val str = "This is a Kotlin string"
    pass_string(str.cstr)

    val useMe = return_string()?.toKString() ?: error("null pointer returned")
    println(useMe)

    val copyFromC = ByteArray(255).usePinned { pinned ->
        val useMe2 = copy_string(pinned.addressOf(0), pinned.get().size - 1)
        if (useMe2 != 0) throw Error("Failed to read a string from C")
        pinned.get().decodeToString()
    }

    println(copyFromC)
}
```

To verify that everything works as expected, run the `runDebugExecutable<YourTargetName>` Gradle task in your IDE or use the console command in your terminal, in this example:

```
./gradlew runDebugExecutableMacosArm64
```

- Previous step

What's next

Learn more in the Interoperability with C documentation that covers more advanced scenarios.

Kotlin/Native as a dynamic library – tutorial

You can create dynamic libraries to use Kotlin code from existing programs. This enables code sharing across many platforms or languages, including JVM, Python, Android, and others.

> For iOS and other Apple targets, we recommend generating a framework. See the Kotlin/Native as an Apple framework tutorial.

You can use the Kotlin/Native code from existing native applications or libraries. For this, you need to compile the Kotlin code into a dynamic library in the `.so`, `.dylib`, or `.dll` format.

In this tutorial, you will:

- Compile Kotlin code to a dynamic library
- Examine generated C headers
- Use the Kotlin dynamic library from C
- Compile and run the project

You can use the command line to generate a Kotlin library, either directly or with a script file (such as `.sh` or `.bat` file). However, this approach doesn't scale well for larger projects that have hundreds of files and libraries. Using a build system simplifies the process by downloading and caching the Kotlin/Native compiler binaries and libraries with transitive dependencies, as well as by running the compiler and tests. Kotlin/Native can use the Gradle build system through the Kotlin Multiplatform plugin.

Let's examine the advanced C interop-related usages of Kotlin/Native and Kotlin Multiplatform builds with Gradle.

> If you use a Mac and want to create and run applications for macOS or other Apple targets, you also need to install the Xcode Command Line Tools, launch it, and accept the license terms first.

Create a Kotlin library

The Kotlin/Native compiler can produce a dynamic library from the Kotlin code. A dynamic library often comes with a `.h` header file, which you use to call the compiled code from C.

Let's create a Kotlin library and use it from a C program.

> See the Get started with Kotlin/Native tutorial for detailed first steps and instructions on how to create a new Kotlin/Native project and open it in IntelliJ IDEA.

1. Navigate to the `src/nativeMain/kotlin` directory and create the `lib.kt` file with the following library contents:

```
package example

object Object {
    val field = "A"
}

class Clazz {
    fun memberFunction(p: Int): ULong = 42UL
}

fun forIntegers(b: Byte, s: Short, i: UInt, l: Long) { }
fun forFloats(f: Float, d: Double) { }

fun strings(str: String) : String? {
    return "That is '$str' from C"
}

val globalString = "A global String"
```

2. Update your `build.gradle(.kts)` Gradle build file with the following:

Kotlin

```
import org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget

plugins {
    kotlin("multiplatform") version "2.4.20"
}

repositories {
    mavenCentral()
}

kotlin {
    macosArm64()    // macOS on Apple Silicon
    // linuxArm64() // Linux on ARM64 platforms
    // linuxX64()   // Linux on x86_64 platforms
    // mingwX64()   // on Windows

    targets.withType<KotlinNativeTarget>().configureEach {
        binaries {
            sharedLib {
                baseName = "native"       // macOS
                // baseName = "native"    // Linux
                // baseName = "libnative" // Windows
            }
        }
    }
}

tasks.wrapper {
    gradleVersion = "9.7.0"
    distributionType = Wrapper.DistributionType.ALL
}
```

Groovy

```
import org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget

plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.4.20'
}

repositories {
    mavenCentral()
}

kotlin {
    macosArm64()    // Apple Silicon macOS
    // linuxArm64() // Linux on ARM64 platforms
    // linuxX64()   // Linux on x86_64 platforms
    // mingwX64()   // Windows

    targets.withType(KotlinNativeTarget).configureEach {
        binaries {
            sharedLib {
                baseName = "native"       // macOS
                // baseName = "native"    // Linux
                // baseName = "libnative" // Windows
            }
        }
    }
}

wrapper {
    gradleVersion = "9.7.0"
    distributionType = "ALL"
}
```

- The `binaries {}` block configures the project to generate a dynamic or shared library.
- `libnative` is used as the library name, the prefix for the generated header file name. It also prefixes all declarations in the header file.

3. To build the library, run the `linkDebugShared<YourTargetName>` Gradle task in your IDE or use the console command in your terminal, in this example:

```
./gradlew linkDebugSharedMacosArm64
```

The build generates the library into the `build/bin/<yourTargetName>/debugShared` directory with the following files:

- macOS: `libnative_api.h` and `libnative.dylib`
- Linux: `libnative_api.h` and `libnative.so`
- Windows: `libnative_api.h`, `libnative.def`, and `libnative.dll`

> You can also use the `linkNative` Gradle task to generate both `debug` and `release` variants of the library.

The Kotlin/Native compiler uses the same rules to generate the `.h` file for all platforms. Let's check out the C API of the Kotlin library.

Generated header file

Let's examine how Kotlin/Native declarations are mapped to C functions.

In the `build/bin/<yourTargetName>/debugShared` directory, open the `libnative_api.h` header file. The very first part contains the standard C/C++ header and footer:

```
#ifndef KONAN_LIBNATIVE_H
#define KONAN_LIBNATIVE_H
#ifdef __cplusplus
extern "C" {
#endif

/// The rest of the generated code

#ifdef __cplusplus
}  /* extern "C" */
#endif
#endif  /* KONAN_LIBNATIVE_H */
```

Following this, the `libnative_api.h` includes a block with the common type definitions:

```
#ifdef __cplusplus
typedef bool            libnative_KBoolean;
#else
typedef _Bool           libnative_KBoolean;
#endif
typedef unsigned short     libnative_KChar;
typedef signed char        libnative_KByte;
typedef short              libnative_KShort;
typedef int                libnative_KInt;
typedef long long          libnative_KLong;
typedef unsigned char      libnative_KUByte;
typedef unsigned short     libnative_KUShort;
typedef unsigned int       libnative_KUInt;
typedef unsigned long long libnative_KULong;
typedef float              libnative_KFloat;
typedef double             libnative_KDouble;
typedef float __attribute__ ((__vector_size__ (16))) libnative_KVector128;
typedef void*              libnative_KNativePtr;
```

Kotlin uses the `libnative_` prefix for all declarations in the created `libnative_api.h` file. Here's the complete list of type mappings:

Kotlin definition	C type
`libnative_KBoolean`	`bool` or `_Bool`
`libnative_KChar`	`unsigned short`
`libnative_KByte`	`signed char`

Kotlin definition	C type
`libnative_KShort`	`short`
`libnative_KInt`	`int`
`libnative_KLong`	`long long`
`libnative_KUByte`	`unsigned char`
`libnative_KUShort`	`unsigned short`
`libnative_KUInt`	`unsigned int`
`libnative_KULong`	`unsigned long long`
`libnative_KFloat`	`float`
`libnative_KDouble`	`double`
`libnative_KVector128`	`float __attribute__ ((__vector_size__ (16))`
`libnative_KNativePtr`	`void*`

The definition section of the `libnative_api.h` file shows how Kotlin primitive types are mapped to C primitive types. The Kotlin/Native compiler generates these entries automatically for every library. The reverse mapping is described in the Mapping primitive data types from C tutorial.

After the automatically generated type definitions, you'll find the separate type definitions used in your library:

```
struct libnative_KType;
typedef struct libnative_KType libnative_KType;

/// Automatically generated type definitions

typedef struct {
  libnative_KNativePtr pinned;
} libnative_kref_example_Object;
typedef struct {
  libnative_KNativePtr pinned;
} libnative_kref_example_Clazz;
```

In C, the `typedef struct { ... } TYPE_NAME` syntax declares the structure.

> See this StackOverflow thread for more explanations of this pattern.

As you can see from these definitions, Kotlin types are mapped using the same pattern: `Object` is mapped to `libnative_kref_example_Object`, and `Clazz` is mapped to `libnative_kref_example_Clazz`. All structs

contain nothing but the `pinned` field with a pointer. The field type `libnative_KNativePtr` is defined as `void*` earlier in the file.

Since C doesn't support namespaces, the Kotlin/Native compiler generates long names to avoid any possible clashes with other symbols in the existing native project.

Service runtime functions

The `libnative_ExportedSymbols` structure defines all the functions provided by Kotlin/Native and your library. It uses nested anonymous structures heavily to mimic packages. The `libnative_` prefix comes from the library name.

`libnative_ExportedSymbols` includes several helper functions in the header file:

```
typedef struct {
  /* Service functions. */
  void (*DisposeStablePointer)(libnative_KNativePtr ptr);
  void (*DisposeString)(const char* string);
```

These functions deal with Kotlin/Native objects. `DisposeStablePointer` is called to release a reference to a Kotlin object, and `DisposeString` is called to release a Kotlin string, which has the `char*` type in C.

The next part of the `libnative_api.h` file consists of structure declarations of runtime functions:

```
libnative_KBoolean (*IsInstance)(libnative_KNativePtr ref, const libnative_KType* type);
libnative_KBoolean (*IsInstance)(libnative_KNativePtr ref, const libnative_KType* type);
libnative_kref_kotlin_Byte (*createNullableByte)(libnative_KByte);
libnative_KByte (*getNonNullValueOfByte)(libnative_kref_kotlin_Byte);
libnative_kref_kotlin_Short (*createNullableShort)(libnative_KShort);
libnative_KShort (*getNonNullValueOfShort)(libnative_kref_kotlin_Short);
libnative_kref_kotlin_Int (*createNullableInt)(libnative_KInt);
libnative_KInt (*getNonNullValueOfInt)(libnative_kref_kotlin_Int);
libnative_kref_kotlin_Long (*createNullableLong)(libnative_KLong);
libnative_KLong (*getNonNullValueOfLong)(libnative_kref_kotlin_Long);
libnative_kref_kotlin_Float (*createNullableFloat)(libnative_KFloat);
libnative_KFloat (*getNonNullValueOfFloat)(libnative_kref_kotlin_Float);
libnative_kref_kotlin_Double (*createNullableDouble)(libnative_KDouble);
libnative_KDouble (*getNonNullValueOfDouble)(libnative_kref_kotlin_Double);
libnative_kref_kotlin_Char (*createNullableChar)(libnative_KChar);
libnative_KChar (*getNonNullValueOfChar)(libnative_kref_kotlin_Char);
libnative_kref_kotlin_Boolean (*createNullableBoolean)(libnative_KBoolean);
libnative_KBoolean (*getNonNullValueOfBoolean)(libnative_kref_kotlin_Boolean);
libnative_kref_kotlin_Unit (*createNullableUnit)(void);
libnative_kref_kotlin_UByte (*createNullableUByte)(libnative_KUByte);
libnative_KUByte (*getNonNullValueOfUByte)(libnative_kref_kotlin_UByte);
libnative_kref_kotlin_UShort (*createNullableUShort)(libnative_KUShort);
libnative_KUShort (*getNonNullValueOfUShort)(libnative_kref_kotlin_UShort);
libnative_kref_kotlin_UInt (*createNullableUInt)(libnative_KUInt);
libnative_KUInt (*getNonNullValueOfUInt)(libnative_kref_kotlin_UInt);
libnative_kref_kotlin_ULong (*createNullableULong)(libnative_KULong);
libnative_KULong (*getNonNullValueOfULong)(libnative_kref_kotlin_ULong);
```

You can use the `IsInstance` function to check if a Kotlin object (referenced with its `.pinned` pointer) is an instance of a type. The actual set of operations generated depends on actual usages.

> Kotlin/Native has its own garbage collector, but it doesn't manage Kotlin objects accessed from C. However, Kotlin/Native provides interoperability with Swift/Objective-C, and the garbage collector is integrated with Swift/Objective-C ARC.

Your library functions

Let's take a look at the separate structure declarations used in your library. The `libnative_kref_example` field mimics the package structure of your Kotlin code with a `libnative_kref.` prefix:

```
typedef struct {
  /* User functions. */
  struct {
    struct {
      struct {
        struct {
          libnative_KType* (*_type)(void);
          libnative_kref_example_Object (*_instance)();
          const char* (*get_field)(libnative_kref_example_Object thiz);
        } Object;
        struct {
          libnative_KType* (*_type)(void);
          libnative_kref_example_Clazz (*Clazz)();
          libnative_KULong (*memberFunction)(libnative_kref_example_Clazz thiz, libnative_KInt p);
        } Clazz;
        const char* (*get_globalString)();
        void (*forFloats)(libnative_KFloat f, libnative_KDouble d);
        void (*forIntegers)(libnative_KByte b, libnative_KShort s, libnative_KUInt i, libnative_KLong
l);
        const char* (*strings)(const char* str);
      } example;
    } root;
  } kotlin;
} libnative_ExportedSymbols;
```

The code uses anonymous structure declarations. Here, `struct { ... } foo` declares a field in the outer struct of the anonymous structure type, which has no name.

Since C doesn't support objects either, function pointers are used to mimic object semantics. A function pointer is declared as `RETURN_TYPE (* FIELD_NAME)(PARAMETERS)`.

The `libnative_kref_example_Clazz` field represents the `Clazz` from Kotlin. The `libnative_KULong` is accessible with the `memberFunction` field. The only difference is that the `memberFunction` accepts a `thiz` reference as the first parameter. Since C doesn't support objects, the `thiz` pointer is passed explicitly.

There is a constructor in the `Clazz` field (aka `libnative_kref_example_Clazz_Clazz`), which acts as the constructor function to create an instance of the `Clazz`.

The Kotlin `object Object` is accessible as `libnative_kref_example_Object`. The `_instance` function retrieves the only instance of the object.

Properties are translated into functions. The `get_` and `set_` prefixes name the getter and the setter functions, respectively. For example, the read-only property `globalString` from Kotlin is turned into a `get_globalString` function in C.

Global functions `forFloats`, `forIntegers`, and `strings` are turned into functions pointers in the `libnative_kref_example` anonymous struct.

Entry point

Now you know how the API is created, the initialization of the `libnative_ExportedSymbols` structure is the starting point. Let's then take a look at the final part of the `libnative_api.h`:

```
extern libnative_ExportedSymbols* libnative_symbols(void);
```

The `libnative_symbols` function allows you to open the gateway from the native code to the Kotlin/Native library. This is the entry point for accessing the library. The library name is used as a prefix for the function name.

> It might be necessary to host the returned `libnative_ExportedSymbols*` pointer per thread.

Use generated headers from C

Using the generated headers from C is straightforward. In the library directory, create the `main.c` file with the following code:

```
#include "libnative_api.h"
#include "stdio.h"

int main(int argc, char** argv) {
  // Obtain reference for calling Kotlin/Native functions
  libnative_ExportedSymbols* lib = libnative_symbols();

  lib->kotlin.root.example.forIntegers(1, 2, 3, 4);
  lib->kotlin.root.example.forFloats(1.0f, 2.0);

  // Use C and Kotlin/Native strings
  const char* str = "Hello from Native!";
  const char* response = lib->kotlin.root.example.strings(str);
  printf("in: %s\nout:%s\n", str, response);
  lib->DisposeString(response);

  // Create Kotlin object instance
  libnative_kref_example_Clazz newInstance = lib->kotlin.root.example.Clazz.Clazz();
  long x = lib->kotlin.root.example.Clazz.memberFunction(newInstance, 42);
  lib->DisposeStablePointer(newInstance.pinned);

  printf("DemoClazz returned %ld\n", x);

  return 0;
}
```

Compile and run the project

On macOS

To compile the C code and link it with the dynamic library, navigate to the library directory and run the following command:

```
clang main.c libnative.dylib
```

The compiler generates an executable called `a.out`. Run it to execute the Kotlin code from the C library.

On Linux

To compile the C code and link it with the dynamic library, navigate to the library directory and run the following command:

```
gcc main.c libnative.so
```

The compiler generates an executable called `a.out`. Run it to execute the Kotlin code from the C library. On Linux, you need to include `.` into the `LD_LIBRARY_PATH` to let the application know to load the `libnative.so` library from the current folder.

On Windows

First, you'll need to install a Microsoft Visual C++ compiler that supports the x64_64 target.

The easiest way to do this is to install Microsoft Visual Studio on a Windows machine. During installation, select the necessary components to work with C++, for example, **Desktop development with C++**.

On Windows, you can include dynamic libraries either by generating a static library wrapper or manually with the LoadLibrary or similar Win32API functions.

Let's use the first option and generate the static wrapper library for the `libnative.dll`:

1. Call `lib.exe` from the toolchain to generate the static library wrapper `libnative.lib` that automates the DLL usage from the code:

   ```
   lib /def:libnative.def /out:libnative.lib
   ```

2. Compile your `main.c` into an executable. Include the generated `libnative.lib` into the build command and start:

   ```
   cl.exe main.c libnative.lib
   ```

   The command produces the `main.exe` file, which you can run.

What's next

- Learn more about interoperability with Swift/Objective-C
- Check out the Kotlin/Native as an Apple framework tutorial

Create an app using C interop and libcurl – tutorial

> The C libraries import is in Beta. All Kotlin declarations generated by the cinterop tool from C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.

This tutorial demonstrates how to use IntelliJ IDEA to create a command-line application. You'll learn how to create a simple HTTP client that can run natively on specified platforms using Kotlin/Native and the libcurl library.

The output will be an executable command-line app that you can run on macOS and Linux and make simple HTTP GET requests.

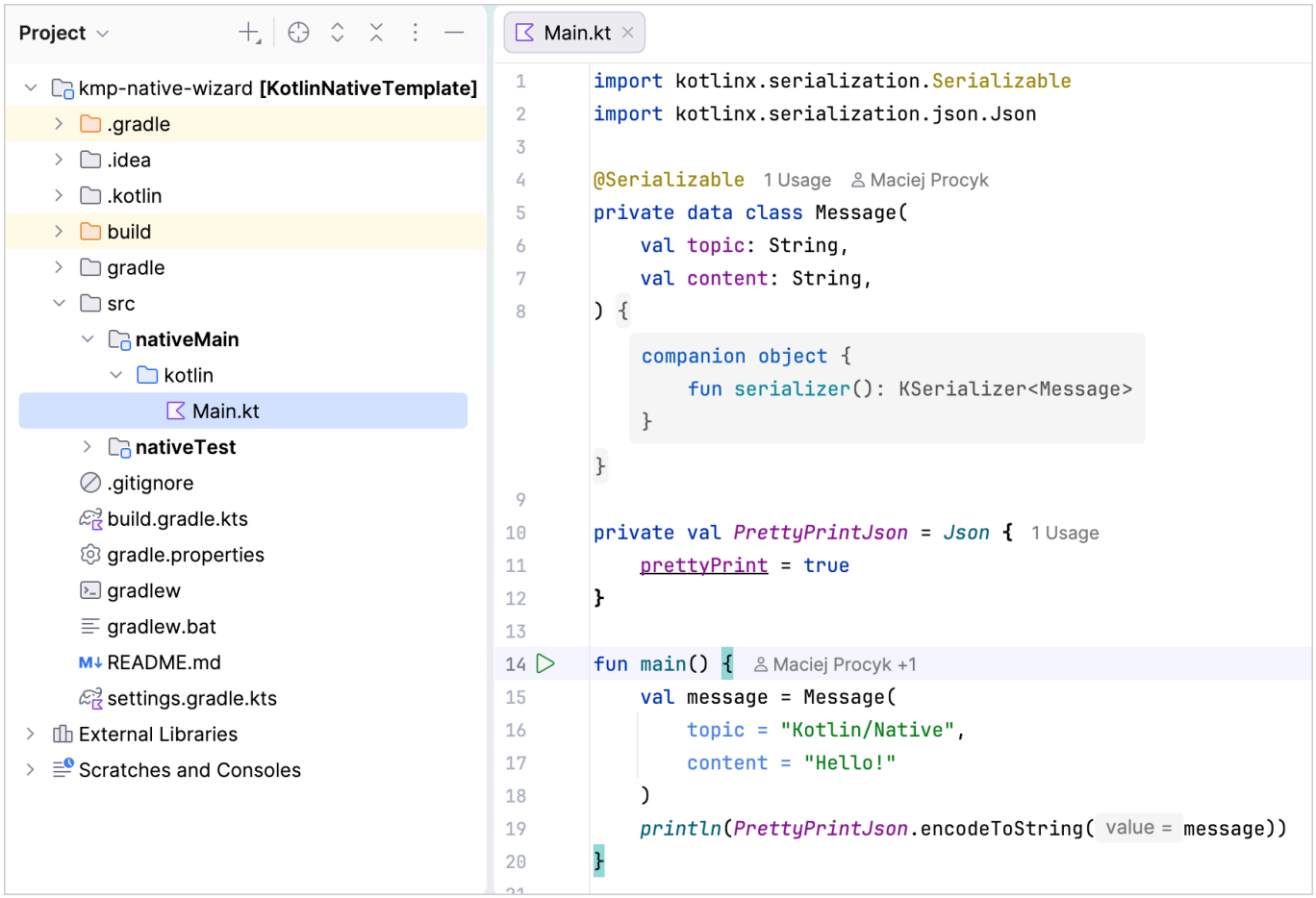
You can use the command line to generate a Kotlin library, either directly or with a script file (such as `.sh` or `.bat` file). However, this approach doesn't scale well for larger projects that have hundreds of files and libraries. Using a build system simplifies the process by downloading and caching the Kotlin/Native compiler binaries and libraries with transitive dependencies, as well as by running the compiler and tests. Kotlin/Native can use the Gradle build system through the Kotlin Multiplatform plugin.

Before you start

1. Download and install the latest version of IntelliJ IDEA.
2. Clone the project template by selecting **File** | **New** | **Project from Version Control** in IntelliJ IDEA and using this URL:

   ```
   https://github.com/Kotlin/kmp-native-wizard
   ```

3. Explore the project structure:



Native application project structure

The template includes a project with the files and folders you need to get started. It's important to understand that an application written in Kotlin/Native can target different platforms if the code does not have platform-specific requirements. Your code is placed in the `nativeMain` directory with a corresponding `nativeTest`. For this tutorial, keep the folder structure as is.

4. Open the `build.gradle.kts` file, the build script that contains the project settings. Pay special attention to the following in the build file:

```
import org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget

kotlin {
    macosArm64()
    linuxArm64()
    linuxX64()
    mingwX64()

    targets.withType<KotlinNativeTarget>().configureEach {
        binaries {
            executable {
                entryPoint = "main"
            }
        }
    }
}
```

- Targets are defined using `macosArm64`, `linuxArm64`, `linuxX64`, and `mingwX64` for macOS, Linux, and Windows. See the complete list of supported platforms.
- The `binaries {}` block defines how the binary is generated and the entry point of the application. These can be left as default values.
- C interoperability is configured as an additional step in the build. By default, all the symbols from C are imported to the `interop` package. You may want to import the whole package in `.kt` files. Learn more about how to configure it.

Create a definition file

When writing native applications, you often need access to certain functionalities that are not included in the Kotlin standard library, such as making HTTP requests, reading and writing from disk, and so on.

Kotlin/Native helps consume standard C libraries, opening up an entire ecosystem of functionality that exists for pretty much anything you may need. Kotlin/Native is already shipped with a set of prebuilt platform libraries, which provide some additional common functionality to the standard library.

An ideal scenario for interop is to call C functions as if you are calling Kotlin functions, following the same signature and conventions. This is when the cinterop tool comes in handy. It takes a C library and generates the corresponding Kotlin bindings, so that the library can be used as if it were Kotlin code.

To generate these bindings, each library needs a definition file, usually with the same name as the library. This is a property file that describes exactly how the library should be consumed.

In this app, you'll need the libcurl library to make some HTTP calls. To create its definition file:

1. Select the `src` folder and create a new directory with **File | New | Directory**.
2. Name the new directory **nativeInterop/cinterop**. This is the default convention for header file locations, though it can be overridden in the `build.gradle.kts` file if you use a different location.
3. Select this new subfolder and create a new `libcurl.def` file with **File | New | File**.
4. Update your file with the following code:

```
headers = curl/curl.h
headerFilter = curl/*

compilerOpts.linux = -I/usr/include -I/usr/include/x86_64-linux-gnu
linkerOpts.osx = -L/opt/local/lib -L/usr/local/opt/curl/lib -lcurl
linkerOpts.linux = -L/usr/lib/x86_64-linux-gnu -lcurl
```

- `headers` is the list of header files to generate Kotlin stubs for. You can add multiple files here, separating each with a space. In this case, it's only `curl.h`. The referenced files need to be available on the specified path (in this case, it's `/usr/include/curl`).
- `headerFilter` shows what exactly is included. In C, all the headers are also included when one file references another one with the `#include` directive. Sometimes it's not necessary, and you can add this parameter using glob patterns to make adjustments.

  You can use `headerFilter` if you don't want to fetch external dependencies (such as system `stdint.h` header) into the interop library. Also, it may be useful for library size optimization and fixing potential conflicts between the system and the provided Kotlin/Native compilation environment.
- If the behavior for a certain platform needs to be modified, you can use a format like `compilerOpts.osx` or `compilerOpts.linux` to provide platform-specific values to the options. In this case, they are macOS (the `.osx` suffix) and Linux (the `.linux` suffix). Parameters without a suffix are also possible (for example, `linkerOpts=`) and applied to all platforms.

For the full list of available options, see Definition file.

> You need to have the `curl` library binaries on your system to make the sample work. On macOS and Linux, they are usually included. On Windows, you can build it from sources (you'll need Microsoft Visual Studio or Windows SDK command-line tools). For more details, see the related blog post. Alternatively, you may want to consider a MinGW/MSYS2 `curl` binary.

Add interoperability to the build process

To use header files, make sure they are generated as a part of the build process. For this, add the following `compilations {}` block to the `build.gradle.kts` file:

```
targets.withType<KotlinNativeTarget>().configureEach {
    compilations.getByName("main") {
        cinterops {
            val libcurl by creating
        }
    }
    binaries {
        executable {
            entryPoint = "main"
        }
    }
}
```

First, `cinterops` is added, and then an entry for a definition file. By default, the name of the file is used. You can override this with additional parameters:

```
cinterops {
    val libcurl by creating {
        definitionFile.set(project.file("src/nativeInterop/cinterop/libcurl.def"))
        packageName("com.jetbrains.handson.http")
        compilerOpts("-I/path")
        includeDirs.allHeaders("path")
    }
}
```

Write the application code

Now that you have the library and the corresponding Kotlin stubs, you can use them from your application. For this tutorial, convert the simple.c example to Kotlin.

In the `src/nativeMain/kotlin/` folder, update your `Main.kt` file with the following code:

```
import kotlinx.cinterop.*
import libcurl.*

@OptIn(ExperimentalForeignApi::class)
fun main(args: Array<String>) {
    val curl = curl_easy_init()
    if (curl != null) {
        curl_easy_setopt(curl, CURLOPT_URL, "https://example.com")
        curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L)
        val res = curl_easy_perform(curl)
        if (res != CURLE_OK) {
            println("curl_easy_perform() failed ${curl_easy_strerror(res)?.toKString()}")
        }
        curl_easy_cleanup(curl)
    }
}
```

As you can see, explicit variable declarations are eliminated in the Kotlin version, but everything else is pretty much the same as the C version. All the calls you'd expect in the libcurl library are available in the Kotlin equivalent.

> This is a line-by-line literal translation. You could also write this in a more Kotlin idiomatic way.

Compile and run the application

1. To compile the application, run the `runDebugExecutable<YourTargetName>` Gradle task from the task list or use the console command in your terminal, for example:

   ```
   ./gradlew runDebugExecutableMacosArm64
   ```

   In this case, the part generated by the cinterop tool is implicitly included in the build.

2. If there are no errors during compilation, click the green **Run** icon in the gutter next to the `main()` function or use the Shift + Cmd + R/Shift + F10 shortcut.

   IntelliJ IDEA opens the **Run** tab and shows the output — the contents of example.com:

```
    div {
        width: 600px;
        margin: 5em auto;
        padding: 2em;
        background-color: #fdfdff;
        border-radius: 0.5em;
        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
    }
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
    }
    @media (max-width: 700px) {
        div {
            margin: 0 auto;
            width: auto;
        }
    }
    </style>
</head>

<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is for use in illustrative examples in documents. You may use this
    domain in literature without prior coordination or asking for permission.</p>
    <p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
```

Application output with HTML-code

You can see the actual output because the call `curl_easy_perform` prints the result to the standard output. You could hide this using `curl_easy_setopt`.

You can get the full project code in our GitHub repository.

What's next

Learn more about Kotlin's interoperability with C.

Definition file

Kotlin/Native enables you to consume C and Objective-C libraries, allowing you to use their functionality in Kotlin. A special tool called cinterop takes a C or an Objective-C library and generates the corresponding Kotlin bindings, so that the library's methods can be used in your Kotlin code as usual.

To generate these bindings, each library needs a definition file, usually with the same name as the library. This is a property file that describes exactly how the library should be consumed. See the full list of available properties.

Here's a general workflow when working with a project:

1. Create a `.def` file describing what to include in the bindings.
2. Use the generated bindings in your Kotlin code.
3. Run the Kotlin/Native compiler to produce the final executable.

Create and configure a definition file

Let's create a definition file and generate bindings for a C library:

1. In your IDE, select the `src` folder and create a new directory with **File | New | Directory**.
2. Name the new directory `nativeInterop/cinterop`.

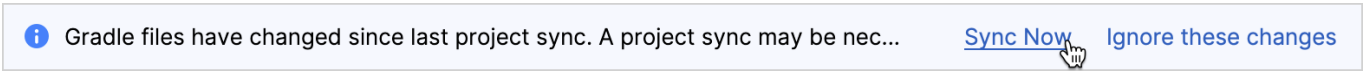
   This is the default convention for `.def` file locations, but it can be overridden in the `build.gradle.kts` file if you use a different location.
3. Select the new subfolder and create a `png.def` file with **File | New | File**.
4. Add the necessary properties:

   ```
   headers = png.h
   headerFilter = png.h
   package = png

   compilerOpts.linux = -I/usr/include -I/usr/include/x86_64-linux-gnu
   linkerOpts.osx = -L/opt/local/lib -L/usr/local/opt/png/lib -lpng
   linkerOpts.linux = -L/usr/lib/x86_64-linux-gnu -lpng
   ```

   - `headers` is the list of header files to generate Kotlin stubs for. You can add multiple files to this entry, separating each with a space. In this case, it's only `png.h`. The referenced files need to be available on the specified path (in this case, it's `/usr/include/png`).
   - `headerFilter` shows what exactly is included. In C, all the headers are also included when one file references another one with the `#include` directive. Sometimes it's not necessary, and you can add this parameter using glob patterns to make adjustments.

     You can use `headerFilter` if you don't want to fetch external dependencies (such as system `stdint.h` header) into the interop library. Also, it may be useful for library size optimization and fixing potential conflicts between the system and the provided Kotlin/Native compilation environment.
   - If the behavior for a certain platform needs to be modified, you can use a format like `compilerOpts.osx` or `compilerOpts.linux` to provide platform-specific values to the options. In this case, they are macOS (the `.osx` suffix) and Linux (the `.linux` suffix). Parameters without a suffix are also possible (for example, `linkerOpts=`) and applied to all platforms.
5. To generate bindings, synchronize the Gradle files by clicking **Sync Now** in the notification.



Synchronize the Gradle files

After the bindings generation, the IDE can use them as a proxy view of the native library.

> You can also configure bindings generation by using the cinterop tool in the command line.

Properties

Here's the full list of properties you can use in your definition file to adjust the content of the generated binaries. For more information, see the corresponding sections below.

Property	Description
`headers`	The list of headers from a library to be included in the bindings.
`modules`	The list of Clang modules from an Objective-C library to be included in the bindings.
`language`	Specifies the language. C is used by default; change to `Objective-C` if necessary.
`compilerOpts`	Compiler options that the cinterop tool passes to the C compiler.
`linkerOpts`	Linker options that the cinterop tool passes to the linker.
`excludedFunctions`	A space-separated list of function names that should be ignored.
`staticLibraries`	Experimental. Includes a static library into `.klib`.
`libraryPaths`	Experimental. A space-separated list of directories where the cinterop tool searches for the library to be included in `.klib`.
`package`	Package prefix for the generated Kotlin API.
`headerFilter`	Filters headers by globs and includes only them when importing a library.
`excludeFilter`	Excludes specific headers when importing a library and takes priority over `headerFilter`.
`strictEnums`	A space-separated list of enums that should be generated as Kotlin enums.
`nonStrictEnums`	A space-separated list of enums that should be generated as integral values.
`noStringConversion`	A space-separated list of functions whose `const char*` parameters should not be auto-converted to Kotlin `String`s.

Property	Description
`allowedOverloadsForCFunctions`	By default, it's assumed that C functions have unique names. If several functions have the same name, only one is picked. However, you can change this by specifying these functions in `allowedOverloadsForCFunctions`.
`disableDesignatedInitializerChecks`	Disables the compiler check that doesn't allow calling a non-designated Objective-C initializer as a `super()` constructor.
`foreignExceptionMode`	Wraps exceptions from Objective-C code into Kotlin exceptions with the `ForeignException` type.
`userSetupHint`	Adds a custom message, for example, to help users resolve linker errors.

In addition to the list of properties, you can include custom declarations in your definition file.

Import headers

If a C library does not have a Clang module and instead consists of a set of headers, use the `headers` property to specify headers that should be imported:

```
headers = curl/curl.h
```

Filter headers by globs

You can filter headers by globs using filter properties from the `.def` file. To include declarations from headers, use the `headerFilter` property. If a header matches any of the globs, its declarations are included in the bindings.

The globs are applied to the header paths relative to the appropriate include path elements, for example, `time.h` or `curl/curl.h`. So if the library is usually included with `#include <SomeLibrary/Header.h>`, you can probably filter headers with the following filter:

```
headerFilter = SomeLibrary/**
```

If `headerFilter` is not provided, all the headers are included. However, we encourage you to use `headerFilter` and specify the glob as precisely as possible. In this case, the generated library contains only the necessary declarations. It can help avoid various issues when upgrading Kotlin or tools in your development environment.

Exclude headers

To exclude specific headers, use the `excludeFilter` property. It can be helpful to remove redundant or problematic headers and optimize compilation, as declarations from the specified headers are not included in the bindings:

```
excludeFilter = SomeLibrary/time.h
```

If the same header is both included with `headerFilter`, and excluded with `excludeFilter`, the specified header will not be included in the bindings.

Import modules

If an Objective-C library has a Clang module, use the `modules` property to specify the module to be imported:

```
modules = UIKit
```

Set the package name

Use the `package` property to specify a package prefix for the generated Kotlin API:

```
package = png
```

If you don't specify this property, the compiler generates declarations in the root package.

The `kotlin` and `kotlinx.cinterop` names are reserved and can't be used as package prefixes.

Pass compiler and linker options

Use the `compilerOpts` property to pass options to the C compiler, which is used to analyze headers under the hood. To pass options to the linker, which is used to link final executables, use `linkerOpts`. For example:

```
compilerOpts = -DFOO=bar
linkerOpts = -lpng
```

You can also specify target-specific options that apply only to a certain target:

```
compilerOpts = -DBAR=bar
compilerOpts.linux_x64 = -DFOO=foo1
compilerOpts.macos_x64 = -DFOO=foo2
```

With this configuration, headers are analyzed using `-DBAR=bar -DFOO=foo1` on Linux and `-DBAR=bar -DFOO=foo2` on macOS. Note that any definition file option can have both common and platform-specific parts.

Ignore specific functions

Use the `excludedFunctions` property to specify a list of the function names that should be ignored. This can be useful if a function declared in the header isn't guaranteed to be callable, and it's difficult or impossible to determine this automatically. You can also use this property to work around a bug in the interop itself.

Experimental

Include a static library

Sometimes it's more convenient to ship a static library with your product, rather than assume it is available within the user's environment. To include a static library into `.klib`, use `staticLibrary` and `libraryPaths` properties:

```
headers = foo.h
staticLibraries = libfoo.a
libraryPaths = /opt/local/lib /usr/local/opt/curl/lib
```

When given the above snippet, the cinterop tool searches `libfoo.a` in `/opt/local/lib` and `/usr/local/opt/curl/lib`, and if found, includes the library binary in the `klib`.

When using a `klib` like this in your program, the library is linked automatically.

Configure enums generation

Use the `strictEnums` property to generate enums as Kotlin enums or `nonStrictEnums` to generate them as integral values. If an enum is not included in either of these lists, it is generated based on heuristics.

Set up string conversion

Use the `noStringConversion` property to disable automatic conversion of the `const char*` function parameters as Kotlin `String`s.

Allow calling a non-designated initializer

By default, the Kotlin/Native compiler doesn't allow calling a non-designated Objective-C initializer as a `super()` constructor. This behavior can be inconvenient if the designated Objective-C initializers aren't marked properly in the library. To disable these compiler checks, use the `disableDesignatedInitializerChecks` property.

Handle Objective-C exceptions

By default, the program crashes if Objective-C exceptions reach the Objective-C to Kotlin interop boundary and get to the Kotlin code.

To propagate Objective-C exceptions to Kotlin, enable wrapping with the `foreignExceptionMode = objc-wrap` property. In this case, Objective-C exceptions are translated into Kotlin exceptions that get the `ForeignException` type.

Help resolve linker errors

Linker errors might occur when a Kotlin library depends on C or Objective-C libraries, for example, using the CocoaPods integration. If dependent libraries aren't installed locally on the machine or configured explicitly in the project build script, the "Framework not found" error occurs.

If you're a library author, you can help your users resolve linker errors with custom messages. To do that, add a `userSetupHint=message` property to your `.def` file or pass the `-Xuser-setup-hint` compiler option to `cinterop`.

Add custom declarations

Sometimes it is required to add custom C declarations to the library before generating bindings (for example, for macros). Instead of creating an additional header file with these declarations, you can include them directly to the end of the `.def` file, after a separating line, containing only the separator sequence `---`:

```
headers = errno.h
---

static inline int getErrno() {
    return errno;
}
```

Note that this part of the `.def` file is treated as part of the header file, so functions with the body should be declared as `static`. The declarations are parsed after including the files from the `headers` list.

Generate bindings using command line

In addition to the definition file, you can specify what to include in bindings by passing the corresponding properties as options in the `cinterop` call.

Here's an example of the command that produces a `png.klib` compiled library:

```
cinterop -def png.def -compiler-option -I/usr/local/include -o png
```

Note that the generated bindings are generally platform-specific, so if you are developing for multiple targets, the bindings need to be regenerated.

- For host libraries that are not included in the sysroot search paths, headers may be needed.
- For a typical UNIX library with a configuration script, the `compilerOpts` will likely contain the output of a configuration script with the `--cflags` option (maybe without exact paths).
- The output of a configuration script with `--libs` can be passed to the `linkerOpts` property.

What's next

- Bindings for C-interoperability
- Interoperability with Swift/Objective-C

C, Objective-C, and Swift library import

Kotlin/Native provides the ability to import C and Objective-C libraries. You can also work around importing pure Swift libraries into your Kotlin/Native projects.

Beta

Stability of C and Objective-C library import

Support for importing C and Objective-C libraries is currently in Beta.

One of the main reasons for the Beta status is that using C and Objective-C libraries might affect the compatibility of your code with different versions of Kotlin, dependencies, and Xcode. This guide lists compatibility problems that happen often in practice, problems that occur only in some cases, and hypothetical potential issues as well.

For simplicity, we divide C and Objective-C libraries, or *native libraries* here into:

- Platform libraries, which Kotlin provides by default to access the "system" native libraries on each platform.

- Third-party libraries, all other native libraries that require additional configuration for Kotlin use.

These two kinds of native libraries have different compatibility specifics.

Platform libraries

Platform libraries are shipped with the Kotlin/Native compiler. So, using different versions of Kotlin in the project results in getting different versions of platform libraries. For Apple targets (like iOS), platform libraries are generated based on the Xcode version supported by the particular compiler version.

Native library APIs shipped with Xcode SDKs change with every Xcode version. Even when such changes are source- and binary-compatible within the native language, they might become breaking for Kotlin due to the interoperability implementation.

As a result, updating the Kotlin version in a project might bring a breaking change in platform libraries. This might matter in two cases:

- There is a source breaking change in platform libraries that affects the compilation of the source code in your project. Usually, it's easy to fix.
- There is a binary breaking change in platform libraries that affects some of your dependencies. There is typically no easy workaround, and you'll need to wait until the library developer fixes this on their side, for example, by updating the Kotlin version.

> Such binary incompatibilities manifest as linkage warnings and runtime exceptions. If you prefer to detect them at compilation time, raise warnings to errors with the `-Xpartial-linkage-loglevel=ERROR` compiler option.

When the JetBrains team updates the Xcode version used to generate the platform libraries, it makes a reasonable effort to avoid breaking changes in platform libraries. Whenever a breaking change might occur, the team conducts impact analysis, and either decides to ignore a particular change (because the affected API is not commonly used) or applies an ad hoc fix.

Another potential reason for breaking changes in platform libraries is changes in the algorithm that translates native APIs to Kotlin. The JetBrains team makes reasonable efforts to avoid breaking changes in such cases as well.

Using new Objective-C classes from platform libraries

The Kotlin compiler doesn't prevent you from using Objective-C classes that are not available with your deployment target.

For example, if your deployment target is iOS 17.0, and you use a class that appeared only in iOS 18.0, the compiler doesn't warn you, and your application might crash during launch on a device with iOS 17.0. Moreover, such a crash happens even when the execution never reaches those usages, so guarding them with version checks is not enough.

For more details, see Strong linking.

Third-party libraries

Apart from system platform libraries, Kotlin/Native allows importing third-party native libraries. For example, you can use CocoaPods integration or set up a cinterops configuration.

Importing libraries with mismatched Xcode version

Importing third-party native libraries can lead to compatibility issues with different Xcode versions.

When processing a native library, the compiler typically uses header files from the locally installed Xcode because almost all native library headers import "standard" headers (for example, `stdint.h`) that come from Xcode.

That's why the Xcode version affects the import of native libraries to Kotlin. This is also one of the reasons why cross-compilation of Apple targets from a non-Mac host is still impossible when using third-party native libraries.

Every Kotlin version is compatible the most with a single Xcode version. This is the recommended version, which is tested the most against the corresponding Kotlin version. Check the compatibility with a particular Xcode version in the compatibility table.

Using newer or older Xcode versions is often possible but might lead to problems, usually affecting the import of third-party native libraries.

Xcode version is newer than recommended

Using an Xcode version newer than recommended might break some Kotlin features. Importing third-party native libraries is most affected by this . It often doesn't work at all with an unsupported version of Xcode.

Xcode version is older than recommended

Typically, Kotlin works well with older Xcode versions. There might be occasional issues that most often result in:

- The Kotlin API referring to a non-existent type as in KT-71694.
- A type from a system library is included in the Kotlin API of the native library. In this case, the project compiles successfully, but a system native type is added to your native library package. For example, you can then see this type unexpectedly in IDE autocompletion.

If your Kotlin library successfully compiles with an older Xcode version, it's safe to publish unless you use types from a third-party library in your Kotlin library API.

Using a transitive third-party native dependency

When a Kotlin library in your project imports a third-party native library as part of its implementation, your project also gets access to that native library. This happens because Kotlin/Native doesn't distinguish between `api` and `implementation` dependency types, so native libraries always end up being `api` dependencies.

Using such a transitive native dependency is prone to more compatibility issues. For example, a change made by the Kotlin library developer might make the Kotlin representation of the native library incompatible, leading to compatibility issues when you update the Kotlin library.

So, instead of relying on a transitive dependency, configure interoperability with the same native library directly. To do that, use another package name for the native library, similar to using custom package name to prevent compatibility issues.

Using native types in library API

If you publish a Kotlin library, be careful with native types in your library API. Such usages are expected to be broken in the future to fix compatibility and other issues, which will affect your library users.

There are cases when using native types in a library API is necessary because it's required for the library's purpose, for example, when a Kotlin library basically provides extensions to the native library. If that's not your case, avoid or limit using native types in the library API.

This recommendation only applies to usages of native types in library APIs and is not related to the application code. It also doesn't apply to library implementations, for example:

```
// Be extra careful! Native types are used in the library API:
public fun createUIView(): UIView
public fun handleThirdPartyNativeType(c: ThirdPartyNativeType)

// Be careful as usual; native types are not used in the library API:
internal fun createUIViewController(): UIViewController
public fun getDate(): String = NSDate().toString()
```

Publishing a library that uses third-party library

If you publish a Kotlin library that uses third-party native libraries, there are a few things you can do to avoid compatibility issues.

Use custom package name

Using custom package names for third-party native libraries might help prevent compatibility issues.

When a native library is imported to Kotlin, it gets a Kotlin package name. If it's not unique, library users might get a clash. For example, if a native library is imported with the same package name elsewhere in the user's project or other dependencies, those two usages will clash.

In such a case, the compilation might fail with the `Linking globals named '...': symbol multiply defined!` error. However, there might be other errors or even a successful compilation.

To use a custom name for third-party native libraries:

- When importing a native library through the CocoaPods integration, use the `packageName` property in the `pod {}` block of your Gradle build script.
- When importing a native library with a `cinterops` configuration, use the `packageName` property in the configuration block.

Check compatibility with older Kotlin versions

When publishing a Kotlin library, the usage of a third-party native library might affect library compatibility with other Kotlin versions, specifically:

- Kotlin Multiplatform libraries don't guarantee forward compatibility (when an older compiler can use a library compiled with a newer compiler).

  In practice, it works in some cases; however, using native libraries might further limit forward compatibility.

- Kotlin Multiplatform libraries provide backward compatibility (when a newer compiler can use libraries produced with an older version).

  Using a native library in the Kotlin library normally shouldn't affect its backward compatibility. But it opens the possibility of more compiler bugs affecting compatibility.

Avoid embedding static libraries

When importing a native library, it's possible to include the associated static library (`.a` file) using the `-staticLibrary` compiler option or the `staticLibraries` property in a `.def` file. In that case, your library users don't need to deal with native dependencies and linker options.

However, it's impossible to configure the usage of the included static library in any way: neither exclude nor replace (substitute) it. So, users won't be able to resolve potential clashes with other Kotlin libraries that include the same static library or adjust its version.

Evolution of native library support

Currently, using C and Objective-C in Kotlin projects can lead to compatibility issues; some of which are listed in this guide. To fix them, some breaking changes might be necessary in the future, which itself contributes to the compatibility problem.

Swift library import

Kotlin/Native does not support direct import of pure Swift libraries. However, there are a couple of options to work around that.

One way is to use manual Objective-C bridging. With this approach, you need to write custom Objective-C wrappers and `.def` files and consume those wrappers through cinterop.

In most cases, however, we recommend using the *reverse import* approach: you define the expected behavior on the Kotlin side, implement the actual functionality on the Swift side, and pass it back to Kotlin.

You can define the expected part in one of the following ways:

- Create an interface. The interface-based approach scales better for multiple functions and testability.
- Use Swift closures. They are great for quick prototypes, but this approach has its limitations — for example, it doesn't hold state.
- Use Swift export. You can implement a Kotlin interface directly in Swift and pass the Swift object back to Kotlin, without Objective-C bridging.

Consider this example of reverse importing the CryptoKit Swift library into a Kotlin project:

1. On the Kotlin side, create an interface to describe what Kotlin expects from Swift:

   Interface

   ```
   // CryptoProvider.kt
   interface CryptoProvider {
       fun hashMD5(input: String): String
   }
   ```

2. On the Kotlin side, pass the platform-specific implementation from `MainViewController` to the `App` composable as a parameter and use it where needed:

   ```
   // App.kt
   @Composable
   fun App(cryptoProvider: CryptoProvider) {
       // Example usage inside your UI
       val hashed = cryptoProvider.hashMD5("Hello, world!")
       androidx.compose.material3.Text("Compose: $hashed")
   }
   ```

```
// MainViewController.kt
fun MainViewController(cryptoProvider: CryptoProvider) = ComposeUIViewController {
    App(cryptoProvider)
}
```

3. On the Swift side, implement the MD5-hashing functionality using a pure Swift library, CryptoKit:

```
// iosApp/ContentView.swift
import CryptoKit

class IosCryptoProvider: CryptoProvider {
    func hashMD5(input: String) -> String {
        guard let data = input.data(using: .utf8) else { return "failed" }
        return Insecure.MD5.hash(data: data).description
    }
}
```

4. Pass the Swift implementation to the Kotlin component:

```
// iosApp/ContentView.swift
struct ComposeView: UIViewControllerRepresentable {
    func makeUIViewController(context: Context) -> UIViewController {
        // Inject the Swift implementation into the Kotlin UI entry point
        MainViewControllerKt.MainViewController(cryptoProvider: IosCryptoProvider())
    }

    func updateUIViewController(_ uiViewController: UIViewController, context: Context) {}
}
```

1. On the Kotlin side, declare a function parameter and use it where needed:

Swift closures

```
// App.kt
@Composable
fun App(md5Hasher: (String) -> String) {
    // Example usage inside your UI
    val hashed = md5Hasher("Hello, world!")
    androidx.compose.material3.Text("Compose: $hashed")
}
```

```
// MainViewController.kt
fun MainViewController(md5Hasher: (String) -> String) = ComposeUIViewController {
    App(md5Hasher)
}
```

2. On the Swift side, build the MD5 hasher with the CryptoKit library and pass it as a closure:

```swift
// iosApp/ContentView.swift
import CryptoKit
import SwiftUI

struct ComposeView: UIViewControllerRepresentable {
    func makeUIViewController(context: Context) -> UIViewController {
        MainViewControllerKt.MainViewController(md5Hasher: { input in
            guard let data = input.data(using: .utf8) else { return "failed" }
            return Insecure.MD5.hash(data: data).description
        })
    }

    func updateUIViewController(_ uiViewController: UIViewController, context: Context) {}
}
```

1. On the Kotlin side, declare an interface, a function that accepts it, and an `open` base class that the Swift implementation can inherit from:

Swift export

```kotlin
// CryptoProvider.kt
interface CryptoProvider {
    fun hashMD5(input: String): String
}

fun processHash(provider: CryptoProvider, input: String): String = provider.hashMD5(input)

open class SwiftBase
```

2. On the Swift side, inherit from the exported `SwiftBase` class, implement the interface using the pure Swift CryptoKit library, and pass the object back to Kotlin:

```swift
// iosApp/ContentView.swift
import CryptoKit

final class IosCryptoProvider: SwiftBase, CryptoProvider {
    func hashMD5(input: String) -> String {
        guard let data = input.data(using: .utf8) else { return "failed" }
        return Insecure.MD5.hash(data: data).description
    }
}

let provider = IosCryptoProvider()

// Calls the Kotlin function, which calls hashMD5() back in Swift
print(processHash(provider: provider, input: "Hello, world!"))
```

When Kotlin receives the Swift object, it treats it like an implementation of a regular Kotlin interface, calling the Swift code directly.

In a more complex project, it's more convenient to use dependency injection to pass the Swift implementation back to Kotlin. For more information, see Dependency injection framework or check out the Koin framework documentation.

Kotlin AI skills

Kotlin AI skills are reusable instructions that help AI agents perform Kotlin-specific tasks more reliably.

A skill gives an AI agent the context it needs before it starts executing a task. For example, a skill can point to a relevant API, define prerequisites, or provide step-by-step workflow guidance.

Kotlin AI skills help agents produce more accurate results and reduce the time you spend explaining the task yourself. For teams, skills also provide a shared framework for common tasks, so everyone gets consistent results.

Explore Kotlin AI skills

Kotlin AI skills follow the Agent Skills standard, so you can use them with compatible AI agents, for example, Junie, Claude Code, OpenAI Codex, Google Gemini, and GitHub Copilot.

Supported workflows

You can use Kotlin AI skills for different Kotlin-specific scenarios. The following examples show some of the tasks that AI skills can help you with.

Convert Java source files to Kotlin

Use the kotlin-tooling-java-to-kotlin skill when you want to convert your Java source files to idiomatic Kotlin while preserving behavior and applying Kotlin-specific conventions.

Learn more about this scenario in Convert Java files to Kotlin.

Migrate multiplatform projects with Android apps to use AGP 9

Use the kotlin-tooling-agp9-migration skill when your Kotlin Multiplatform project needs to migrate to AGP 9, and you want an AI agent to apply the required project and Gradle configuration changes.

Learn more about this scenario in Update multiplatform projects with Android apps to use AGP 9.

Migrate multiplatform projects from CocoaPods to SwiftPM dependencies

Use the kotlin-tooling-cocoapods-spm-migration when your Kotlin Multiplatform project uses CocoaPods for iOS integration, and you want an AI agent to move the setup to SwiftPM.

Learn more about this scenario in Migrate multiplatform projects from CocoaPods to SwiftPM dependencies.

Get support

If you have questions or encounter problems, ask for help in Slack: get an invite and share your experience in the `#ai` channel.

Kotlin for AI-powered app development

Kotlin provides a modern and pragmatic foundation for building AI-powered applications.
It can be used across platforms, integrates well with established AI frameworks, and supports common AI development patterns.

Koog

Koog is an open-source framework from JetBrains for building AI agents, from simple to complex. It provides multiplatform support, Spring Boot and Ktor integrations, an idiomatic DSL, and production-ready features out of the box.

Create a simple agent in a few lines

```
fun main() {
    runBlocking {
        val agent = AIAgent(
            // Use Anthropic, Google, OpenRouter, or any other provider
            executor = simpleOpenAIExecutor(System.getenv("OPENAI_API_KEY")),
            systemPrompt = "You are a helpful assistant. Answer user questions concisely.",
            llmModel = OpenAIModels.Chat.GPT4o
        )

        val result = agent.run("Hello! How can you help me?")
        println(result)
    }
}
```

Get started with Koog

Key features

- **Support for multiplatform development**. Multiplatform support enables agentic application development for the JVM, JavaScript, WebAssembly, Android, and iOS.
- **Reliability and fault-tolerance**. With built-in retries, Koog lets developers handle failures such as timeouts or tool errors. Agent persistence makes it possible to restore full agent state machines instead of just chat messages.
- **Built-in history compression techniques for long contexts**. Koog comes with advanced strategies to compress and manage long-running conversations without extra setup.
- **Enterprise-ready integrations**. Koog integrates with popular JVM frameworks like Spring Boot and Ktor.
- **Observability with OpenTelemetry exporters**. Koog offers ready-to-use integration with popular observability providers such as W&B Weave and Langfuse for monitoring and debugging AI applications.
- **LLM switching and seamless history adaptation**. Koog allows switching to a different LLM with a new set of tools at any point, without losing the existing conversation history. It also enables rerouting between multiple LLM providers, including OpenAI, Anthropic, Google, and others. You can run agents locally with local models through Koog's integration with Ollama.
- **Integration with JVM and Kotlin applications**. Koog provides an idiomatic, type-safe DSL specifically for the JVM and Kotlin developers.
- **Model Context Protocol (MCP) integration**. Koog enables the use of MCP tools in agents.
- **Knowledge retrieval and memory**. With embeddings, ranked document storage, and shared agent memory, Koog itself actively retains knowledge across conversations.
- **Streaming capabilities**. Koog lets developers process responses in real-time with streaming support and parallel tool calls.

Where to start

- Explore Koog capabilities in the Overview.
- Build your first Koog agent with the Quickstart guide.
- See the latest updates in the Koog release notes.
- Learn from the examples.

Model Context Protocol (MCP) Kotlin SDK

The MCP Kotlin SDK is a Kotlin Multiplatform implementation of the Model Context Protocol. The SDK lets developers build AI-powered applications in Kotlin and integrate with LLM surfaces across the JVM, WebAssembly, and iOS.

With the MCP Kotlin SDK, you can:

- Provide context to LLMs in a structured and standardized way by separating context handling from LLM interaction.
- Build MCP clients that consume resources from existing servers.
- Create MCP servers that expose prompts, tools, and resources to LLMs.
- Use standard communication transports such as stdio, SSE, and WebSocket.
- Handle all MCP protocol messages and lifecycle events.

Explore other AI-powered application scenarios

Thanks to seamless Java interoperability and Kotlin Multiplatform, you can combine Kotlin with established AI SDKs and frameworks, build backend and desktop/mobile UIs, and adopt patterns like RAG and agent-based workflows.

> You can explore and run examples from the Kotlin-AI-Examples repository. Each project is self-contained. You can use each project as a reference or template for building Kotlin-based AI applications.

Connect to major model providers

Use Kotlin to connect to major model providers such as OpenAI, Anthropic, Google, and others:

- OpenAI — official Java SDK for the OpenAI API. It covers responses and chats, images, and audio.
- Anthropic (Claude) — official Java SDK for the Claude Messages API. It includes modules for Vertex AI and Bedrock integrations.
- Google AI (Gemini / Vertex AI) — official Java SDK with a single client that switches between Gemini API and Vertex AI.
- Azure OpenAI — official Java client for the Azure OpenAI Service. It supports chat completions and embeddings.
- AWS Bedrock — official SDKs to invoke foundation models. It includes the Kotlin SDK and Java SDK for Bedrock and Bedrock Runtime.

Create RAG pipelines and agent-based apps

- Spring AI — multi-provider abstraction for prompts, chats, embeddings, tools and function calling, and vector stores.
- LangChain4j — JVM toolkit with Kotlin extensions for prompts, tools, retrieval-augmented generation (RAG) pipelines, and agents.

What's next

- Complete the Create a Kotlin app that answers questions with Spring AI tutorial to learn more about using Spring AI with Kotlin in IntelliJ IDEA
- Join the Kotlin community to connect with other developers building AI applications with Kotlin
- Learn about Kotlin AI skills

Create a Kotlin app that answers questions with Spring AI — tutorial

In this tutorial, you'll learn how to build a Kotlin app that connects to an LLM via Spring AI, stores documents in a vector database, and answers questions using context from those documents.

You will use the following tools during this tutorial:

- Spring Boot as the base to configure and run the web application.
- Spring AI to interact with the LLM and perform context-based retrieval.
- IntelliJ IDEA to generate the project and implement the application logic.
- Qdrant as the vector database for similarity search.
- Docker to run Qdrant locally.
- OpenAI as the LLM provider.

Before you start

1. Download and install the latest version of IntelliJ IDEA and use an Ultimate subscription.

   > If you use IntelliJ IDEA without an Ultimate subscription or you use another IDE, you can generate a Spring Boot project using a web-based project generator.

2. Create an OpenAI API key on the OpenAI platform to access the API.
3. Install Docker to run the Qdrant vector database locally.
4. After installing Docker, open your terminal and run the following command to start the container:

```
docker run -p 6333:6333 -p 6334:6334 qdrant/qdrant
```

Create the project

> You can use the Spring Boot web-based project generator as an alternative to generate your project.

Create a new Spring Boot project in IntelliJ IDEA with an Ultimate subscription:

1. In IntelliJ IDEA, select **File** | **New** | **Project**.
2. In the panel on the left, select **New Project** | **Spring Boot**.
3. Specify the following fields and options in the **New Project** window:
   - **Name**: springAIDemo
   - **Language**: Kotlin
   - **Type**: Gradle - Kotlin

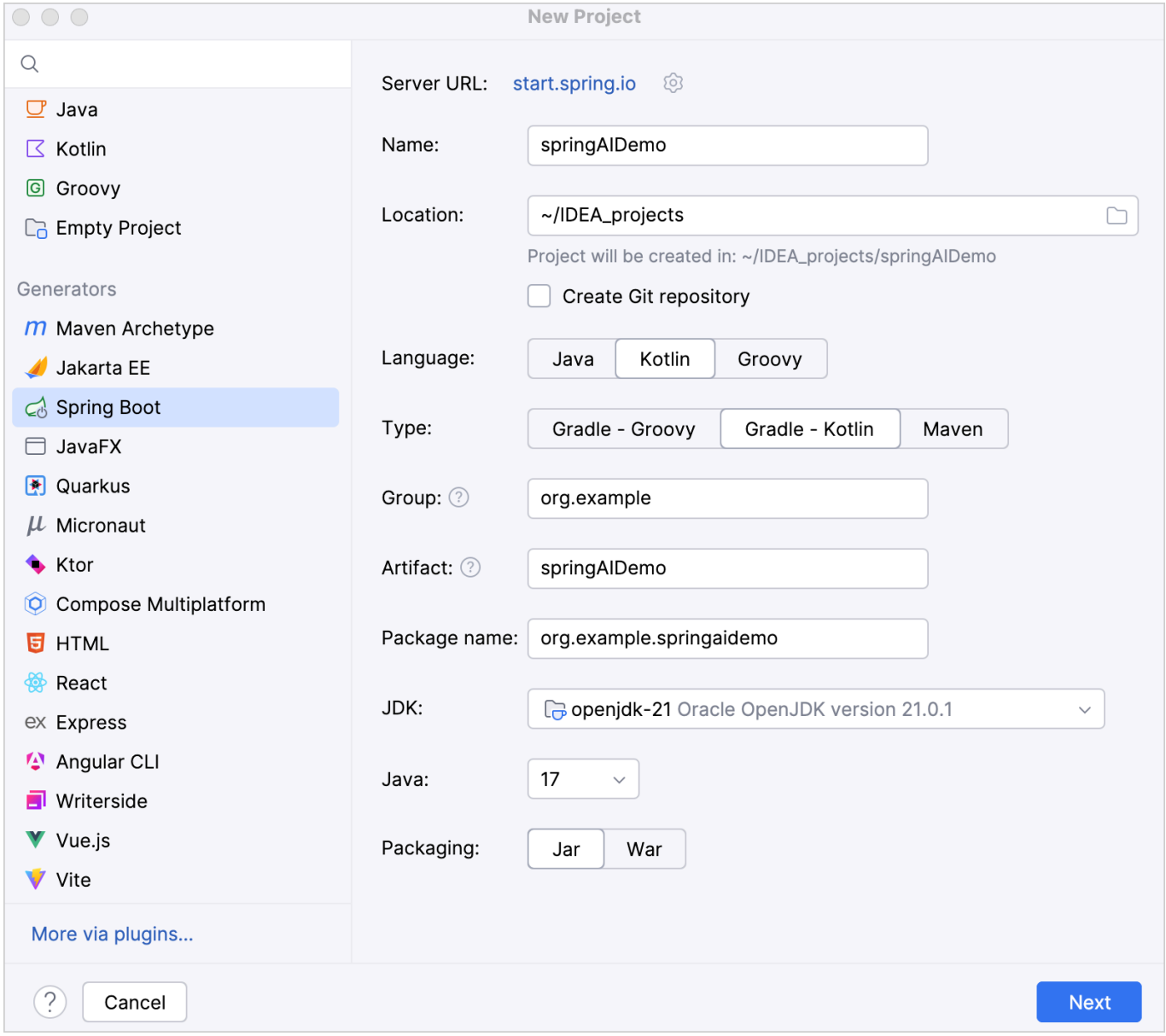
     > This option specifies the build system and the DSL.

   - **Package name**: org.example.springaidemo
   - **JDK**: Java JDK

     > This tutorial uses **Oracle OpenJDK version 21.0.1**. If you don't have a JDK installed, you can download it from the dropdown list.

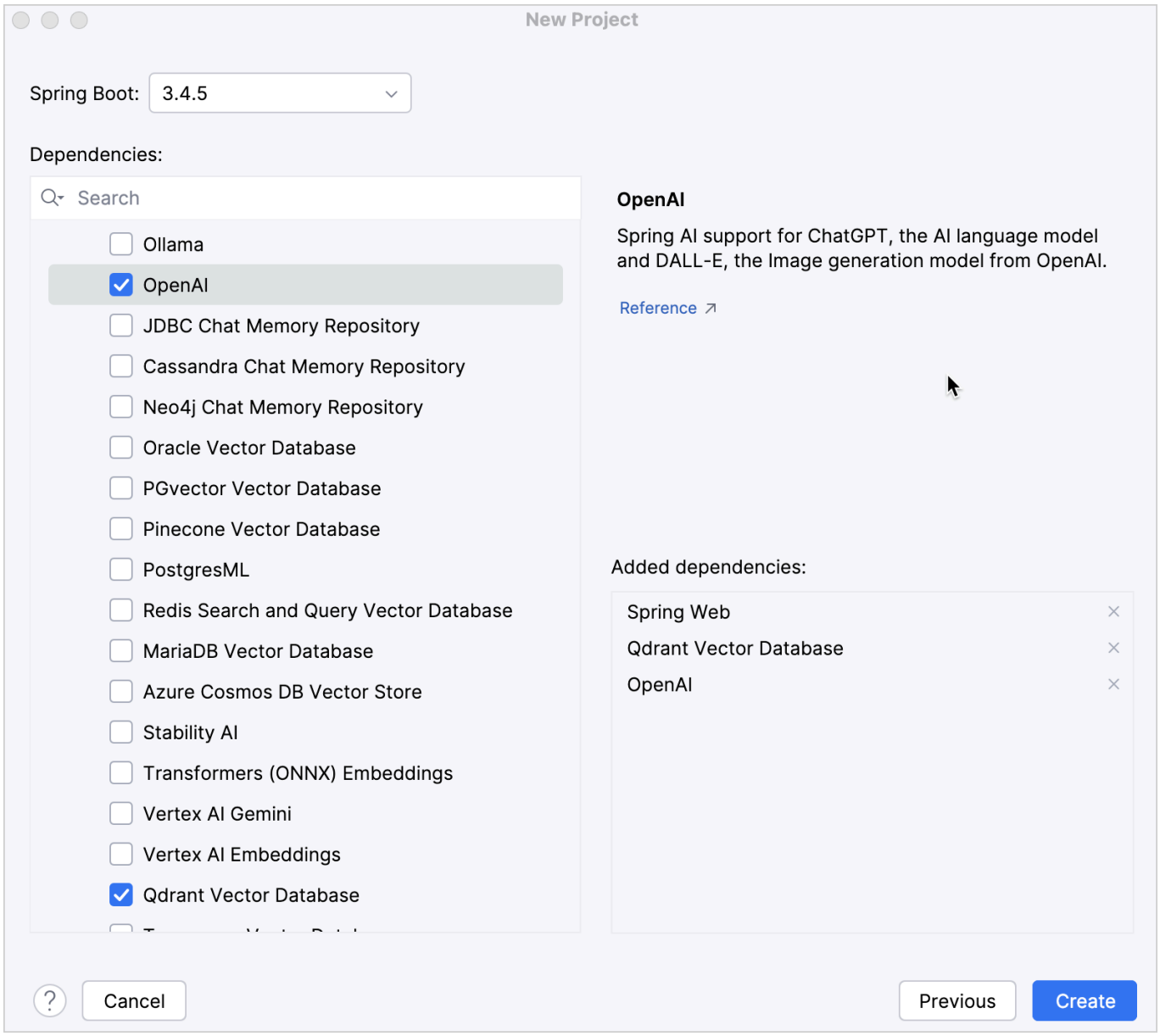
   - **Java**: 17

     > If you don't have Java 17 installed, you can download it from the JDK dropdown list.



Create Spring Boot project

4. Make sure that you have specified all the fields and click **Next**.
5. Select the latest stable Spring Boot version in the **Spring Boot** field.
6. Select the following dependencies required for this tutorial:
   - **Web | Spring Web**
   - **AI | OpenAI**
   - **SQL | Qdrant Vector Database**

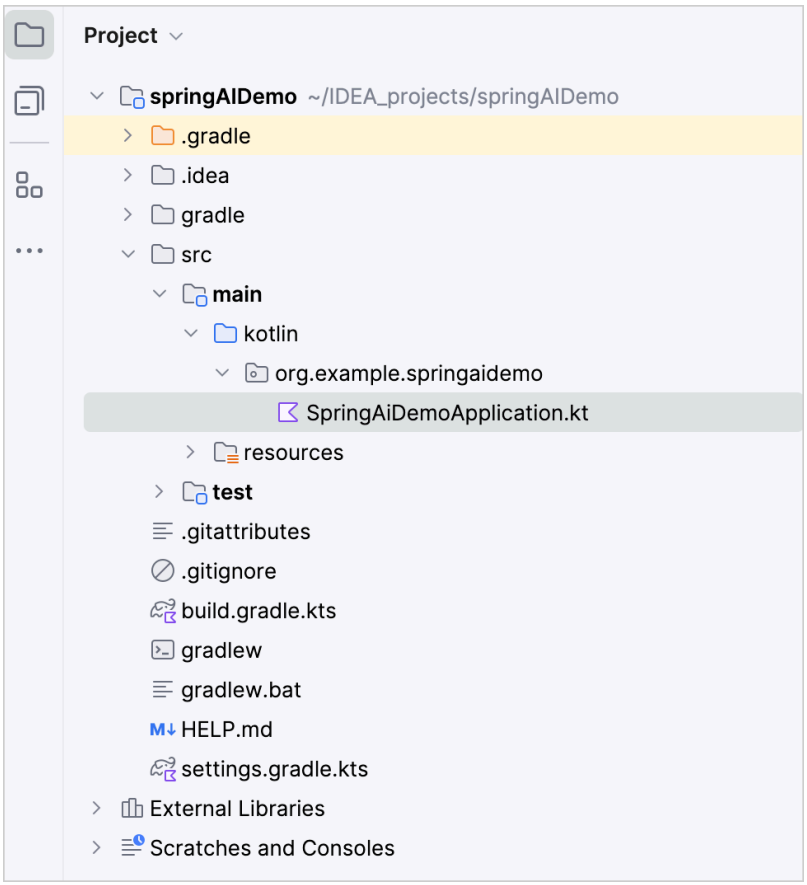


Set up Spring Boot project

7. Click **Create** to generate and set up the project.

> The IDE will generate and open a new project. It may take some time to download and import the project dependencies.

After this, you can see the following structure in the **Project view**:



Spring Boot project view

The generated Gradle project corresponds to the Maven's standard directory layout:

- There are packages and classes under the `main/kotlin` folder that belong to the application.
- The entry point to the application is the `main()` method of the `SpringAiDemoApplication.kt` file.

Update the project configuration

1. Update your `build.gradle.kts` Gradle build file with the following:

```
plugins {
    kotlin("jvm") version "2.4.20"
    kotlin("plugin.spring") version "2.4.20"
    // Rest of the plugins
}
```

2. Set `springAiVersion` to `2.0.0`:

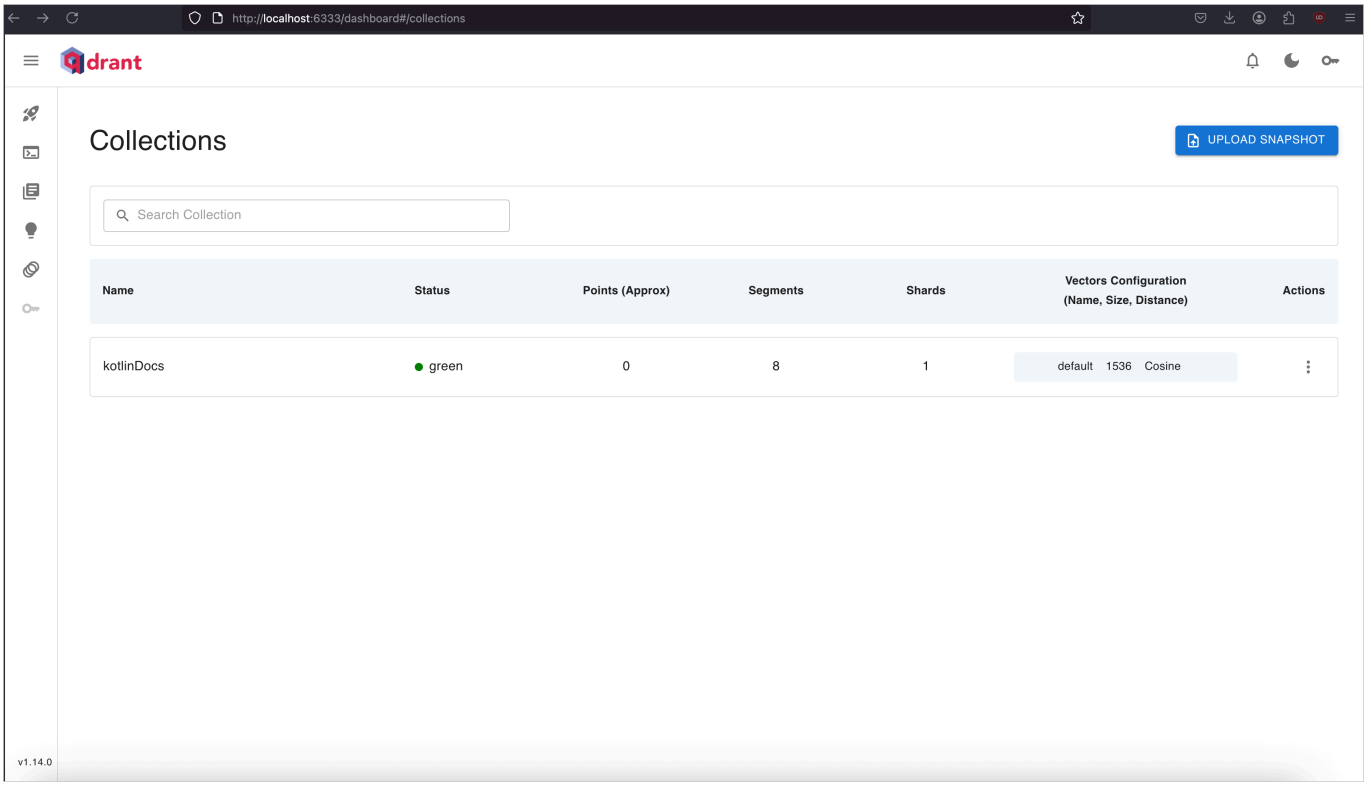
```
extra["springAiVersion"] = "2.0.0"
```

3. Click the **Sync Gradle Changes** button to synchronize the Gradle files.
4. Update your `src/main/resources/application.properties` file with the following:

```
# OpenAI
spring.ai.openai.api-key=YOUR_OPENAI_API_KEY
spring.ai.openai.chat.model=gpt-4o-mini
spring.ai.openai.embedding.model=text-embedding-ada-002
# Qdrant
spring.ai.vectorstore.qdrant.host=localhost
spring.ai.vectorstore.qdrant.port=6334
spring.ai.vectorstore.qdrant.collection-name=kotlinDocs
spring.ai.vectorstore.qdrant.initialize-schema=true
```

Set your OpenAI API key to the `spring.ai.openai.api-key` property.

5. Run the `SpringAiDemoApplication.kt` file to start the Spring Boot application. Once it's running, open the Qdrant collections page in your browser to see the result:



Qdrant collections

Create a controller to load and search documents

Create a Spring `@RestController` to search documents and store them in the Qdrant collection:

1. In the `src/main/kotlin/org/example/springaidemo` directory, create a new file named `KotlinSTDController.kt`, and add the following code:

```kotlin
package org.example.springaidemo

// Imports the required Spring and utility classes
import org.slf4j.LoggerFactory
import org.springframework.ai.document.Document
import org.springframework.ai.vectorstore.SearchRequest
import org.springframework.ai.vectorstore.VectorStore
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PostMapping
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.RestController
import org.springframework.web.client.RestTemplate
import kotlin.uuid.ExperimentalUuidApi
import kotlin.uuid.Uuid

@RestController
@RequestMapping("/kotlin")
class KotlinSTDController(
    private val restTemplate: RestTemplate,
    private val vectorStore: VectorStore,
) {
    private val logger = LoggerFactory.getLogger(this::class.java)

    @OptIn(ExperimentalUuidApi::class)
    @PostMapping("/load-docs")
    fun load() {
        // Loads a list of documents from the Kotlin documentation
        val kotlinStdTopics = listOf(
            "collections-overview", "constructing-collections", "iterators", "ranges", "sequences",
            "collection-operations", "collection-transformations", "collection-filtering",
"collection-plus-minus",
            "collection-grouping", "collection-parts", "collection-elements", "collection-
ordering",
            "collection-aggregate", "collection-write", "list-operations", "set-operations",
            "map-operations", "read-standard-input", "opt-in-requirements", "scope-functions",
"time-measurement",
        )
        // Base URL for the documents
        val url = "https://raw.githubusercontent.com/JetBrains/kotlin-web-
site/refs/heads/master/docs/topics/"
        // Retrieves each document from the URL and adds it to the vector store
        kotlinStdTopics.forEach { topic ->
            val data = restTemplate.getForObject("$url$topic.md", String::class.java)
            data?.let { it ->
                val doc = Document.builder()
                    // Builds a document with a random UUID
                    .id(Uuid.random().toString())
                    .text(it)
                    .metadata("topic", topic)
                    .build()
                vectorStore.add(listOf(doc))
                logger.info("Document $topic loaded.")
            } ?: logger.warn("Failed to load document for topic: $topic")
        }
    }

    @GetMapping("docs")
    fun query(
        @RequestParam query: String = "operations, filtering, and transformations",
        @RequestParam topK: Int = 2
    ): List<Document>? {
        val searchRequest = SearchRequest.builder()
```

```
            .query(query)
            .topK(topK)
            .build()
        val results = vectorStore.similaritySearch(searchRequest)
        logger.info("Found ${results?.size ?: 0} documents for query: '$query'")
        return results
    }
}
```

2. Update the `SpringAiDemoApplication.kt` file to declare a `RestTemplate` bean:

```
package org.example.springaidemo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.context.annotation.Bean
import org.springframework.web.client.RestTemplate

@SpringBootApplication
class SpringAiDemoApplication {
    @Bean
    fun restTemplate(): RestTemplate = RestTemplate()
}

fun main(args: Array<String>) {
    runApplication<SpringAiDemoApplication>(*args)
}
```

3. Run the application.

4. In the terminal, send a POST request to the `/kotlin/load-docs` endpoint to load the documents:

```
curl -X POST http://localhost:8080/kotlin/load-docs
```

5. Once the documents are loaded, you can search for them with a GET request:

```
curl -X GET http://localhost:8080/kotlin/docs
```



GET request results

> You can also view the results on the Qdrant collections page.

Implement an AI chat endpoint

Once the documents are loaded, the final step is to add an endpoint that answers questions using the documents in Qdrant through Spring AI's Retrieval-Augmented Generation (RAG) support:

1. Open the `KotlinSTDController.kt` file, and import the following classes:

```
import org.springframework.ai.chat.client.ChatClient
import org.springframework.ai.chat.client.advisor.SimpleLoggerAdvisor
import org.springframework.ai.chat.client.advisor.vectorstore.QuestionAnswerAdvisor
import org.springframework.ai.chat.prompt.Prompt
import org.springframework.ai.chat.prompt.PromptTemplate
import org.springframework.web.bind.annotation.RequestBody
```

2. Define a `ChatRequest` data class:

```
// Represents the request payload for chat queries
data class ChatRequest(val query: String, val topK: Int = 3)
```

3. Add `ChatClient.Builder` to the controller's constructor parameters:

```
class KotlinSTDController(
    // Provides the builder for creating the ChatClient
    private val chatClientBuilder: ChatClient.Builder,

    private val restTemplate: RestTemplate,
    private val vectorStore: VectorStore,
)
```

4. Inside the controller class, create a `ChatClient` instance:

```
// Builds the chat client with a simple logging advisor
private val chatClient = chatClientBuilder.defaultAdvisors(SimpleLoggerAdvisor()).build()
```

5. At the bottom of your `KotlinSTDController.kt` file, add a new `chatAsk()` endpoint, with the following logic:

```
@PostMapping("/chat/ask")
fun chatAsk(@RequestBody request: ChatRequest): String? {
    // Defines the prompt template with placeholders
    val promptTemplate = PromptTemplate(
        """
        {query}.
        Please provide a concise answer based on the "Kotlin standard library" documentation.
    """.trimIndent()
    )

    // Creates the prompt by substituting placeholders with actual values
    val prompt: Prompt =
        promptTemplate.create(mapOf("query" to request.query))

    // Configures the retrieval advisor to augment the query with relevant documents
    val retrievalAdvisor = QuestionAnswerAdvisor.builder(vectorStore)
        .searchRequest(
            SearchRequest.builder()
                .similarityThreshold(0.7)
                .topK(request.topK)
                .build()
        )
        .promptTemplate(promptTemplate)
        .build()

    // Sends the prompt to the LLM with the retrieval advisor and retrieves the generated content
    val response = chatClient.prompt(prompt)
        .advisors(retrievalAdvisor)
        .call()
        .content()
    logger.info("Chat response generated for query: '${request.query}'")
    return response
}
```

6. Run the application.
7. In the terminal, send a POST request to the new endpoint to see the results:

```
curl -X POST "http://localhost:8080/kotlin/chat/ask" \
     -H "Content-Type: application/json" \
     -d '{"query": "What are the performance implications of using lazy sequences in Kotlin for large datasets?", "topK": 3}'
```



OpenAI answer to chat request

Congratulations! You now have a Kotlin app that connects to OpenAI and answers questions using context retrieved from documentation stored in Qdrant. Try experimenting with different queries or importing other documents to explore more possibilities.

You can view the completed project in the Spring AI demo GitHub repository.

What's next

- Explore other Spring AI examples in Kotlin AI Examples
- Create a task manager application with Spring Boot and Claude

Kotlin for data analysis

Exploring and analyzing data is something you may not do every day, but it's a crucial skill you need as a software developer.

Let's think about software development duties where data analysis is key: analyzing what's actually inside collections when debugging, digging into memory dumps or databases, or receiving JSON files with large amounts of data when working with REST APIs, to mention some.

With Kotlin's Exploratory Data Analysis (EDA) tools, such as Kotlin DataFrame and Kandy, you have at your disposal a rich set of capabilities to enhance your analytics skills and support you across different scenarios:

- **Load, transform, and visualize data in various formats:** with our Kotlin EDA tools, you can perform tasks like filtering, sorting, and aggregating data. Our tools can seamlessly read data right in the IDE from different data sources, such as CSV, JSON, SQL Databases, or Parquet files. See all supported formats in the DataFrame documentation.

  Kandy, our plotting tool, allows you to create a wide range of charts to visualize and gain insights from the dataset.

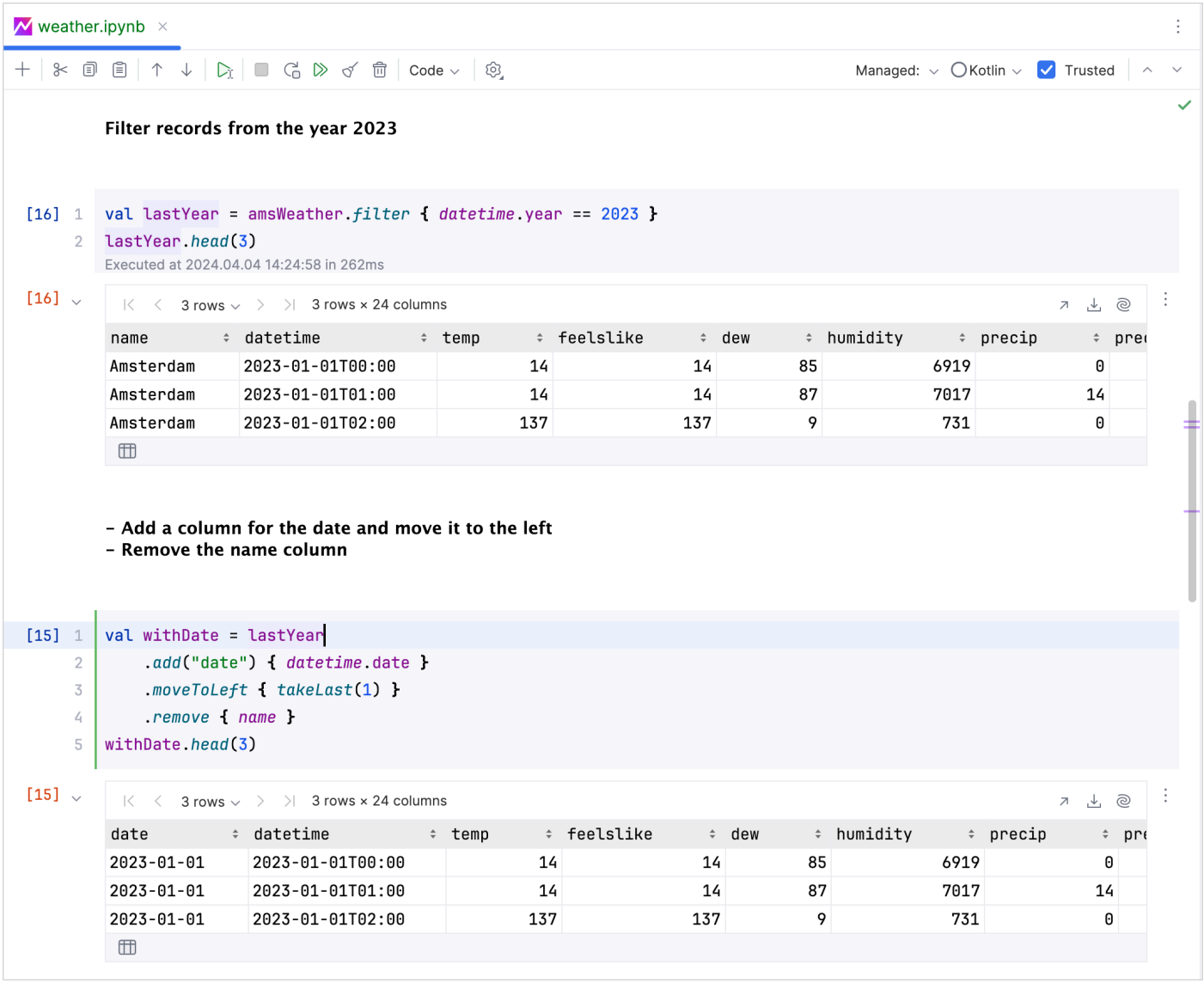
- **Efficiently analyze data stored in relational databases:** Kotlin DataFrame seamlessly integrates with databases and provides capabilities similar to SQL queries. You can retrieve, manipulate, and visualize data directly from various databases.

- **Fetch and analyze real-time and dynamic datasets from web APIs:** the EDA tools' flexibility allows integration with external APIs via protocols like OpenAPI. This feature helps you fetch data from web APIs, to then clean and transform the data to your needs.

Kotlin DataFrame

The Kotlin DataFrame library lets you manipulate structured data in your Kotlin projects. From data creation and cleaning to in-depth analysis and feature engineering, this library has you covered.

With the Kotlin DataFrame library, you can work with different file formats, including CSV, JSON, XLS, and XLSX. This library also facilitates the data retrieval process with its ability to connect with SQL databases or APIs. See all supported formats in the DataFrame documentation.



Kotlin DataFrame

Kandy

Kandy is an open-source Kotlin library that provides a powerful and flexible DSL for plotting charts of various types. This library is a simple, idiomatic, readable, and type-safe tool to visualize data. You can also easily combine Kandy with Kotlin Dataframe to complete different data-related tasks.



Kandy

What's next

- Retrieve and transform data using the Kotlin DataFrame library
- Visualize data using the Kandy library
- Learn more about Kotlin and Java libraries for data analysis

Retrieve data from files

Kotlin DataFrame library enables you to work with both non-structured and structured data. For data transformations, you can use such methods as `.add()`, `.split()`, `.convert()`, and `.parse()`. Additionally, this toolset enables the retrieval and manipulation of data from various structured file formats, including CSV, JSON, XLS, Parquet, and Apache Arrow. See all supported formats in the DataFrame documentation.

In this guide, you can learn how to retrieve, refine, and handle data through multiple examples.

Before you start

Starting with IntelliJ IDEA 2026.2, Kotlin Notebook will no longer be bundled with the IDE or officially supported by JetBrains. The source code will remain available on GitHub.

Learn more in the blog post.

To follow this tutorial:

1. Select **File** | **New** | **Kotlin Notebook**.
2. Import Kotlin DataFrame:

   ```
   %use dataframe
   ```

   Run the code cell with the `%use dataframe` line before any other code cells to make sure the DataFrame library and its APIs are available in the notebook.

To follow the tutorial, you can also use DataFrame as a Gradle or Maven dependency.

Retrieve data

To retrieve data from a file, use the `DataFrame.read()` function:

```
val movies = DataFrame.read("movies.csv")
```

The `DataFrame.read()` function detects the input format based on the file extension and content.

You can also pass additional arguments to control how the DataFrame library reads the input data. For example, the following code specifies a custom delimiter (`;`) for a CSV file:

```
val movies = DataFrame.read("movies.csv", delimiter = ';')
```

For a comprehensive overview of additional file formats and a variety of read functions, see the Kotlin DataFrame library documentation.

Display data

Once you have the data, you can display it. The easiest way is to store your data in a variable and then return it:

```
val jsonDf = DataFrame.read("jsonFile.json")
jsonDf
```

This code displays the data from your file as an interactive table:

```
1 val jsonDf = DataFrame.read("jsonFile.json")
2 jsonDf
```

✓ [8] 254ms

4 rows 4 rows × 6 cols

id	first_name	last_name	email	gender	ip_address
1	Jeanette	Penddreth	jpenddreth0@census.gov	Female	26.58.193.2
2	Giavani	Frediani	gfrediani1@senate.gov	Male	229.179.4.212
3	Noell	Bea	nbea2@imageshack.us	Female	180.66.162.255
4	Willard	Valek	wvalek3@vk.com	Male	67.76.188.26

Display data

You can use this view to inspect values, check column names, and easily understand the state of your dataset.

Inspect data structure

To gain insights into the structure or schema of your data, use the `.schema()` function on your DataFrame variable.

For example, run `jsonDf.schema()` to list the type of each column in your JSON dataset:

```
jsonDf.schema()
```

✓ [9] 37ms

```
id: Int
first_name: String
last_name: String
email: String
gender: String
ip_address: String
```

Schema example

You can also use the autocompletion feature. It allows you to quickly access and manipulate the properties of your DataFrame. After loading your data, simply type the DataFrame variable followed by a period (`.`) to see a list of available columns and their types.

```
jsonDf.
```

```
first_name for ColumnsScope<Line_17_jupyter._DataFrameTy…  DataColumn<String>
id for ColumnsScope<Line_17_jupyter._DataFrameType> in Line…  DataColumn<Int>
gender for ColumnsScope<Line_17_jupyter._DataFrameType> …  DataColumn<String>
email for ColumnsScope<Line_17_jupyter._DataFrameType> i…  DataColumn<String>
last_name for ColumnsScope<Line_17_jupyter._DataFrameTyp…  DataColumn<String>
```

Available properties

Refine data

Kotlin DataFrame provides various operations for refining your dataset. For example, grouping, filtering, updating, or adding new columns. These functions are essential for data analysis, allowing you to organize, clean, and transform your data effectively.

For example, let's look at the `movies.csv` dataset. It stores movie titles and release years in the same cell. The goal is to refine this dataset for easier analysis:

1. **Load the data**

   Load the file into a `DataFrame` using the `.read()` function:

   ```
   val movies = DataFrame.read("movies.csv")
   ```

2. **Add a column**

   To extract the release year from the `title` column, add a new `year` column:

   ```
   val moviesWithYear = movies
       .add("year") {
           "\\d{4}".toRegex()
               .findAll(title)
               .lastOrNull()
               ?.value
               ?.toInt()
               ?: -1
       }

   moviesWithYear
   ```

3. **Update values**

   To remove the release year from the movie title, update the `title` column:

   ```
   val moviesTitle = moviesWithYear
       .update("title") {
           "\\s*\\(\\d{4}\\)\\s*$".toRegex().replace(title, "")
   }

   moviesTitle
   ```

   The code keeps the movie titles in one column and moves the release years into another column.

4. **Filter rows**

   To focus on specific data, use the `.filter()` function. For example, to keep only the movies released after 1986, run:

   ```
   val newMovies = moviesTitle.filter {
       year ≥ 1996
   }

   newMovies
   ```

5. **Remove column**

   To remove a column that you do not need, use the `.remove()` function:

```
val refinedMovies = newMovies.remove {
    movieID
}

refinedMovies
```

For comparison, here is the dataset before refinement:

movieId	title	genres
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance
5	Father of the Bride Part II (1995)	Comedy
6	Heat (1995)	Action|Crime|Thriller
7	Sabrina (1995)	Comedy|Romance
8	Tom and Huck (1995)	Adventure|Children
9	Sudden Death (1995)	Action
10	GoldenEye (1995)	Action|Adventure|Thriller

Original dataset

The dataset after refinement:

title	genres	year
Eye for an Eye	Drama|Thriller	1996
Don't Be a Menace to South Central While Drink…	Comedy|Crime	1996
Two if by Sea	Comedy|Romance	1996
Bio-Dome	Comedy	1996
Lawnmower Man 2: Beyond Cyberspace	Action|Sci-Fi|Thriller	1996
From Dusk Till Dawn	Action|Comedy|Horror|Thriller	1996
Bed of Roses	Drama|Romance	1996
Big Bully	Comedy|Drama	1996
Juror, The	Drama|Thriller	1996
White Squall	Action|Adventure|Drama	1996

Data refinement result

For additional use cases and detailed examples, see Examples of Kotlin Dataframe.

Export data

After refining the data, you can easily export it.

You can utilize a variety of `.write()` functions for this purpose. It supports saving in multiple formats, including CSV, JSON, XLS, XLSX, Apache Arrow, and even HTML tables. See all supported formats in the DataFrame documentation. This can be particularly useful for sharing your findings, creating reports, or making your data available for further analysis.

For example, let's save the result as:

- JSON file using the `.writeJson()` function:

```
refinedMovies.writeJson("movies.json")
```

- CSV file using the `.writeCsv()` function:

```
refinedMovies.writeCsv("movies.csv")
```

- Apache Arrow files using the `.writeArrorIPC()` and `.writeArrorFeather()` functions:

```
refinedMovies.writeArrowIPC("movies.arrow")
refinedMovies.writeArrowFeather("movies.feather")
```

You can also open a standalone HTML table in your browser with the `.toStandaloneHTML()` function:

```
refinedMoviesDf
    .toStandaloneHTML(DisplayConfiguration(rowsLimit = null))
    .openInBrowser()
```

What's next

- Explore data visualization using the Kandy library
- Find additional information about data visualization in Data visualization with Kandy
- For an extensive overview of tools and resources available for data science and analysis in Kotlin, see Kotlin and Java libraries for data analysis

Retrieve data from web sources and APIs

With Kotlin DataFrame library, you can access and manipulate data from various web sources and APIs. It also assists in reshaping this data for comprehensive analysis and visualization.

Explore DataFrame examples on GitHub.

Before you start

Starting with IntelliJ IDEA 2026.2, Kotlin Notebook will no longer be bundled with the IDE or officially supported by JetBrains. The source code will remain available on GitHub.

Learn more in the blog post.

Create a new Kotlin Notebook:

1. Select **File** | **New** | **Kotlin Notebook**.
2. Import the Kotlin DataFrame library by running the following command:

```
%use dataframe
```

To follow the tutorial, you can also use DataFrame as a Gradle or Maven dependency.

Fetch data from an API

Fetching data from APIs using the Kotlin DataFrame library is achieved through the `.read()` function, which is similar to retrieving data from files, such as CSV or JSON. However, when working with web-based sources, you might require additional formatting to transform the raw API data into a structured format.

Let's look at an example of fetching data from the YouTube Data API:

1. Securely add your API key in a new code cell, which is necessary for authenticating requests to the YouTube Data API. You can obtain your API key from the credentials tab:

```
val apiKey = "YOUR-API_KEY"
```

2. Create a load function that takes a path as a string and uses the DataFrame's `.read()` function to fetch data from the YouTube Data API:

```
fun load(path: String): AnyRow =
DataRow.read("https://www.googleapis.com/youtube/v3/$path&key=$apiKey")
```

3. Organize the fetched data into rows and handle the YouTube API's pagination through the `nextPageToken`. This ensures you gather data across multiple pages:

```
fun load(path: String, maxPages: Int): AnyFrame {
    // Initializes a mutable list to store rows of data.
    val rows = mutableListOf<AnyRow>()

    // Sets the initial page path for data loading.
    var pagePath = path
    do {
        // Loads data from the current page path.
        val row = load(pagePath)
        // Adds the loaded data as a row to the list.
        rows.add(row)

        // Retrieves the token for the next page, if available.
        val next = row.getValueOrNull<String>("nextPageToken")
        // Updates the page path for the next iteration, including the new token.
        pagePath = path + "&pageToken=" + next

        // Continues loading pages until there's no next page.
    } while (next != null && rows.size < maxPages)

    // Concatenates and returns all loaded rows as a DataFrame.
    return rows.concat()
}
```

4. Use the previously defined `load()` function to fetch data and create a DataFrame in a new code cell. This example fetches data, or in this case, videos related to Kotlin, with a maximum of 50 results per page, up to a maximum of 5 pages. The result is stored in the `df` variable:

```
val df = load("search?q=kotlin&maxResults=50&part=snippet", 5)
df
```

5. Finally, extract and concatenate items from the DataFrame:

```
val items = df.items.concat()
items
```

Clean and refine data

Cleaning and refining data are crucial steps in preparing your dataset for analysis. The Kotlin DataFrame library offers powerful functionalities for these tasks. Methods like `move`, `concat`, `select`, `parse`, and `join` are instrumental in organizing and transforming your data.

Let's explore an example where the data is already fetched using YouTube's data API. The goal is to clean and restructure the dataset to prepare for in-depth analysis:

1. You can start by reorganizing and cleaning your data. This involves moving certain columns under new headers and removing unnecessary ones for clarity:

```
val videos = items.dropNulls { id.videoId }
    .select { id.videoId named "id" and snippet }
    .distinct()
videos
```

2. Chunk IDs from the cleaned data and load corresponding video statistics. This involves breaking the data into smaller batches and fetching additional details:

```
val statPages = clean.id.chunked(50).map {
    val ids = it.joinToString("%2C")
    load("videos?part=statistics&id=$ids")
}
statPages
```

3. Concatenate the fetched statistics and select relevant columns:

```
val stats = statPages.items.concat().select { id and statistics.all() }.parse()
stats
```

4. Join the existing cleaned data with the newly fetched statistics. This merges two sets of data into a comprehensive DataFrame:

```
val joined = clean.join(stats)
joined
```

This example demonstrates how to clean, reorganize, and enhance your dataset using Kotlin DataFrame's various functions. Each step is designed to refine the data, making it more suitable for in-depth analysis.

Analyze data in Kotlin Notebook

After you've successfully fetched and cleaned and refined your data using functions from the Kotlin DataFrame library, the next step is to analyze this prepared dataset to extract meaningful insights.

Methods such as `groupBy` for categorizing data, `sum` and `maxBy` for summary statistics, and `sortBy` for ordering data are particularly useful. These tools allow you to perform complex data analysis tasks efficiently.

Let's look at an example, using `groupBy` to categorize videos by channel, `sum` to calculate total views per category, and `maxBy` to find the latest or most viewed video in each group:

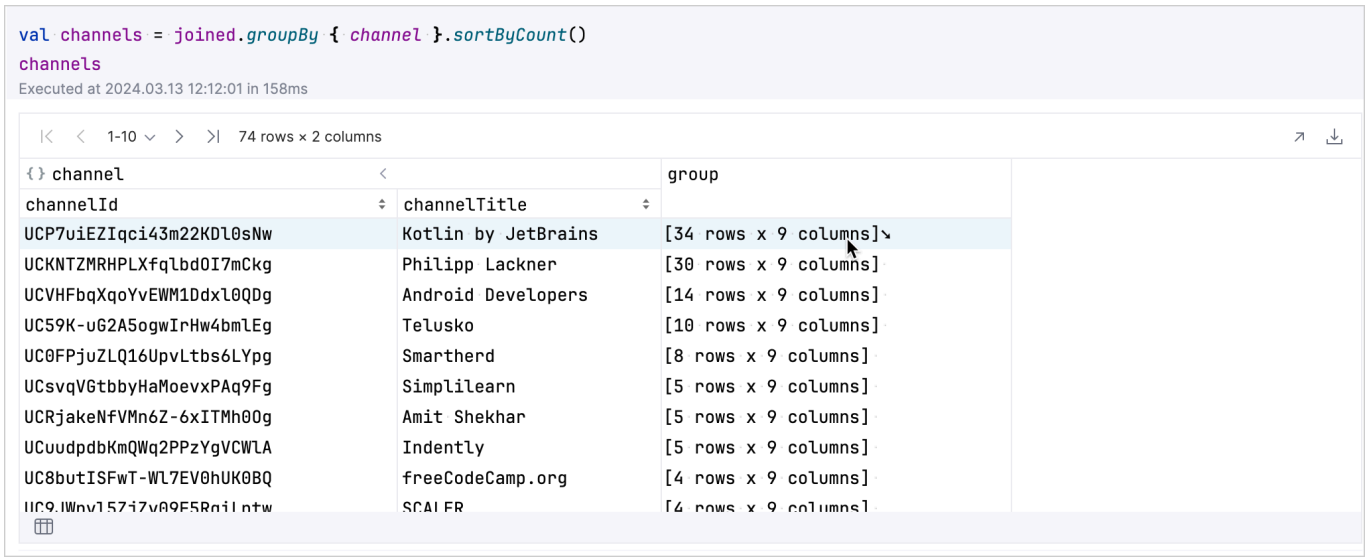
1. Simplify the access to specific columns by setting up references:

```
val view by column<Int>()
```

2. Use the `groupBy` method to group the data by the `channel` column and sort it.

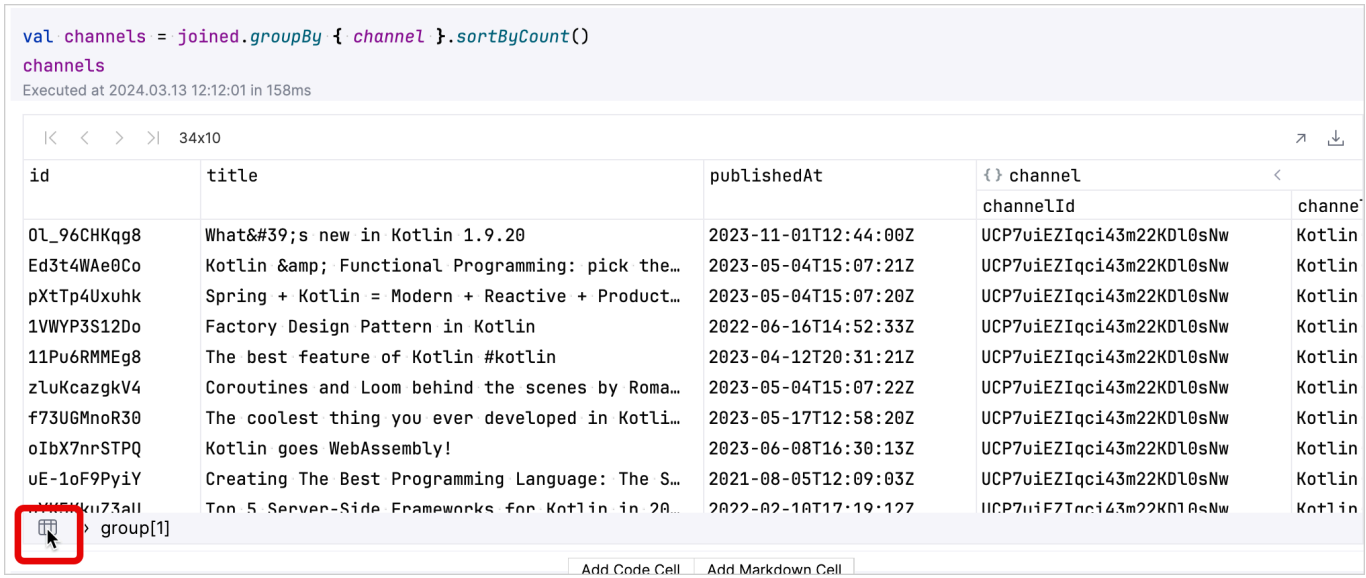
```
val channels = joined.groupBy { channel }.sortByCount()
```

In the resulting table, you can interactively explore the data. Clicking on the `group` field of a row corresponding to a channel expands that row to reveal more details about that channel's videos.



Expanding a row to reveal more details

You can click the table icon in the bottom left to return to the grouped dataset.



Click on the table icon in the bottom left to return

3. Use `aggregate`, `sum`, `maxBy`, and `flatten` to create a DataFrame summarizing each channel's total views and details of its latest or most viewed video:

```
val aggregated = channels.aggregate {
    viewCount.sum() into view

    val last = maxBy { publishedAt }
    last.title into "last title"
    last.publishedAt into "time"
    last.viewCount into "viewCount"
    // Sorts the DataFrame in descending order by view count and transform it into a flat
structure.
}.sortByDesc(view).flatten()
aggregated
```

The results of the analysis:

```
val view by column<Int>() // Simplify column access by setting up references, making subsequent operations more intuitive.

val channels = joined.groupBy { channel }.sortByCount().aggregate
 { this: AggregateGroupedDsl<_DataFrameType33>  it: AggregateGroupedDsl<_DataFrameType33>
    viewCount.sum() into view

    val last = maxBy { publishedAt }
    last.title into "last title"
    last.publishedAt into "time"
    last.viewCount into "viewCount" ^aggregate
}.sortByDesc(view).flatten() // Sort the DataFrame in descending order based on the values in the view column and transform it into
 a flat structure.
channels
Executed at 2024.03.04 10:53:40 in 127ms
```

1-10 78 rows × 6 columns

channelId	channelTitle	view	last title	time
UCVEANauRpBFEuaQrmt66cBA	Ketchupy Kotlin	5330608	KRÓL MYCIA RĄCZEK FEAT. KOTLIN - MYCIE RĄCZEK…	2020-11
UCxPD7bsocoAMq8Dj18kmGyQ	MoureDev by Brais Moure	2985221	KOTLIN: Curso ANDROID desde CERO para PRINCIP…	2020-03
UC8butISFwT-Wl7EV0hUK0BQ	freeCodeCamp.org	2884525	Kotlin & Android Development Course: Buil…	2023-10
UCVHFbqXqoYvEWM1Ddxl0QDg	Android Developers	2180328	5 years of Kotlin on Android #Shorts	2022-08
UCKNTZMRHPLXfqlbdOI7mCkg	Philipp Lackner	1253153	Kotlin Multiplatform in a Nutshell	2024-01
UC29ju8bIPH5as8OGnQzwJyA	Traversy Media	1089377	Build A Simple Android App With Kotlin	2020-11
UCsBjURrPoezykLs9EqgamOA	Fireship	1083892	Kotlin in 100 Seconds	2021-11
UC58_wzhvJta3hDSPvRLDAqg	Anuj Bhaiya	599068	Complete Android Development Roadmap for Begi…	2021-07
UCwRXb5dUK4cvsHbx-rGzSgw	Derek Banas	540471	Kotlin Tutorial	2017-05
UCP7uiEZIqci43m22KDl0sNw	Kotlin by JetBrains	509896	Advent of Code 2023 in Kotlin Day 12	2023-12

Analysis results

For more advanced techniques, see the Kotlin DataFrame documentation.

What's next

- Explore data visualization using the Kandy library
- Find additional information about data visualization in Data visualization with Kandy
- For an extensive overview of tools and resources available for data science and analysis in Kotlin, see Kotlin and Java libraries for data analysis

Connect and retrieve data from databases

Kotlin DataFrame library provides support for the most common SQL databases:

- DuckDB
- H2
- MariaDB
- Microsoft SQL Server
- MySQL
- PostgreSQL
- SQLite

Explore the Kotlin DataFrame SQL Examples on GitHub.

Before you start

> Starting with IntelliJ IDEA 2026.2, Kotlin Notebook will no longer be bundled with the IDE or officially supported by JetBrains. The source code will remain available on GitHub.
>
> Learn more in the blog post.

To follow this tutorial:

1. Select **File** | **New** | **Kotlin Notebook**.
2. Add the Java Database Connectivity (JDBC) driver dependency for your database in the first cell of your notebook.

   For example, to connect to a MariaDB database, add:

   ```
   USE {
       dependencies("org.mariadb.jdbc:mariadb-java-client:$version")
   }
   ```

3. Import Kotlin DataFrame:

   ```
   %use dataframe
   ```

   Run the code cell with the `%use dataframe` line before any other code cells to make sure the DataFrame library and its APIs are available in the notebook.

To follow the tutorial, you can also use DataFrame as a Gradle or Maven dependency.

Connect to a database

To connect to a database, create a connection configuration using the `DbConnectionConfig()` function:

1. Import the following functionality:

   ```
   import org.jetbrains.kotlinx.dataframe.io.DbConnectionConfig
   import org.jetbrains.kotlinx.dataframe.schema.DataFrameSchema
   ```

2. Define connection parameters (URL, username, password) using the `DbConnectionConfig()` function:

```
val URL = "YOUR_URL"
val USER_NAME = "YOUR_USERNAME"
val PASSWORD = "YOUR_PASSWORD"

val dbConfig = DbConnectionConfig(URL, USER_NAME, PASSWORD)
```

For more information on connecting to SQL databases, see Read from SQL databases in the Kotlin DataFrame documentation.

Inspect database schema

Before loading the data, inspect the database schemas to understand what tables you have and what columns they contain. You can use the schemas to decide which table to load into a DataFrame.

To retrieve schemas for all user-created tables in your database, use the `DataFrameSchema.readAllSqlTables()` function:

```
val dataSchemas = DataFrameSchema.readAllSqlTables(dbConfig)

dataSchemas.forEach { (tableName, schema) ->
    println("---Schema for table: $tableName---")
    println(schema)
    println()
}
```

Load data

After you inspect the database schemas and select the data, load the data into a DataFrame.

Kotlin DataFrame provides two ways to load data from a database:

- Load data directly from a table.
- Load the result of a custom SQL query.

Both approaches return a DataFrame that you can inspect, transform, and analyze.

Load data from a table

To load data from a table, use the `DataFrame.readSqlTable()` function.

The following example loads the first 100 rows from the `movies` table:

```
val moviesDf = DataFrame.readSqlTable(
    dbConfig = dbConfig,
    tableName = "movies",
    limit = 100
)

moviesDf
```

Load data with an SQL query

To execute a specific SQL query on your database, use the `DataFrame.readSqlQuery()` function. This approach is useful when you need to load specific columns, join tables, filter rows, or aggregate data in a database.

Let's retrieve a specific dataset related to movies directed by Quentin Tarantino. This query selects movie details and combines genres for each movie:

```
val TARANTINO_FILMS_SQL_QUERY = """
    SELECT name, year, rank, GROUP_CONCAT(genre) as "genres"
    FROM movies JOIN movies_directors ON movie_id = movies.id
    JOIN directors ON directors.id=director_id LEFT JOIN movies_genres ON movies.id =
movies_genres.movie_id
    WHERE directors.first_name = "Quentin" AND directors.last_name = "Tarantino"
    GROUP BY name, year, rank
    ORDER BY year
    """

val tarantinoMoviesDf = DataFrame.readSqlQuery(dbConfig, TARANTINO_FILMS_SQL_QUERY)

tarantinoMoviesDf
```

Process data

After loading your database into a DataFrame, you can use DataFrame operations to process retrieved data.

For example, let's manipulate data from the previous section. The following code:

1. Replaces missing values in the `year` column using the `.fillNA()` function.
2. Converts the column to `Int` using the `.convert()` function.
3. Keeps only films released after 2000 using the `.filter()` function.

```
val filteredTarantinoMovies = tarantinoMoviesDf
    .fillNA { year }.with { 0 }
    .convert { year }.toInt()
    .filter { year > 2000 }

filteredTarantinoMovies
```

Analyze data

Use the DataFrame library to group, sort, and aggregate data so you can uncover and understand patterns in your data.

For example, let's read actor data from the `actors` table and find the 20 most common first names:

```
// Extract data from the actors table
val actorDf = DataFrame.readSqlTable(dbConfig, "actors", 10000)
val top20ActorNames = actorDf
   // Groups the data by the first_name column
   .groupBy { first_name }

   // Counts the occurrences of each unique first name
   .count()

    // Sorts the results in descending order of count
   .sortByDesc("count")

   // Selects the top 20 most frequent names for analysis.
   .take(20)
```

What's next

- Explore data visualization using the Kandy library
- Find additional information about data visualization in Data visualization with Kandy
- For an extensive overview of tools and resources available for data science and analysis in Kotlin, see Kotlin and Java libraries for data analysis

Data visualization with Kandy

Kotlin offers an all-in-one-place solution for powerful and flexible data visualization, providing an intuitive way to present and explore data before diving into complex models.

This tutorial demonstrates how to create different chart types in IntelliJ IDEA using the Kandy and Kotlin DataFrame libraries.

Before you start

Starting with IntelliJ IDEA 2026.2, Kotlin Notebook will no longer be bundled with the IDE or officially supported by JetBrains. The source code will remain available on GitHub.

Learn more in the blog post.

To follow this tutorial:

1. Select **File** | **New** | **Kotlin Notebook**.
2. In your notebook, import Kandy and Kotlin DataFrame:

   ```
   %use kandy
   %use dataframe
   ```

   Run the code cell with the `%use dataframe` line before any other code cells to make sure the DataFrame library and its APIs are available in the notebook.

To follow the tutorial, you can also use DataFrame as a Gradle or Maven dependency.

Create a DataFrame

To start, let's create a DataFrame with data to visualize. This DataFrame stores simulated monthly average temperatures for Berlin, Madrid, and Caracas:

```
// The months variable stores a list with the 12 months of the year
val months = listOf(
    "January", "February",
    "March", "April", "May",
    "June", "July", "August",
    "September", "October", "November",
    "December"
)
// The tempBerlin, tempMadrid, and tempCaracas variables store a list
// with temperature values for each month
val tempBerlin =
    listOf(-0.5, 0.0, 4.8, 9.0, 14.3, 17.5, 19.2, 18.9, 14.5, 9.7, 4.7, 1.0)
val tempMadrid =
    listOf(6.3, 7.9, 11.2, 12.9, 16.7, 21.1, 24.7, 24.2, 20.3, 15.4, 9.9, 6.6)
val tempCaracas =
    listOf(27.5, 28.9, 29.6, 30.9, 31.7, 35.1, 33.8, 32.2, 31.3, 29.4, 28.9, 27.6)
```

Now let's create a new variable (`df`) and use the `dataFrameOf()` function to generate a DataFrame of three columns (Month, Temperature, and City):

```
val df = dataFrameOf(
    "Month" to months + months + months,
    "Temperature" to tempBerlin + tempMadrid + tempCaracas,
    "City" to List(12) { "Berlin" } + List(12) { "Madrid" } + List(12) { "Caracas" }
)
```

To preview the data, use the `.head()` function:

```
df.head(4) // Returns the first four rows
```

In our dataset, the first four rows store temperature in Berlin from January to April:

4 rows 4 rows × 3 columns

Month	Temperature	City
January	-0.5	Berlin
February	0	Berlin
March	4.8	Berlin
April	9	Berlin

Dataframe exploration

There are different options to access a column's records that can help you increase type safety when working with the Kandy and Kotlin DataFrame libraries together. For more information, see Access APIs.

Create a line chart

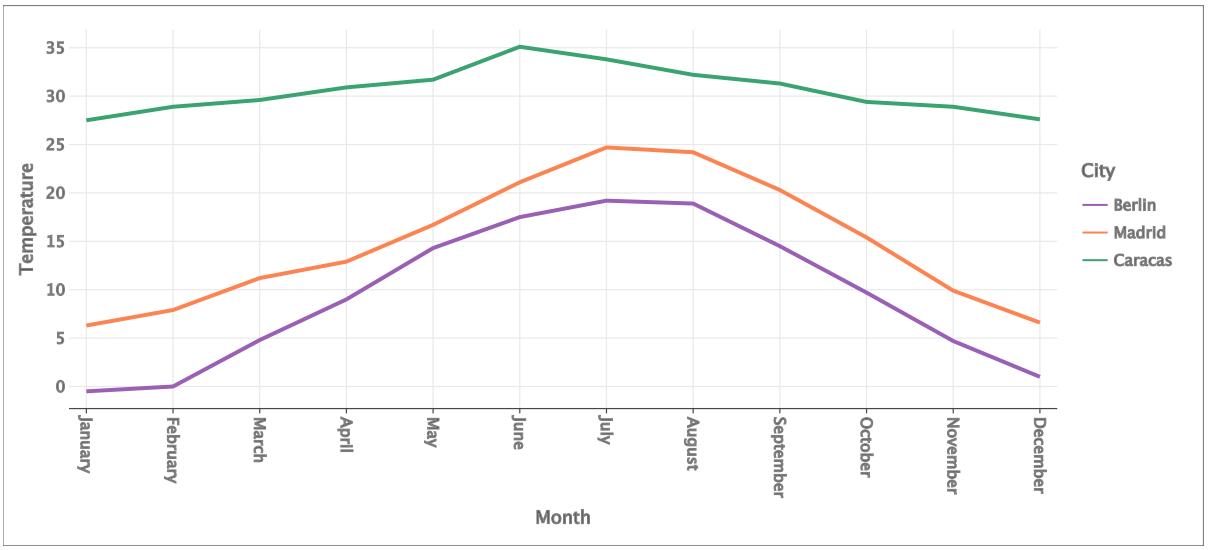
Let's create a line chart using the `df` DataFrame from the previous section:

1. Call the `.plot()` function from the Kandy library.
2. Apply the `line()` layer.
3. Map the `Month` and `Temperature` columns to the `X` and `Y` axes accordingly.
4. (Optional) Customize colors and sizes.

```
df.plot {
    line {
        x(Month)
        y(Temperature)

        color(City) {
            scale = categorical(
                "Berlin" to Color.hex("#6F4E37"),
                "Madrid" to Color.hex("#C2D4AB"),
                "Caracas" to Color.hex("#B5651D")
            )
        }
        width = 1.5
    }
    layout {
        size = 1000 to 450
    }
}
```

Here's the result:



Line chart

Create a points chart

Now, let's visualize the `df` DataFrame in a points (scatter) chart:

1. Call the `.plot()` function from the Kandy library.
2. Apply the `points()` layer.

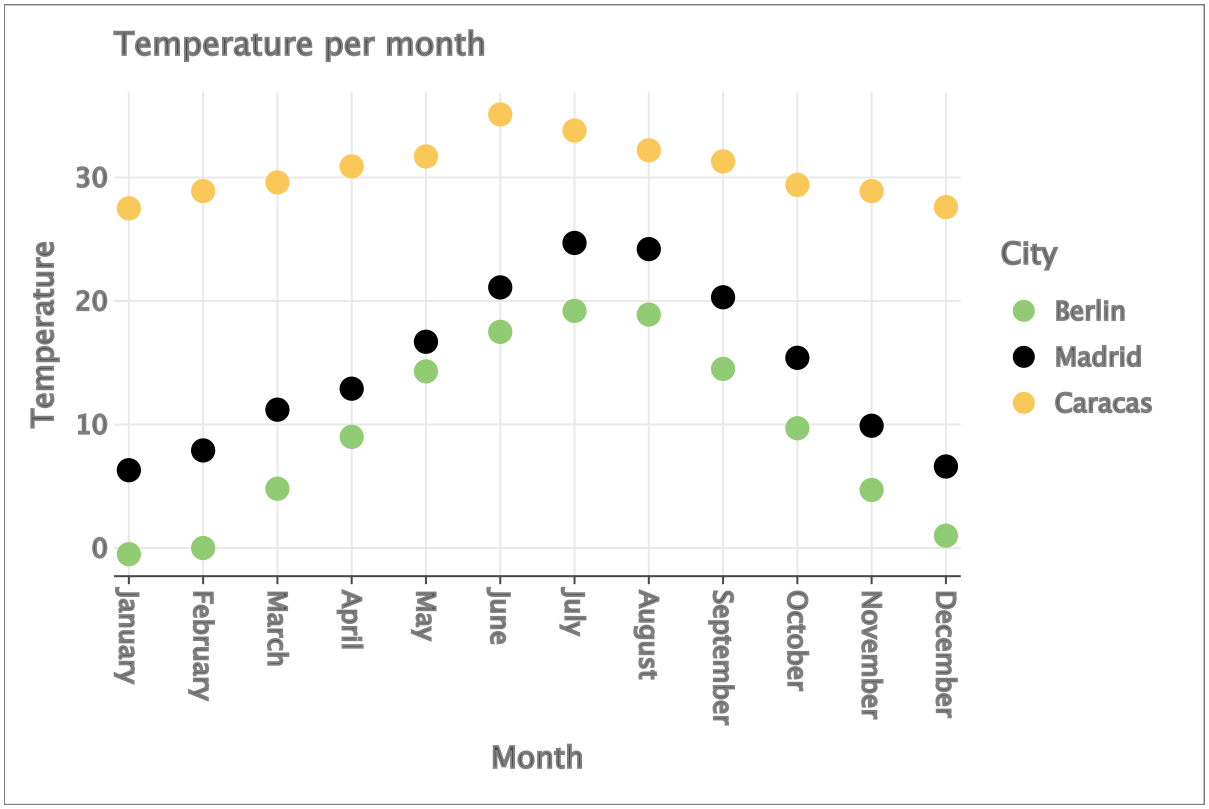
3. Map the `Month` and `Temperature` columns to the `X` and `Y` axes accordingly.

4. (Optional) Customize colors, axis labels, point sizes, and chart title.

```
df.plot {
   points {
      x(Month) {
         axis.name = "Month"
      }
      y(Temperature) {
         axis.name = "Temperature"
      }

      color(City) {
         scale = categorical(
            "Berlin" to Color.hex("#6F4E37"),
            "Madrid" to Color.hex("#C2D4AB"),
            "Caracas" to Color.hex("#B5651D")
         )
      }
      size = 5.5
   }
   layout {
      title = "Temperature per month"
   }
}
```

Here's the result:



Points chart

Create a bar chart

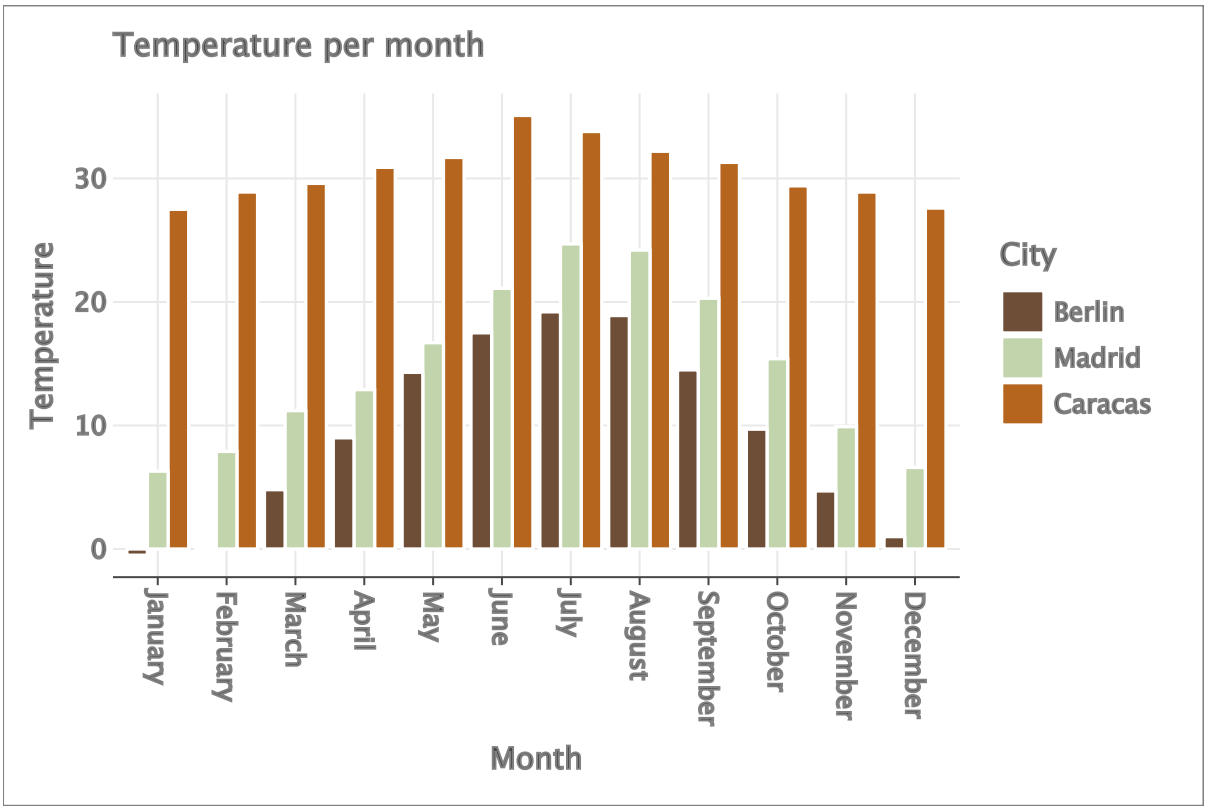
Finally, let's create a bar chart for each city:

1. Use the `.groupBy()` function to group the DataFrame by the `City` column.
2. Call the `plot()` function from the Kandy library.
3. Apply the `bars()` layer.
4. (Optional) Add a title for the chart, customize colors.

```
df.groupBy { City }.plot {
    bars {
        x(Month)
        y(Temperature)

        fillColor(City) {
            scale = categorical(
                "Berlin" to Color.hex("#6F4E37"),
                "Madrid" to Color.hex("#C2D4AB"),
                "Caracas" to Color.hex("#B5651D")
            )
        }
    }
    layout.title {
       title = "Temperature per month"
    }
}
```

Here's the result:



Bar chart

What's next

- Explore more chart examples in the Kandy library documentation
- Explore more advanced plotting options in the Lets-Plot library documentation

- Find additional information about creating, exploring, and managing data frames in the Kotlin DataFrame library documentation

Data visualization with Lets-Plot for Kotlin

Lets-Plot for Kotlin (LPK) is a multiplatform plotting library that ports the R's ggplot2 library to Kotlin. LPK brings the feature-rich ggplot2 API to the Kotlin ecosystem, making it suitable for scientists and statisticians who require sophisticated data visualization capabilities.

LPK targets various platforms, including Kotlin/JS, JVM's Swing, JavaFX, and Compose Multiplatform. Additionally, LPK has seamless integration with IntelliJ, DataGrip, DataSpell, and PyCharm.



Lets-Plot

This tutorial demonstrates how to create different plot types with the LPK and Kotlin DataFrame libraries in IntelliJ IDEA.

Before you start

Starting with IntelliJ IDEA 2026.2, Kotlin Notebook will no longer be bundled with the IDE or officially supported by JetBrains. The source code will remain available on GitHub.

Learn more in the blog post.

Create a new Kotlin Notebook to work with Lets-Plot:

1. Select **File** | **New** | **Kotlin Notebook**.
2. In your notebook, import the LPK and Kotlin DataFrame libraries by running the following command:

```
%use lets-plot
%use dataframe
```

To follow the tutorial, you can also use DataFrame as a Gradle or Maven dependency.

Prepare the data

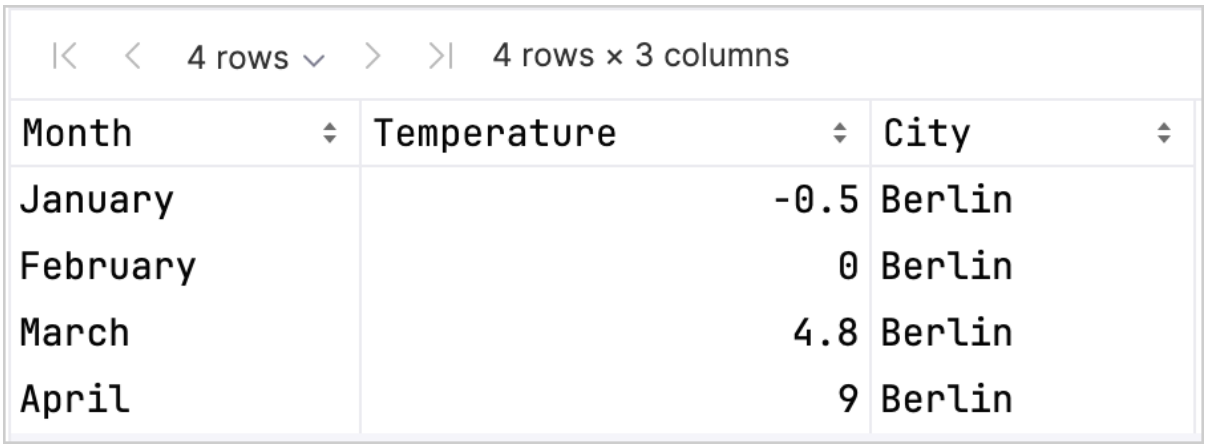
Let's create a DataFrame that stores simulated numbers of the monthly average temperature in three cities: Berlin, Madrid, and Caracas.

Use the `dataFrameOf()` function from the Kotlin DataFrame library to generate the DataFrame. Paste and run the following code snippet:

```
// The months variable stores a list with 12 months of the year
val months = listOf(
    "January", "February",
    "March", "April", "May",
    "June", "July", "August",
    "September", "October", "November",
    "December"
)
// The tempBerlin, tempMadrid, and tempCaracas variables store a list with temperature values for each
month
val tempBerlin =
    listOf(-0.5, 0.0, 4.8, 9.0, 14.3, 17.5, 19.2, 18.9, 14.5, 9.7, 4.7, 1.0)
val tempMadrid =
    listOf(6.3, 7.9, 11.2, 12.9, 16.7, 21.1, 24.7, 24.2, 20.3, 15.4, 9.9, 6.6)
val tempCaracas =
    listOf(27.5, 28.9, 29.6, 30.9, 31.7, 35.1, 33.8, 32.2, 31.3, 29.4, 28.9, 27.6)

// The df variable stores a DataFrame of three columns, including monthly records, temperature, and
cities
val df = dataFrameOf(
    "Month" to months + months + months,
    "Temperature" to tempBerlin + tempMadrid + tempCaracas,
    "City" to List(12) { "Berlin" } + List(12) { "Madrid" } + List(12) { "Caracas" }
)
df.head(4)
```

You can see that the DataFrame has three columns: Month, Temperature, and City. The first four rows of the DataFrame contain records of the temperature in Berlin from January to April:



4 rows 4 rows × 3 columns

Month	Temperature	City
January	-0.5	Berlin
February	0	Berlin
March	4.8	Berlin
April	9	Berlin

Dataframe exploration

To create a plot using the LPK library, you need to convert your data (`df`) into a `Map` type that stores the data in key-value pairs. You can easily convert a DataFrame into a `Map` using the `.toMap()` function:

```
val data = df.toMap()
```

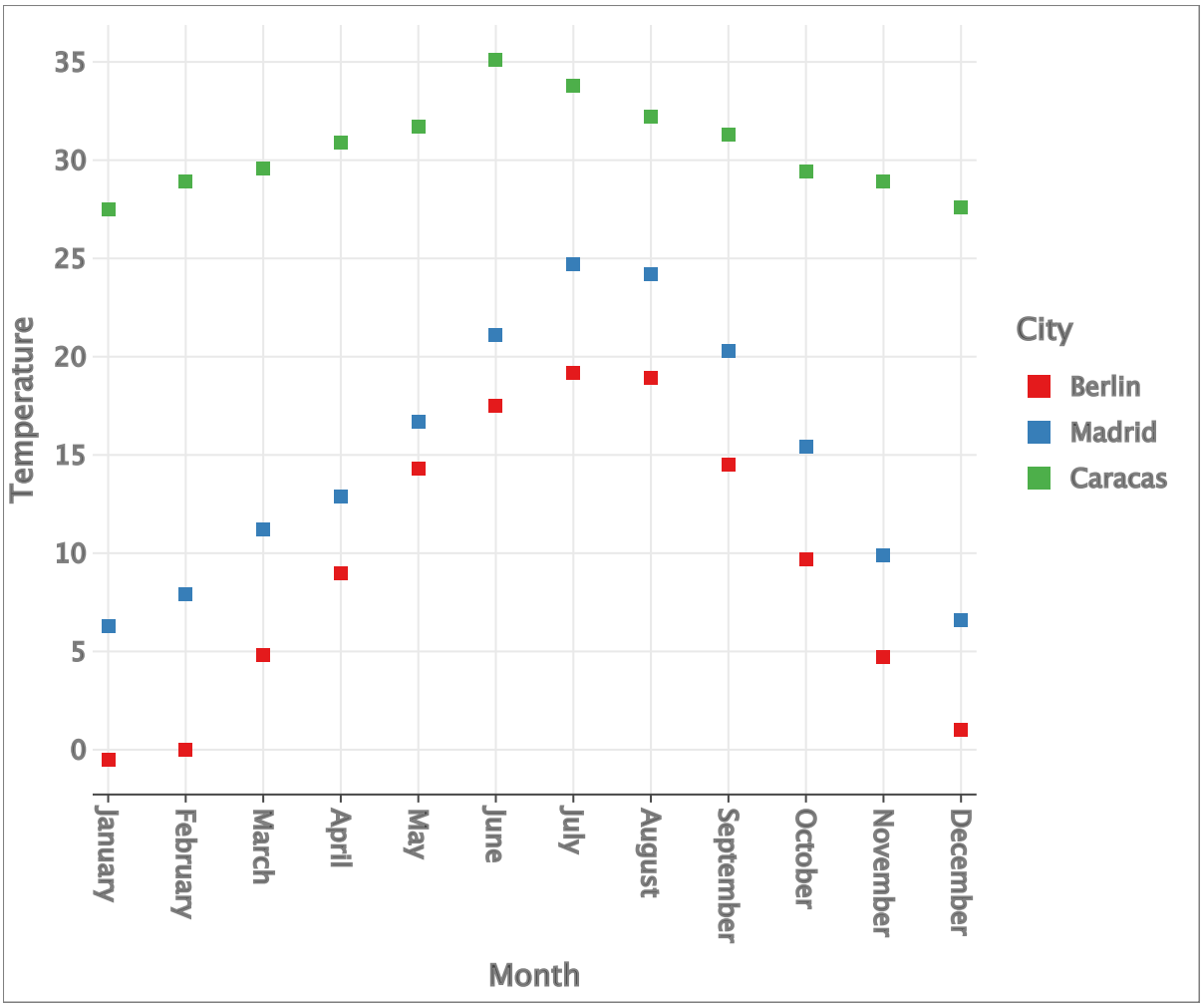
Create a scatter plot

Let's create a scatter plot with the LPK library.

Once you have your data in the `Map` format, use the `geomPoint()` function from the LPK library to generate the scatter plot. You can specify the values for the X and Y axes, as well as define categories and their color. Additionally, you can customize the plot's size and point shapes to suit your needs:

```
// Specifies X and Y axes, categories and their color, plot size, and plot type
val scatterPlot =
    letsPlot(data) { x = "Month"; y = "Temperature"; color = "City" } + ggsize(600, 500) +
geomPoint(shape = 15)
scatterPlot
```

Here's the result:



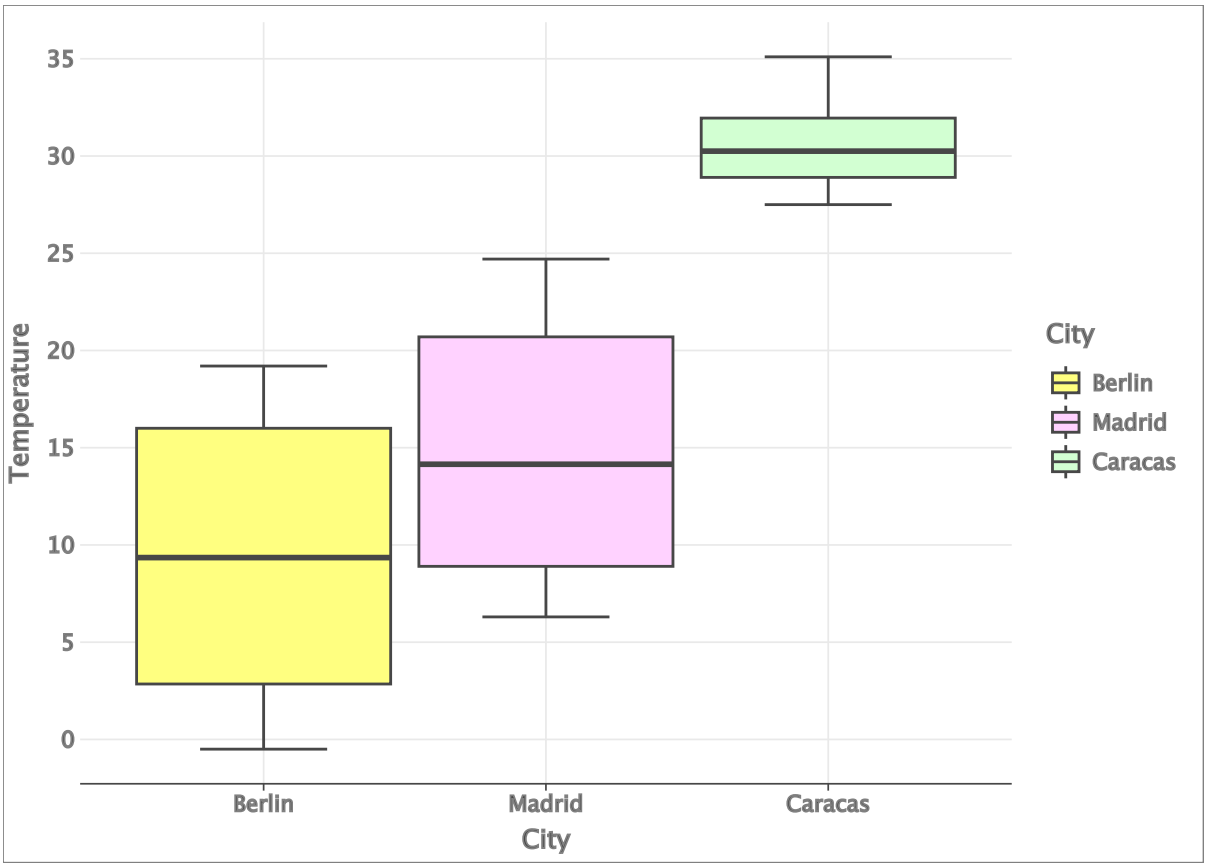
Scatter plot

Create a box plot

Let's visualize the data in a box plot. Use the `geomBoxplot()` function from the LPK library to generate the plot and customize colors with the `scaleFillManual()` function:

```
// Specifies X and Y axes, categories, plot size, and plot type
val boxPlot = ggplot(data) { x = "City"; y = "Temperature" } + ggsize(700, 500) + geomBoxplot { fill =
"City" } +
    // Customizes colors
    scaleFillManual(values = listOf("light_yellow", "light_magenta", "light_green"))
boxPlot
```

Here's the result:



Box plot

Create a 2D density plot

Now, let's create a 2D density plot to visualize the distribution and concentration of some random data.

Prepare the data for the 2D density plot

1. Import the dependencies to process the data and generate the plot:

```
%use lets-plot

@file:DependsOn("org.apache.commons:commons-math3:3.6.1")
import org.apache.commons.math3.distribution.MultivariateNormalDistribution
```

2. Paste and run the following code snippet to create sets of 2D data points:

```
// Defines covariance matrices for three distributions
val cov0: Array<DoubleArray> = arrayOf(
    doubleArrayOf(1.0, -.8),
    doubleArrayOf(-.8, 1.0)
)

val cov1: Array<DoubleArray> = arrayOf(
    doubleArrayOf(1.0, .8),
    doubleArrayOf(.8, 1.0)
)

val cov2: Array<DoubleArray> = arrayOf(
    doubleArrayOf(10.0, .1),
    doubleArrayOf(.1, .1)
)

// Defines the number of samples
val n = 400

// Defines means for three distributions
val means0: DoubleArray = doubleArrayOf(-2.0, 0.0)
val means1: DoubleArray = doubleArrayOf(2.0, 0.0)
val means2: DoubleArray = doubleArrayOf(0.0, 1.0)

// Generates random samples from three multivariate normal distributions
val xy0 = MultivariateNormalDistribution(means0, cov0).sample(n)
val xy1 = MultivariateNormalDistribution(means1, cov1).sample(n)
val xy2 = MultivariateNormalDistribution(means2, cov2).sample(n)
```

From the code above, the `xy0`, `xy1`, and `xy2` variables store arrays with 2D (`x, y`) data points.

3. Convert your data into a `Map` type:

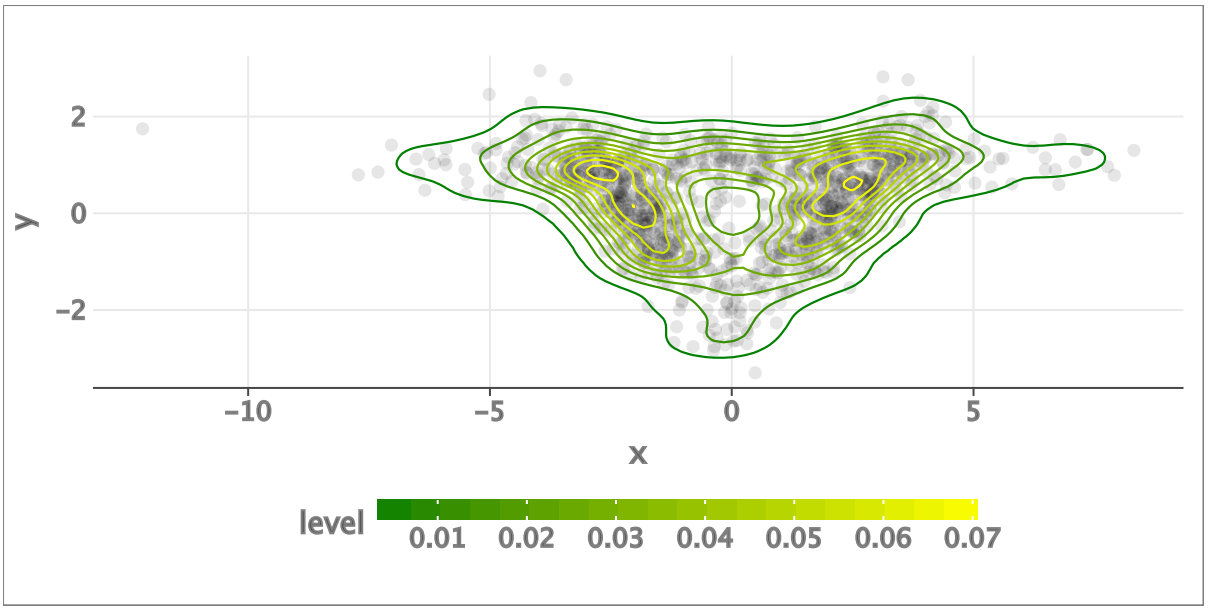
```
val data = mapOf(
    "x" to (xy0.map { it[0] } + xy1.map { it[0] } + xy2.map { it[0] }).toList(),
    "y" to (xy0.map { it[1] } + xy1.map { it[1] } + xy2.map { it[1] }).toList()
)
```

Generate the 2D density plot

Using the `Map` from the previous step, create a 2D density plot (`geomDensity2D`) with a scatter plot (`geomPoint`) in the background to better visualize the data points and outliers. You can use the `scaleColorGradient()` function to customize the scale of colors:

```
val densityPlot = letsPlot(data) { x = "x"; y = "y" } + ggsize(600, 300) + geomPoint(
    color = "black",
    alpha = .1
) + geomDensity2D { color = "..level.." } +
        scaleColorGradient(low = "dark_green", high = "yellow", guide = guideColorbar(barHeight = 10,
barWidth = 300)) +
        theme().legendPositionBottom()
densityPlot
```

Here's the result:



2D density plot

What's next

- Explore more plot examples in the Lets-Plot for Kotlin's documentation.
- Check the Lets-Plot for Kotlin's API reference.
- Learn about transforming and visualizing data with Kotlin in the Kotlin DataFrame and Kandy library documentation.

Kotlin and Java libraries for data analysis

From data collection to model building, Kotlin offers robust libraries facilitating different tasks in the data pipeline.

In addition to its own libraries, Kotlin is 100% interoperable with Java. This interoperability helps to leverage the entire ecosystem of tried-and-true Java libraries with excellent performance. With this perk, you can easily use either Kotlin or Java libraries when working on Kotlin data projects.

Kotlin libraries

Library	Purpose	Features
Kotlin DataFrame	• Data collection • Data cleaning and processing	• Operations for creating, sorting, and cleaning data frames, feature engineering, and more • Processing of structured data • Support for CSV, JSON, and other input formats • Reading from SQL databases • Connecting with different APIs to access data and increase type safety

Library	Purpose	Features
Kandy	• Data exploration and visualization	• Powerful, readable, and typesafe DSL for plotting charts of various types • Open-source library written in Kotlin for the JVM • Support for Datalore • Seamless integration with Kotlin DataFrame
KotlinDL	• Model building	• Deep learning API written in Kotlin and inspired by Keras • Training deep learning models from scratch or importing existing Keras and ONNX models for inference • Transferring learning for tailoring existing pre-trained models to your tasks • Support for the Android platform
Multik	• Data cleaning and processing • Model building	• Mathematical operations over multidimensional arrays (linear algebra, statistics, arithmetics, and other calculations) • Creating, copying, indexing, slicing, and other array operations • Kotlin-idiomatic library with benefits such as type and dimension safety and swappable computational engines, running on the JVM or as native code
Kotlin for Apache Spark	• Data collection • Data cleaning and processing • Data exploration and visualization • Model building	• Layer of compatibility between Apache Spark and Kotlin • Apache Spark data transformation operations in Kotlin-idiomatic code • Simple usage of Kotlin features, such as data classes and lambda expressions, in curly braces or method reference
Lets-Plot	• Data exploration and visualization	• Plotting statistical data written in Kotlin • Support for Datalore • Compatible with the JVM, JS, and Python • Embedding charts in Compose Multiplatform applications
KMath	• Data cleaning and processing • Data exploration and visualization • Model building	• Modular library to work with mathematical abstractions in Kotlin Multiplatform (JVM, JS, Native, and Wasm) • APIs for algebraic structures, mathematical expressions, histograms, and streaming operations • Interchangeable wrappers over existing Java and Kotlin libraries, including ND4J, Apache Commons Math, and Multik •

Library	Purpose	Features
		Inspired by Python's NumPy but with other additional features like type safety
kravis	• Data exploration and visualization	• Visualization of tabular data • Inspired by R's ggplot

Java libraries

Since Kotlin provides first-class interoperability with Java, you can use Java libraries for data tasks in your Kotlin code. Here are some examples of such libraries:

Library	Purpose	Features
Tablesaw	• Data collection • Data cleaning and processing • Data exploration and visualization	• Tools for loading, cleaning, transforming, filtering, and summarizing data • Inspired by Plot.ly
CoreNLP	• Data cleaning and processing	• Natural language processing toolkit • Linguistic annotations for text, such as sentiment and quote attributions • Support for eight languages
Smile	• Data cleaning and processing • Data exploration and visualization • Model building	• Ready-made algorithms for machine learning and natural language processing • Linear algebra, graph, interpolation, and visualization tools • Provides functional Kotlin API, Scala API, Clojure API, and more
Smile-NLP-kt	• Data cleaning and processing	• Kotlin rewrite of the Scala implicits for the natural language processing part of Smile • Operations in the format of Kotlin extension functions and interfaces • Sentence breaking, stemming, bag of words, and other tasks

Library	Purpose	Features
ND4J	• Data cleaning and processing • Model building	• Matrix mathematics library for the JVM • Over 500 mathematical, linear algebra, and deep learning operations
Apache Commons Math	• Data cleaning and processing • Model building	• Mathematics and statistics operations for Java • Correlations, distributions, linear algebra, geometry, and other operations • Machine learning models
NM Dev	• Data cleaning and processing • Model building	• Java math library of numerical algorithms • Object-oriented numerical methods • Linear algebra, optimization, statistics, calculus, and more operations
Apache OpenNLP	• Data cleaning and processing • Model building	• Machine-learning-based toolkit for the processing of natural language text • Tokenization, sentence segmentation, part-of-speech tagging, and other tasks • Built-in tools for data modeling and model validation
Charts	• Data exploration and visualization	• JavaFX library for scientific charts • Complex charts, such as logarithmic, heatmap, and force-directed graph
DeepLearning4J	• Model building	• Deep learning library for Java • Importing and retraining models (Pytorch, Tensorflow, Keras) • Deploying in JVM microservice environments, mobile devices, IoT, and Apache Spark
Timefold	• Model building	• Solver utility for optimization planning problems • Compatible with object-oriented and functional programming

Collections overview

The Kotlin Standard Library provides a comprehensive set of tools for managing *collections* – groups of a variable number of items (possibly zero) that are significant to the problem being solved and are commonly operated on.

Collections are a common concept for most programming languages, so if you're familiar with, for example, Java or Python collections, you can skip this introduction and proceed to the detailed sections.

A collection usually contains a number of objects of the same type (and its subtypes). Objects in a collection are called *elements* or *items*. For example, all the students in a department form a collection that can be used to calculate their average age.

The following collection types are relevant for Kotlin:

- *List* is an ordered collection with access to elements by indices – integer numbers that reflect their position. Elements can occur more than once in a list. An example of a list is a telephone number: it's a group of digits, their order is important, and they can repeat.
- *Set* is a collection of unique elements. It reflects the mathematical abstraction of set: a group of objects without repetitions. Generally, the order of set elements has no significance. For example, the numbers on lottery tickets form a set: they are unique, and their order is not important.
- *Map* (or *dictionary*) is a set of key-value pairs. Keys are unique, and each of them maps to exactly one value. The values can be duplicates. Maps are useful for storing logical connections between objects, for example, an employee's ID and their position.

Kotlin lets you manipulate collections independently of the exact type of objects stored in them. In other words, you add a `String` to a list of `String`s the same way as you would do with `Int`s or a user-defined class. So, the Kotlin Standard Library offers generic interfaces, classes, and functions for creating, populating, and managing collections of any type.

The collection interfaces and related functions are located in the `kotlin.collections` package. Let's get an overview of its contents.

Arrays are not a type of collection. For more information, see Arrays.

Collection types

The Kotlin Standard Library provides implementations for basic collection types: sets, lists, and maps. A pair of interfaces represent each collection type:

- A *read-only* interface that provides operations for accessing collection elements.
- A *mutable* interface that extends the corresponding read-only interface with write operations: adding, removing, and updating its elements.

Note that a mutable collection doesn't have to be assigned to a `var`. Write operations with a mutable collection are still possible even if it is assigned to a `val`. The benefit of assigning mutable collections to `val` is that you protect the reference to the mutable collection from modification. Over time, as your code grows and becomes more complex, it becomes even more important to prevent unintentional modification to references. Use `val` as much as possible for safer and more robust code. If you try to reassign a `val` collection, you get a compilation error:

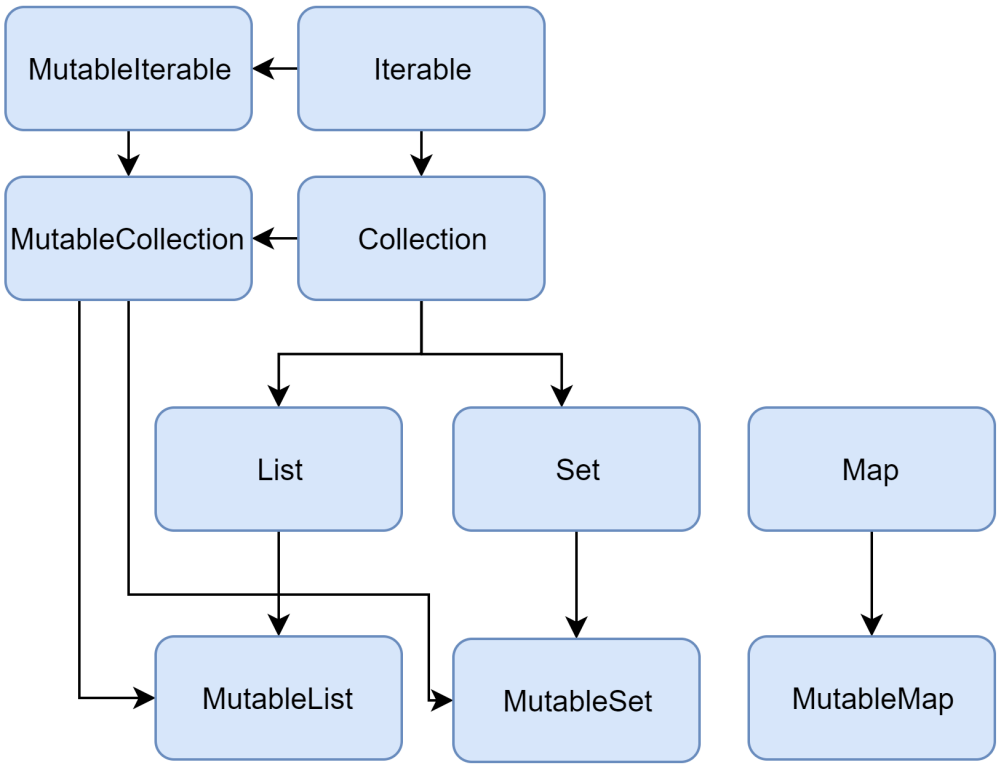
```
fun main() {
    val numbers = mutableListOf("one", "two", "three", "four")
    numbers.add("five")   // this is OK
    println(numbers)
    //numbers = mutableListOf("six", "seven")      // compilation error

}
```

The read-only collection types are covariant. This means that if a `Rectangle` class inherits from `Shape`, you can use a `List<Rectangle>` anywhere the `List<Shape>` is required. In other words, the collection types have the same subtyping relationship as the element types. Maps are covariant on the value type, but not on the key type.

In turn, mutable collections aren't covariant; otherwise, this would lead to runtime failures. If `MutableList<Rectangle>` was a subtype of `MutableList<Shape>`, you could insert other `Shape` inheritors (for example, `Circle`) into it, thus violating its `Rectangle` type argument.

Below is a diagram of the Kotlin collection interfaces:



Collection interfaces hierarchy

Let's walk through the interfaces and their implementations. To learn about `Collection`, read the section below. To learn about `List`, `Set`, and `Map`, you can either read the corresponding sections or watch a video by Sebastian Aigner, Kotlin Developer Advocate:

Watch video online.

Collection

`Collection<T>` is the root of the collection hierarchy. This interface represents the common behavior of a read-only collection: retrieving size, checking item membership, and so on. `Collection` inherits from the `Iterable<T>` interface that defines the operations for iterating elements. You can use `Collection` as a parameter of a function that applies to different collection types. For more specific cases, use the `Collection`'s inheritors: `List` and `Set`.

```
fun printAll(strings: Collection<String>) {
    for(s in strings) print("$s ")
    println()
}

fun main() {
    val stringList = listOf("one", "two", "one")
    printAll(stringList)

    val stringSet = setOf("one", "two", "three")
    printAll(stringSet)
}
```

`MutableCollection<T>` is a `Collection` with write operations, such as `add` and `remove`.

```
fun List<String>.getShortWordsTo(shortWords: MutableList<String>, maxLength: Int) {
    this.filterTo(shortWords) { it.length <= maxLength }
    // throwing away the articles
    val articles = setOf("a", "A", "an", "An", "the", "The")
    shortWords -= articles
}

fun main() {
    val words = "A long time ago in a galaxy far far away".split(" ")
    val shortWords = mutableListOf<String>()
    words.getShortWordsTo(shortWords, 3)
    println(shortWords)
}
```

List

`List<T>` stores elements in a specified order and provides indexed access to them. Indices start from zero – the index of the first element – and go to `lastIndex` which is the `(list.size - 1)`.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    println("Number of elements: ${numbers.size}")
    println("Third element: ${numbers.get(2)}")
    println("Fourth element: ${numbers[3]}")
    println("Index of element \"two\" ${numbers.indexOf("two")}")
}
```

List elements (including nulls) can duplicate: a list can contain any number of equal objects or occurrences of a single object. Two lists are considered equal if they have the same sizes and structurally equal elements at the same positions.

```
data class Person(var name: String, var age: Int)

fun main() {
    val bob = Person("Bob", 31)
    val people = listOf(Person("Adam", 20), bob, bob)
    val people2 = listOf(Person("Adam", 20), Person("Bob", 31), bob)
    println(people == people2)
    bob.age = 32
    println(people == people2)
}
```

`MutableList<T>` is a `List` with list-specific write operations, for example, to add or remove an element at a specific position.

```
fun main() {
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.add(5)
    numbers.removeAt(1)
    numbers[0] = 0
    numbers.shuffle()
    println(numbers)
}
```

As you see, in some aspects lists are very similar to arrays. However, there is one important difference: an array's size is defined upon initialization and is never changed; in turn, a list doesn't have a predefined size; a list's size can be changed as a result of write operations: adding, updating, or removing elements.

In Kotlin, the default implementation of `MutableList` is `ArrayList` which you can think of as a resizable array.

Set

`Set<T>` stores unique elements; their order is generally undefined. `null` elements are unique as well: a `Set` can contain only one `null`. Two sets are equal if they have the same size, and for each element of a set there is an equal element in the other set.

```
fun main() {
    val numbers = setOf(1, 2, 3, 4)
    println("Number of elements: ${numbers.size}")
    if (numbers.contains(1)) println("1 is in the set")

    val numbersBackwards = setOf(4, 3, 2, 1)
    println("The sets are equal: ${numbers == numbersBackwards}")
}
```

`MutableSet` is a `Set` with write operations from `MutableCollection`.

The default implementation of `MutableSet` – `LinkedHashSet` – preserves the order of elements insertion. Hence, the functions that rely on the order, such as `first()` or `last()`, return predictable results on such sets.

```
fun main() {
    val numbers = setOf(1, 2, 3, 4)  // LinkedHashSet is the default implementation
    val numbersBackwards = setOf(4, 3, 2, 1)

    println(numbers.first() == numbersBackwards.first())
    println(numbers.first() == numbersBackwards.last())
}
```

An alternative implementation – `HashSet` – says nothing about the elements order, so calling such functions on it returns unpredictable results. However, `HashSet` requires less memory to store the same number of elements.

Map

`Map<K, V>` is not an inheritor of the `Collection` interface; however, it's a Kotlin collection type as well. A `Map` stores *key-value* pairs (or *entries*); keys are unique, but different keys can be paired with equal values. The `Map` interface provides specific functions, such as access to value by key, searching keys and values, and so on.

```
fun main() {
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)

    println("All keys: ${numbersMap.keys}")
    println("All values: ${numbersMap.values}")
    if ("key2" in numbersMap) println("Value by key \"key2\": ${numbersMap["key2"]}")
    if (1 in numbersMap.values) println("The value 1 is in the map")
    if (numbersMap.containsValue(1)) println("The value 1 is in the map") // same as previous
}
```

Two maps containing the equal pairs are equal regardless of the pair order.

```
fun main() {
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
    val anotherMap = mapOf("key2" to 2, "key1" to 1, "key4" to 1, "key3" to 3)

    println("The maps are equal: ${numbersMap == anotherMap}")
}
```

`MutableMap` is a `Map` with map write operations, for example, you can add a new key-value pair or update the value associated with the given key.

```
fun main() {
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap.put("three", 3)
    numbersMap["one"] = 11

    println(numbersMap)
}
```

The default implementation of `MutableMap` – `LinkedHashMap` – preserves the order of elements insertion when iterating the map. In turn, an alternative implementation – `HashMap` – says nothing about the elements order.

ArrayDeque

`ArrayDeque<T>` is an implementation of a double-ended queue, which allows you to add or remove elements both at the beginning or end of the queue. As such, `ArrayDeque` also fills the role of both a Stack and Queue data structure in Kotlin. Behind the scenes, `ArrayDeque` is realized using a resizable array that automatically adjusts in size when required:

```
fun main() {
    val deque = ArrayDeque(listOf(1, 2, 3))

    deque.addFirst(0)
    deque.addLast(4)
    println(deque) // [0, 1, 2, 3, 4]

    println(deque.first()) // 0
    println(deque.last()) // 4

    deque.removeFirst()
    deque.removeLast()
    println(deque) // [1, 2, 3]
}
```

Constructing collections

Construct from elements

The most common way to create a collection is with the standard library functions `listOf<T>()`, `setOf<T>()`, `mutableListOf<T>()`, `mutableSetOf<T>()`. If you provide a comma-separated list of collection elements as arguments, the compiler detects the element type automatically. When creating empty collections, specify the type explicitly.

```
val numbersSet = setOf("one", "two", "three", "four")
val emptySet = mutableSetOf<String>()
```

The same is available for maps with the functions `mapOf()` and `mutableMapOf()`. The map's keys and values are passed as `Pair` objects (usually created with `to` infix function).

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
```

Note that the `to` notation creates a short-living `Pair` object, so it's recommended that you use it only if performance isn't critical. To avoid excessive memory usage, use alternative ways. For example, you can create a mutable map and populate it using the write operations. The `apply()` function can help to keep the initialization fluent here.

```
val numbersMap = mutableMapOf<String, String>().apply { this["one"] = "1"; this["two"] = "2" }
```

Create with collection builder functions

Another way of creating a collection is to call a builder function – `buildList()`, `buildSet()`, or `buildMap()`. They create a new, mutable collection of the corresponding type, populate it using write operations, and return a read-only collection with the same elements:

```
val map = buildMap { // this is MutableMap<String, Int>, types of key and value are inferred from the
`put()` calls below
    put("a", 1)
    put("b", 0)
    put("c", 4)
}

println(map) // {a=1, b=0, c=4}
```

Empty collections

There are also functions for creating collections without any elements: `emptyList()`, `emptySet()`, and `emptyMap()`. When creating empty collections, you should specify the type of elements that the collection will hold.

```
val empty = emptyList<String>()
```

Initializer functions for lists

For lists, there is a constructor-like function that takes the list size and the initializer function that defines the element value based on its index.

```
fun main() {
    val doubled = List(3, { it * 2 })  // or MutableList if you want to change its content later
    println(doubled)
}
```

Concrete type constructors

To create a concrete type collection, such as an `ArrayList` or `LinkedList`, you can use the available constructors for these types. Similar constructors are available for implementations of `Set` and `Map`.

```
val linkedList = LinkedList<String>(listOf("one", "two", "three"))
val presizedSet = HashSet<Int>(32)
```

Copy

To create a collection with the same elements as an existing collection, you can use copying functions. Collection copying functions from the standard library create *shallow* copy collections with references to the same elements. Thus, a change made to a collection element reflects in all its copies.

Collection copying functions, such as `toList()`, `toMutableList()`, `toSet()` and others, create a snapshot of a collection at a specific moment. Their result is a new collection of the same elements. If you add or remove elements from the original collection, this won't affect the copies. Copies may be changed independently of the source as well.

```
class Person(var name: String)
fun main() {
    val alice = Person("Alice")
    val sourceList = mutableListOf(alice, Person("Bob"))
    val copyList = sourceList.toList()
    sourceList.add(Person("Charles"))
    alice.name = "Alicia"
    println("First item's name is: ${sourceList[0].name} in source and ${copyList[0].name} in copy")
    println("List size is: ${sourceList.size} in source and ${copyList.size} in copy")
}
```

These functions can also be used for converting collections to other types, for example, build a set from a list or vice versa.

```
fun main() {
    val sourceList = mutableListOf(1, 2, 3)
    val copySet = sourceList.toMutableSet()
    copySet.add(3)
    copySet.add(4)
    println(copySet)
}
```

Alternatively, you can create new references to the same collection instance. New references are created when you initialize a collection variable with an existing collection. So, when the collection instance is altered through a reference, the changes are reflected in all its references.

```
fun main() {
    val sourceList = mutableListOf(1, 2, 3)
    val referenceList = sourceList
    referenceList.add(4)
    println("Source size: ${sourceList.size}")
}
```

Collection initialization can be used for restricting mutability. For example, if you create a `List` reference to a `MutableList`, the compiler will produce errors if you try to modify the collection through this reference.

```
fun main() {
//sampleStart
    val sourceList = mutableListOf(1, 2, 3)
    val referenceList: List<Int> = sourceList
    //referenceList.add(4)            //compilation error
    sourceList.add(4)
    println(referenceList) // shows the current state of sourceList
}
```

Invoke functions on other collections

Collections can be created as a result of various operations on other collections. For example, filtering a list creates a new list of elements that match the filter:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val longerThan3 = numbers.filter { it.length > 3 }
    println(longerThan3)
}
```

Mapping produces a list from a transformation's results:

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3)
    println(numbers.map { it * 3 })
    println(numbers.mapIndexed { idx, value -> value * idx })
}
```

Association produces maps:

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
}
```

For more information about operations on collections in Kotlin, see Collection operations overview.

Iterators

For traversing collection elements, the Kotlin standard library supports the commonly used mechanism of *iterators* – objects that provide access to the elements sequentially without exposing the underlying structure of the

collection. Iterators are useful when you need to process all the elements of a collection one-by-one, for example, print values or make similar updates to them.

Iterators can be obtained for inheritors of the `Iterable<T>` interface, including `Set` and `List`, by calling the `iterator()` function.

Once you obtain an iterator, it points to the first element of a collection; calling the `next()` function returns this element and moves the iterator position to the following element if it exists.

Once the iterator passes through the last element, it can no longer be used for retrieving elements; neither can it be reset to any previous position. To iterate through the collection again, create a new iterator.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    val numbersIterator = numbers.iterator()
    while (numbersIterator.hasNext()) {
        println(numbersIterator.next())
        // one
        // two
        // three
        // four
    }
}
```

Another way to go through an `Iterable` collection is the well-known `for` loop. When using `for` on a collection, you obtain the iterator implicitly. So, the following code is equivalent to the example above:

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    for (item in numbers) {
        println(item)
        // one
        // two
        // three
        // four
    }
}
```

Finally, there is a useful `forEach()` function that lets you automatically iterate a collection and execute the given code for each element. So, the same example would look like this:

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    numbers.forEach {
        println(it)
        // one
        // two
        // three
        // four
    }
}
```

List iterators

For lists, there is a special iterator implementation: `ListIterator`. It supports iterating lists in both directions: forwards and backwards.

Backward iteration is implemented by the functions `hasPrevious()` and `previous()`. Additionally, the `ListIterator` provides information about the element indices with the functions `nextIndex()` and `previousIndex()`.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    val listIterator = numbers.listIterator()
    while (listIterator.hasNext()) listIterator.next()
    println("Iterating backwards:")
    // Iterating backwards:
    while (listIterator.hasPrevious()) {
        print("Index: ${listIterator.previousIndex()}")
        println(", value: ${listIterator.previous()}")
        // Index: 3, value: four
        // Index: 2, value: three
        // Index: 1, value: two
        // Index: 0, value: one
    }
}
```

Having the ability to iterate in both directions, means the `ListIterator` can still be used after it reaches the last element.

Mutable iterators

For iterating mutable collections, there is `MutableIterator` that extends `Iterator` with the element removal function `remove()`. So, you can remove elements from a collection while iterating it.

```
fun main() {
    val numbers = mutableListOf("one", "two", "three", "four")
    val mutableIterator = numbers.iterator()

    mutableIterator.next()
    mutableIterator.remove()
    println("After removal: $numbers")
    // After removal: [two, three, four]
}
```

In addition to removing elements, the `MutableListIterator` can also insert and replace elements while iterating the list by using the `add()` and `set()` functions.

```
fun main() {
    val numbers = mutableListOf("one", "four", "four")
    val mutableListIterator = numbers.listIterator()

    mutableListIterator.next()
    mutableListIterator.add("two")
    println(numbers)
    // [one, two, four, four]
    mutableListIterator.next()
    mutableListIterator.set("three")
    println(numbers)
    // [one, two, three, four]
}
```

Ordering

The order of elements is an important aspect of certain collection types. For example, two lists of the same elements are not equal if their elements are ordered differently.

In Kotlin, the orders of objects can be defined in several ways.

First, there is *natural* order. It is defined for implementations of the `Comparable` interface. Natural order is used for sorting them when no other order is specified.

Most built-in types are comparable:

- Numeric types use the traditional numerical order: `1` is greater than `0`; `-3.4f` is greater than `-5f`, and so on.
- `Char` and `String` use the lexicographical order: `b` is greater than `a`; `world` is greater than `hello`.

To define a natural order for a user-defined type, make the type an implementer of `Comparable`. This requires implementing the `compareTo()` function. `compareTo()` must take another object of the same type as an argument and return an integer value showing which object is greater:

- Positive values show that the receiver object is greater.
- Negative values show that it's less than the argument.
- Zero shows that the objects are equal.

Below is a class for ordering versions that consist of the major and the minor part.

```
class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int = when {
        this.major != other.major -> this.major compareTo other.major // compareTo() in the infix form
        this.minor != other.minor -> this.minor compareTo other.minor
        else -> 0
    }
}

fun main() {
    println(Version(1, 2) > Version(1, 3))
    println(Version(2, 0) > Version(1, 5))
}
```

Custom orders let you sort instances of any type in a way you like. Particularly, you can define an order for non-comparable objects or define an order other than natural for a comparable type. To define a custom order for a type, create a `Comparator` for it. `Comparator` contains the `compare()` function: it takes two instances of a class and returns the integer result of the comparison between them. The result is interpreted in the same way as the result of a `compareTo()` as is described above.

```
fun main() {
    val lengthComparator = Comparator { str1: String, str2: String -> str1.length - str2.length }
    println(listOf("aaa", "bb", "c").sortedWith(lengthComparator))
}
```

Having the `lengthComparator`, you are able to arrange strings by their length instead of the default lexicographical order.

A shorter way to define a `Comparator` is the `compareBy()` function from the standard library. `compareBy()` takes a lambda function that produces a `Comparable` value from an instance and defines the custom order as the natural order of the produced values.

With `compareBy()`, the length comparator from the example above looks like this:

```
fun main() {
//sampleStart
    println(listOf("aaa", "bb", "c").sortedWith(compareBy { it.length }))
}
```

You can also define an order based on multiple criteria. For example, to sort strings by their length and alphabetically when the lengths are equal, you can write:

```
fun main() {
    val sortedStrings = listOf("aaa", "bb", "c", "b", "a", "aa", "ccc")
        .sortedWith { a, b ->
           when (val compareLengths = a.length.compareTo(b.length)) {
             0 -> a.compareTo(b)
             else -> compareLengths
           }
         }

    println(sortedStrings)
    // [a, b, c, aa, bb, aaa, ccc]
}
```

Since sorting by multiple criteria is a common scenario, the Kotlin standard library provides the `.thenBy()` function that you can use to add a secondary sorting rule.

For example, you can combine `compareBy()` with `.thenBy()` to sort strings by their length first and alphabetically second, just like in the previous example:

```
fun main() {
    val sortedStrings = listOf("aaa", "bb", "c", "b", "a", "aa", "ccc")
        .sortedWith(compareBy<String> { it.length }.thenBy { it })

    println(sortedStrings)
    // [a, b, c, aa, bb, aaa, ccc]
}
```

The Kotlin collections package provides functions for sorting collections in natural, custom, and even random orders. On this page, we'll describe sorting functions that apply to read-only collections. These functions return their result as a new collection containing the elements of the original collection in the requested order. To learn about functions for sorting mutable collections in place, see the List-specific operations.

Natural order

The basic functions `.sorted()` and `.sortedDescending()` return elements of a collection sorted into ascending and descending sequence according to their natural order. These functions apply to collections of `Comparable` elements.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")

    println("Sorted ascending: ${numbers.sorted()}")
    println("Sorted descending: ${numbers.sortedDescending()}")
}
```

Custom orders

For sorting in custom orders or sorting non-comparable objects, there are the functions `.sortedBy()` and `.sortedByDescending()`. They take a selector function that maps collection elements to `Comparable` values and sort the collection in natural order of that values.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")

    val sortedNumbers = numbers.sortedBy { it.length }
    println("Sorted by length ascending: $sortedNumbers")
    val sortedByLast = numbers.sortedByDescending { it.last() }
    println("Sorted by the last letter descending: $sortedByLast")
}
```

To define a custom order for the collection sorting, you can provide your own `Comparator`. To do this, call the `.sortedWith()` extension function passing in your `Comparator`. With this function, sorting strings by their length looks like this:

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    println("Sorted by length ascending: ${numbers.sortedWith(compareBy { it.length })}")
}
```

Check sorted order

You can use the following extension functions to check whether elements already follow a specified order:

- `.isSorted()`
- `.isSortedDescending()`
- `.isSortedWith(comparator)`
- `.isSortedBy(selector)`
- `.isSortedByDescending(selector)`

These extension functions return `true` if the elements are in the specified order or if there are fewer than two elements. They return `false` and stop checking as soon as they find an out-of-order pair.

For collections without a guaranteed iteration order, such as `HashSet`, the result may vary across calls. The same applies to sequences that don't produce elements in a consistent order. To get the same result across calls, use these functions only on collections with a guaranteed iteration order, such as `List`.

When checking `Double` and `Float` values, these functions treat `NaN` as greater than any other value and `-0.0` as less than `0.0`. Additionally, the `.isSortedBy()` and `.isSortedByDescending()` functions treat `null` selector results as less than any non-null value.

When you call these functions on a sequence, the operation is terminal. It consumes the sequence to produce a `Boolean` value instead of returning another sequence.

> These sorted-order functions are also available for arrays, primitive arrays, and unsigned arrays. Unsigned arrays and operations on them are Experimental and require opt-in with the `@ExperimentalUnsignedTypes` annotation.

Here's an example of checking sorted order with the `.isSorted()` and `.isSortedBy()` functions:

```
data class User(val name: String, val age: Int)

fun main() {
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.isSorted())
    // true

    val users = listOf(
        User("Alice", 24),
        User("Bob", 31),
        User("Charlie", 29),
    )
    println(users.isSortedBy(User::age))
    // false

    val descending = listOf(4, 3, 2, 1)
    println(descending.isSortedDescending())
    // true

}
```

Reverse order

You can retrieve the collection in the reversed order using the `.reversed()` function.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.reversed())
}
```

The `.reversed()` extension function returns a new collection with the copies of the elements. So, if you change the original collection later, this won't affect the previously obtained results of `.reversed()`.

Another reversing function - `.asReversed()`

- returns a reversed view of the same collection instance, so it may be more lightweight and preferable than `.reversed()` if the original list is not going to change.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    val reversedNumbers = numbers.asReversed()
    println(reversedNumbers)
}
```

If the original list is mutable, all its changes reflect in its reversed views and vice versa.

```
fun main() {
    val numbers = mutableListOf("one", "two", "three", "four")
    val reversedNumbers = numbers.asReversed()
    println(reversedNumbers)
    numbers.add("five")
    println(reversedNumbers)
}
```

However, if the mutability of the list is unknown or the source is not a list at all, `.reversed()` is more preferable since its result is a copy that won't change in the future.

Random order

Finally, there is a function that returns a new `List` containing the collection elements in a random order - `.shuffled()`. You can call it without arguments or with a `Random` object.

```
fun main() {
     val numbers = listOf("one", "two", "three", "four")
     println(numbers.shuffled())
}
```

Grouping

The Kotlin standard library provides extension functions for grouping collection elements. The basic function `groupBy()` takes a lambda function and returns a `Map`. In this map, each key is the lambda result, and the corresponding value is the `List` of elements on which this result is returned. This function can be used, for example, to group a list of `String`s by their first letter.

You can also call `groupBy()` with a second lambda argument – a value transformation function. In the result map of `groupBy()` with two lambdas, the keys produced by `keySelector` function are mapped to the results of the value transformation function instead of the original elements.

This example illustrates using the `groupBy()` function to group the strings by their first letter, iterating through the groups on the resulting `Map` with the `for` operator, and then transforming the values to uppercase using the `valueTransform` function:

```
fun main() {
    val numbers = listOf("one", "two", "three", "four", "five")

    // Groups the strings by their first letter using groupBy()
    val groupedByFirstLetter = numbers.groupBy { it.first().uppercase() }
    println(groupedByFirstLetter)
    // {O=[one], T=[two, three], F=[four, five]}

    // Iterates through each group and prints the key and its associated values
    for ((key, value) in groupedByFirstLetter) {
        println("Key: $key, Values: $value")
    }
    // Key: O, Values: [one]
    // Key: T, Values: [two, three]
    // Key: F, Values: [four, five]

    // Groups the strings by their first letter and transforms the values to uppercase
    val groupedAndTransformed = numbers.groupBy(keySelector = { it.first() }, valueTransform = {
it.uppercase() })
    println(groupedAndTransformed)
    // {o=[ONE], t=[TWO, THREE], f=[FOUR, FIVE]}
}
```

If you want to group elements and then apply an operation to all groups at one time, use the function `groupingBy()`. It returns an instance of the `Grouping` type. The `Grouping` instance lets you apply operations to all groups in a lazy manner: the groups are actually built right before the operation execution.

Namely, `Grouping` supports the following operations:

- `eachCount()` counts the elements in each group.

- `fold()` and `reduce()` perform fold and reduce operations on each group as a separate collection and return the results.
- `aggregate()` applies a given operation subsequently to all the elements in each group and returns the result. This is the generic way to perform any operations on a `Grouping`. Use it to implement custom operations when fold or reduce are not enough.

You can use the `for` operator on the resulting `Map` to iterate through the groups created by the `groupingBy()` function. This allows you to access each key and the count of elements associated with that key.

The following example demonstrates how to group strings by their first letter using the `groupingBy()` function, count the elements in each group, and then iterate through each group to print the key and count of elements:

```
fun main() {
    val numbers = listOf("one", "two", "three", "four", "five")

    // Groups the strings by their first letter using groupingBy() and counts the elements in each
group
    val grouped = numbers.groupingBy { it.first() }.eachCount()

    // Iterates through each group and prints the key and its associated values
    for ((key, count) in grouped) {
        println("Key: $key, Count: $count")
        // Key: o, Count: 1
        // Key: t, Count: 2
        // Key: f, Count: 2
    }
}
```

Collection transformation operations

The Kotlin standard library provides a set of extension functions for collection *transformations*. These functions build new collections from existing ones based on the transformation rules provided. In this page, we'll give an overview of the available collection transformation functions.

Map

The *mapping* transformation creates a collection from the results of a function on the elements of another collection. The basic mapping function is `map()`. It applies the given lambda function to each subsequent element and returns the list of the lambda results. The order of results is the same as the original order of elements. To apply a transformation that additionally uses the element index as an argument, use `mapIndexed()`.

```
fun main() {
    val numbers = setOf(1, 2, 3)
    println(numbers.map { it * 3 })
    println(numbers.mapIndexed { idx, value -> value * idx })
}
```

If the transformation produces `null` on certain elements, you can filter out the `null`s from the result collection by calling the `mapNotNull()` function instead of `map()`, or `mapIndexedNotNull()` instead of `mapIndexed()`.

```
fun main() {
    val numbers = setOf(1, 2, 3)
    println(numbers.mapNotNull { if ( it == 2) null else it * 3 })
    println(numbers.mapIndexedNotNull { idx, value -> if (idx == 0) null else value * idx })
}
```

When transforming maps, you have two options: transform keys leaving values unchanged and vice versa. To apply a given transformation to keys, use `mapKeys()`; in turn, `mapValues()` transforms values. Both functions use the transformations that take a map entry as an argument, so you can operate both its key and value.

```
fun main() {
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    println(numbersMap.mapKeys { it.key.uppercase() })
    println(numbersMap.mapValues { it.value + it.key.length })
}
```

Zip

Zipping transformation is building pairs from elements with the same positions in both collections. In the Kotlin standard library, this is done by the `zip()` extension function.

When called on a collection or an array with another collection (or array) as an argument, `zip()` returns the `List` of `Pair` objects. The elements of the receiver collection are the first elements in these pairs.

If the collections have different sizes, the result of the `zip()` is the smaller size; the last elements of the larger collection are not included in the result.

`zip()` can also be called in the infix form `a zip b`.

```
fun main() {
    val colors = listOf("red", "brown", "grey")
    val animals = listOf("fox", "bear", "wolf")
    println(colors zip animals)

    val twoAnimals = listOf("fox", "bear")
    println(colors.zip(twoAnimals))
}
```

You can also call `zip()` with a transformation function that takes two parameters: the receiver element and the argument element. In this case, the result `List` contains the return values of the transformation function called on pairs of the receiver and the argument elements with the same positions.

```
fun main() {
    val colors = listOf("red", "brown", "grey")
    val animals = listOf("fox", "bear", "wolf")

    println(colors.zip(animals) { color, animal -> "The ${animal.replaceFirstChar { it.uppercase() }}
is $color"})
}
```

When you have a `List` of `Pair`s, you can do the reverse transformation – *unzipping* – that builds two lists from these pairs:

- The first list contains the first elements of each `Pair` in the original list.
- The second list contains the second elements.

To unzip a list of pairs, call `unzip()`.

```
fun main() {
    val numberPairs = listOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)
    println(numberPairs.unzip())
}
```

Associate

Association transformations allow building maps from the collection elements and certain values associated with them. In different association types, the elements can be either keys or values in the association map.

The basic association function `associateWith()` creates a `Map` in which the elements of the original collection are keys, and values are produced from them by the given transformation function. If two elements are equal, only the last one remains in the map.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
}
```

For building maps with collection elements as values, there is the function `associateBy()`. It takes a function that returns a key based on an element's value. If two elements' keys are equal, only the last one remains in the map.

`associateBy()` can also be called with a value transformation function.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")

    println(numbers.associateBy { it.first().uppercaseChar() })
    println(numbers.associateBy(keySelector = { it.first().uppercaseChar() }, valueTransform = {
it.length }))
}
```

Another way to build maps in which both keys and values are somehow produced from collection elements is the function `associate()`. It takes a lambda function that returns a `Pair`: the key and the value of the corresponding map entry.

Note that `associate()` produces short-living `Pair` objects which may affect the performance. Thus, `associate()` should be used when the performance isn't critical or it's more preferable than other options.

An example of the latter is when a key and the corresponding value are produced from an element together.

```
fun main() {
data class FullName (val firstName: String, val lastName: String)

fun parseFullName(fullName: String): FullName {
    val nameParts = fullName.split(" ")
    if (nameParts.size == 2) {
        return FullName(nameParts[0], nameParts[1])
    } else throw Exception("Wrong name format")
}

    val names = listOf("Alice Adams", "Brian Brown", "Clara Campbell")
    println(names.associate { name -> parseFullName(name).let { it.lastName to it.firstName } })
}
```

Here we call a transform function on an element first, and then build a pair from the properties of that function's result.

Flatten

If you operate nested collections, you may find the standard library functions that provide flat access to nested collection elements useful.

The first function is `flatten()`. You can call it on a collection of collections, for example, a `List` of `Set`s. The function returns a single `List` of all the elements of the nested collections.

```
fun main() {
    val numberSets = listOf(setOf(1, 2, 3), setOf(4, 5, 6), setOf(1, 2))
    println(numberSets.flatten())
}
```

Another function – `flatMap()` provides a flexible way to process nested collections. It takes a function that maps a collection element to another collection. As a result, `flatMap()` returns a single list of its return values on all the elements. So, `flatMap()` behaves as a subsequent call of `map()` (with a collection as a mapping result) and `flatten()`.

```
data class StringContainer(val values: List<String>)

fun main() {
    val containers = listOf(
        StringContainer(listOf("one", "two", "three")),
        StringContainer(listOf("four", "five", "six")),
        StringContainer(listOf("seven", "eight"))
    )
    println(containers.flatMap { it.values })
}
```

String representation

If you need to retrieve the collection content in a readable format, use functions that transform the collections to strings: `joinToString()` and `joinTo()`.

`joinToString()` builds a single `String` from the collection elements based on the provided arguments. `joinTo()` does the same but appends the result to the given `Appendable` object.

When called with the parameters' default values, the functions return the result similar to calling `toString()` on the collection: a `String` of elements' string representations separated by commas with spaces.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")

    println(numbers)
    println(numbers.joinToString())

    val listString = StringBuffer("The list of numbers: ")
    numbers.joinTo(listString)
    println(listString)
}
```

To build a custom string representation, you can specify its parameters in function arguments `separator`, `prefix`, and `postfix`. The resulting string will start with the `prefix` and end with the `postfix`. The `separator` will come after each element except the last.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.joinToString(separator = " | ", prefix = "start: ", postfix = ": end"))
}
```

For bigger collections, you may want to specify the `limit` – a number of elements that will be included into result. If the collection size exceeds the `limit`, all the other elements will be replaced with a single value of the `truncated` argument.

```
fun main() {
    val numbers = (1..100).toList()
    println(numbers.joinToString(limit = 10, truncated = "<...>"))
}
```

Finally, to customize the representation of elements themselves, provide the `transform` function.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.joinToString { "Element: ${it.uppercase()}"})
}
```

Filtering collections

Filtering is one of the most popular tasks in collection processing. In Kotlin, filtering conditions are defined by *predicates* – lambda functions that take a collection element and return a boolean value: `true` means that the given element matches the predicate, `false` means the opposite.

The standard library contains a group of extension functions that let you filter collections in a single call. These functions leave the original collection unchanged, so they are available for both mutable and read-only collections. To operate the filtering result, you should assign it to a variable or chain the functions after filtering.

Filter by predicate

The basic filtering function is `filter()`. When called with a predicate, `filter()` returns the collection elements that match it. For both `List` and `Set`, the resulting collection is a `List`, for `Map` it's a `Map` as well.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    val longerThan3 = numbers.filter { it.length > 3 }
    println(longerThan3)

    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
    println(filteredMap)
}
```

The predicates in `filter()` can only check the values of the elements. If you want to use element positions in the filter, use `filterIndexed()`. It takes a predicate with two arguments: the index and the value of an element.

To filter collections by negative conditions, use `filterNot()`. It returns a list of elements for which the predicate yields `false`.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")

    val filteredIdx = numbers.filterIndexed { index, s -> (index != 0) && (s.length < 5)  }
    val filteredNot = numbers.filterNot { it.length <= 3 }

    println(filteredIdx)
    println(filteredNot)
}
```

There are also functions that narrow the element type by filtering elements of a given type:

- `filterIsInstance()` returns collection elements of a given type. Being called on a `List<Any>`, `filterIsInstance<T>()` returns a `List<T>`, thus allowing you to call functions of the `T` type on its items.

```
fun main() {
    val numbers = listOf(null, 1, "two", 3.0, "four")
    println("All String elements in upper case:")
    numbers.filterIsInstance<String>().forEach {
        println(it.uppercase())
    }
}
```

- `filterNotNull()` returns all non-nullable elements. Being called on a `List<T?>`, `filterNotNull()` returns a `List<T: Any>`, thus allowing you to treat the elements as non-nullable objects.

```
fun main() {
    val numbers = listOf(null, "one", "two", null)
    numbers.filterNotNull().forEach {
        println(it.length)   // length is unavailable for nullable Strings
    }
}
```

Partition

Another filtering function – `partition()` – filters a collection by a predicate and keeps the elements that don't match it in a separate list. So, you have a `Pair` of `List`s as a return value: the first list containing elements that match the predicate and the second one containing everything else from the original collection.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    val (match, rest) = numbers.partition { it.length > 3 }

    println(match)
    println(rest)
}
```

Test predicates

Finally, there are functions that simply test a predicate against collection elements:

- `any()` returns `true` if at least one element matches the given predicate.
- `none()` returns `true` if none of the elements match the given predicate.

- `all()` returns `true` if all elements match the given predicate. Note that `all()` returns `true` when called with any valid predicate on an empty collection. Such behavior is known in logic as *vacuous truth*.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")

    println(numbers.any { it.endsWith("e") })
    println(numbers.none { it.endsWith("a") })
    println(numbers.all { it.endsWith("e") })

    println(emptyList<Int>().all { it > 5 })   // vacuous truth
}
```

`any()` and `none()` can also be used without a predicate: in this case, they just check the collection emptiness. `any()` returns `true` if there are elements and `false` if there aren't; `none()` does the opposite.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    val empty = emptyList<String>()

    println(numbers.any())
    println(empty.any())

    println(numbers.none())
    println(empty.none())
}
```

Retrieve collection parts

The Kotlin standard library contains extension functions for retrieving parts of a collection. These functions provide a variety of ways to select elements for the result collection: listing their positions explicitly, specifying the result size, and others.

Slice

`slice()` returns a list of the collection elements with given indices. The indices may be passed either as a range or as a collection of integer values.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.slice(1..3))
    println(numbers.slice(0..4 step 2))
    println(numbers.slice(setOf(3, 5, 0)))
}
```

Take and drop

To get the specified number of elements starting from the first, use the `take()` function. For getting the last elements, use `takeLast()` . When called with a number larger than the collection size, both functions return the whole collection.

To take all the elements except a given number of first or last elements, call the `drop()` and `dropLast()` functions respectively.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.take(3))
    println(numbers.takeLast(3))
    println(numbers.drop(1))
    println(numbers.dropLast(5))
}
```

You can also use predicates to define the number of elements for taking or dropping. There are four functions similar to the ones described above:

- `takeWhile()` is `take()` with a predicate: it takes the elements up to but excluding the first one not matching the predicate. If the first collection element doesn't match the predicate, the result is empty.
- `takeLastWhile()` is similar to `takeLast()`: it takes the range of elements matching the predicate from the end of the collection. The first element of the range is the element next to the last element not matching the predicate. If the last collection element doesn't match the predicate, the result is empty;
- `dropWhile()` is the opposite to `takeWhile()` with the same predicate: it returns the elements from the first one not matching the predicate to the end.
- `dropLastWhile()` is the opposite to `takeLastWhile()` with the same predicate: it returns the elements from the beginning to the last one not matching the predicate.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.takeWhile { !it.startsWith('f') })
    println(numbers.takeLastWhile { it != "three" })
    println(numbers.dropWhile { it.length == 3 })
    println(numbers.dropLastWhile { it.contains('i') })
}
```

Chunked

To break a collection into parts of a given size, use the `chunked()` function. `chunked()` takes a single argument – the size of the chunk – and returns a `List` of `List`s of the given size. The first chunk starts from the first element and contains the `size` elements, the second chunk holds the next `size` elements, and so on. The last chunk may have a smaller size.

```
fun main() {
    val numbers = (0..13).toList()
    println(numbers.chunked(3))
}
```

You can also apply a transformation for the returned chunks right away. To do this, provide the transformation as a lambda function when calling `chunked()`. The lambda argument is a chunk of the collection. When `chunked()` is called with a transformation, the chunks are short-living `List`s that should be consumed right in that lambda.

```
fun main() {
    val numbers = (0..13).toList()
    println(numbers.chunked(3) { it.sum() })  // `it` is a chunk of the original collection
}
```

Windowed

You can retrieve all possible ranges of the collection elements of a given size. The function for getting them is called `windowed()`: it returns a list of element ranges that you would see if you were looking at the collection through a sliding window of the given size. Unlike `chunked()`, `windowed()` returns element ranges (*windows*) starting from *each* collection element. All the windows are returned as elements of a single `List`.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.windowed(3))
}
```

`windowed()` provides more flexibility regarding parameters with default values:

- `step` defines a distance between first elements of two adjacent windows. By default, the value is 1, so the result contains windows starting from all elements. If you increase the step to 2, you will receive only windows starting from odd elements: first, third, and so on.
- `partialWindows` includes windows of smaller sizes that start from the elements at the end of the collection. For example, if you request windows of three elements, you can't build them for the last two elements. Enabling `partialWindows` in this case includes two more lists of sizes 2 and 1.

Finally, you can apply a transformation to the returned ranges right away. To do this, provide the transformation as a lambda function when calling `windowed()`.

```
fun main() {
    val numbers = (1..10).toList()
    println(numbers.windowed(3, step = 2, partialWindows = true))
    println(numbers.windowed(3) { it.sum() })
}
```

To build two-element windows, there is a separate function - `zipWithNext()`. It creates pairs of adjacent elements of the receiver collection. Note that `zipWithNext()` doesn't break the collection into pairs; it creates a `Pair` for *each* element except the last one, so its result on `[1, 2, 3, 4]` is `[[1, 2], [2, 3], [3, 4]]`, not `[[1, 2], [3, 4]]`. `zipWithNext()` can be called with a transformation function as well; it should take two elements of the receiver collection as arguments.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.zipWithNext())
    println(numbers.zipWithNext() { s1, s2 -> s1.length > s2.length})
}
```

Retrieve single elements

Kotlin collections provide a set of functions for retrieving single elements from collections. Functions described on this page apply to both lists and sets.

As the definition of list says, a list is an ordered collection. Hence, every element of a list has its position that you can use for referring. In addition to functions described on this page, lists offer a wider set of ways to retrieve and search for elements by indices. For more details, see List-specific operations.

In turn, set is not an ordered collection by definition. However, the Kotlin `Set` stores elements in certain orders. These can be the order of insertion (in `LinkedHashSet`), natural sorting order (in `SortedSet`), or another order.

The order of a set of elements can also be unknown. In such cases, the elements are still ordered somehow, so the functions that rely on the element positions still return their results. However, such results are unpredictable to the caller unless they know the specific implementation of `Set` used.

Retrieve by position

For retrieving an element at a specific position, there is the function `elementAt()`. Call it with the integer number as an argument, and you'll receive the collection element at the given position. The first element has the position `0`, and the last one is `(size - 1)`.

`elementAt()` is useful for collections that do not provide indexed access, or are not statically known to provide one. In case of `List`, it's more idiomatic to use indexed access operator (`get()` or `[]`).

```
fun main() {
    val numbers = linkedSetOf("one", "two", "three", "four", "five")
    println(numbers.elementAt(3))

    val numbersSortedSet = sortedSetOf("one", "two", "three", "four")
    println(numbersSortedSet.elementAt(0)) // elements are stored in the ascending order
}
```

There are also useful aliases for retrieving the first and the last element of the collection: `first()` and `last()`.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.first())
    println(numbers.last())
}
```

To avoid exceptions when retrieving element with non-existing positions, use safe variations of `elementAt()`:

- `elementAtOrNull()` returns null when the specified position is out of the collection bounds.
- `elementAtOrElse()` additionally takes a lambda function that maps an `Int` argument to an instance of the collection element type. When called with an out-of-bounds position, the `elementAtOrElse()` returns the result of the lambda on the given value.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.elementAtOrNull(5))
    println(numbers.elementAtOrElse(5) { index -> "The value for index $index is undefined"})
}
```

Retrieve by condition

Functions `first()` and `last()` also let you search a collection for elements matching a given predicate. When you call `first()` with a predicate that tests a collection element, you'll receive the first element on which the predicate yields `true`. In turn, `last()` with a predicate returns the last element matching it.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.first { it.length > 3 })
    println(numbers.last { it.startsWith("f") })
}
```

If no elements match the predicate, both functions throw exceptions. To avoid them, use `firstOrNull()` and `lastOrNull()` instead: they return `null` if no matching elements are found.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.firstOrNull { it.length > 6 })
}
```

Use the aliases if their names suit your situation better:

- `find()` instead of `firstOrNull()`
- `findLast()` instead of `lastOrNull()`

```
fun main() {
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.find { it % 2 == 0 })
    println(numbers.findLast { it % 2 == 0 })
}
```

Retrieve with selector

If you need to map the collection before retrieving the element, there is a function `firstNotNullOf()`. It combines 2 actions:

- Maps the collection with the selector function
- Returns the first non-null value in the result

`firstNotNullOf()` throws the `NoSuchElementException` if the resulting collection doesn't have a non-nullable element. Use the counterpart `firstNotNullOfOrNull()` to return null in this case.

```
fun main() {
    val list = listOf<Any>(0, "true", false)
    // Converts each element to string and returns the first one that has required length
    val longEnough = list.firstNotNullOf { item -> item.toString().takeIf { it.length >= 4 } }
    println(longEnough)
}
```

Random element

If you need to retrieve an arbitrary element of a collection, call the `random()` function. You can call it without arguments or with a `Random` object as a source of the randomness.

```
fun main() {
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.random())
}
```

On empty collections, `random()` throws an exception. To receive `null` instead, use `randomOrNull()`

Check element existence

To check the presence of an element in a collection, use the `contains()` function. It returns `true` if there is a collection element that `equals()` the function argument. You can call `contains()` in the operator form with the `in` keyword.

To check the presence of multiple instances together at once, call `containsAll()` with a collection of these instances as an argument.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.contains("four"))
    println("zero" in numbers)

    println(numbers.containsAll(listOf("four", "two")))
    println(numbers.containsAll(listOf("one", "zero")))
}
```

Additionally, you can check if the collection contains any elements by calling `isEmpty()` or `isNotEmpty()`.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.isEmpty())
    println(numbers.isNotEmpty())

    val empty = emptyList<String>()
    println(empty.isEmpty())
    println(empty.isNotEmpty())
}
```

Collection operations overview

The Kotlin standard library offers a broad variety of functions for performing operations on collections. This includes simple operations, such as getting or adding elements, as well as more complex ones including search, sorting, filtering, transformations, and so on.

Extension and member functions

Collection operations are declared in the standard library in two ways: member functions of collection interfaces and extension functions.

Member functions define operations that are essential for a collection type. For example, `Collection` contains the function `isEmpty()` for checking its emptiness; `List` contains `get()` for index access to elements, and so on.

When you create your own implementations of collection interfaces, you must implement their member functions. To make the creation of new implementations easier, use the skeletal implementations of collection interfaces from the standard library: `AbstractCollection`, `AbstractList`, `AbstractSet`, `AbstractMap`, and their mutable counterparts.

Other collection operations are declared as extension functions. These are filtering, transformation, ordering, and other collection processing functions.

Common operations

Common operations are available for both read-only and mutable collections. Common operations fall into these groups:

- Transformations
- Filtering
- `plus` and `minus` operators
- Grouping
- Retrieving collection parts
- Retrieving single elements
- Ordering
- Aggregate operations

Operations described on these pages return their results without affecting the original collection. For example, a filtering operation produces a *new collection* that contains all the elements matching the filtering predicate. Results of such operations should be either stored in variables, or used in some other way, for example, passed in other functions.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    numbers.filter { it.length > 3 }  // nothing happens with `numbers`, result is lost
    println("numbers are still $numbers")
    val longerThan3 = numbers.filter { it.length > 3 } // result is stored in `longerThan3`
    println("numbers longer than 3 chars are $longerThan3")
}
```

For certain collection operations, there is an option to specify the *destination* object. Destination is a mutable collection to which the function appends its resulting items instead of returning them in a new object. For performing operations with destinations, there are separate functions with the `To` postfix in their names, for example, `filterTo()` instead of `filter()` or `associateTo()` instead of `associate()`. These functions take the destination collection as an additional parameter.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    val filterResults = mutableListOf<String>()  //destination object
    numbers.filterTo(filterResults) { it.length > 3 }
    numbers.filterIndexedTo(filterResults) { index, _ -> index == 0 }
    println(filterResults) // contains results of both operations
}
```

For convenience, these functions return the destination collection back, so you can create it right in the corresponding argument of the function call:

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    // filter numbers right into a new hash set,
    // thus eliminating duplicates in the result
    val result = numbers.mapTo(HashSet()) { it.length }
    println("distinct item lengths are $result")
}
```

Functions with destination are available for filtering, association, grouping, flattening, and other operations. For the complete list of destination operations see the Kotlin collections reference.

Write operations

For mutable collections, there are also *write operations* that change the collection state. Such operations include adding, removing, and updating elements. Write operations are listed in the Write operations and corresponding sections of List-specific operations and Map specific operations.

For certain operations, there are pairs of functions for performing the same operation: one applies the operation in-place and the other returns the result as a separate collection. For example, `sort()` sorts a mutable collection in-place, so its state changes; `sorted()` creates a new collection that contains the same elements in the sorted order.

```
fun main() {
    val numbers = mutableListOf("one", "two", "three", "four")
    val sortedNumbers = numbers.sorted()
    println(numbers == sortedNumbers)  // false
    numbers.sort()
    println(numbers == sortedNumbers)  // true
}
```

Aggregate operations

Kotlin collections contain functions for commonly used *aggregate operations* – operations that return a single value based on the collection content. Most of them are well known and work the same way as they do in other languages:

- `minOrNull()` and `maxOrNull()` return the smallest and the largest element respectively. On empty collections, they return `null`.
- `average()` returns the average value of elements in the collection of numbers.
- `sum()` returns the sum of elements in the collection of numbers.
- `count()` returns the number of elements in a collection.

```
fun main() {
    val numbers = listOf(6, 42, 10, 4)

    println("Count: ${numbers.count()}")
    println("Max: ${numbers.maxOrNull()}")
    println("Min: ${numbers.minOrNull()}")
    println("Average: ${numbers.average()}")
    println("Sum: ${numbers.sum()}")
}
```

There are also functions for retrieving the smallest and the largest elements by certain selector function or custom `Comparator`:

- `maxByOrNull()` and `minByOrNull()` take a selector function and return the element for which it returns the largest or the smallest value.
- `maxWithOrNull()` and `minWithOrNull()` take a `Comparator` object and return the largest or smallest element according to that `Comparator`.

- `maxOfOrNull()` and `minOfOrNull()` take a selector function and return the largest or the smallest return value of the selector itself.
- `maxOfWithOrNull()` and `minOfWithOrNull()` take a `Comparator` object and return the largest or smallest selector return value according to that `Comparator`.

These functions return `null` on empty collections. There are also alternatives – `maxOf`, `minOf`, `maxOfWith`, and `minOfWith` – which do the same as their counterparts but throw a `NoSuchElementException` on empty collections.

```
fun main() {
    val numbers = listOf(5, 42, 10, 4)
    val min3Remainder = numbers.minByOrNull { it % 3 }
    println(min3Remainder)

    val strings = listOf("one", "two", "three", "four")
    val longestString = strings.maxWithOrNull(compareBy { it.length })
    println(longestString)
}
```

Besides regular `sum()`, there is an advanced summation function `sumOf()` that takes a selector function and returns the sum of its application to all collection elements. Selector can return different numeric types: `Int`, `Long`, `Double`, `UInt`, and `ULong` (also `BigInteger` and `BigDecimal` on the JVM).

```
fun main() {
    val numbers = listOf(5, 42, 10, 4)
    println(numbers.sumOf { it * 2 })
    println(numbers.sumOf { it.toDouble() / 2 })
}
```

Fold and reduce

For more specific cases, there are the functions `reduce()` and `fold()` that apply the provided operation to the collection elements sequentially and return the accumulated result. The operation takes two arguments: the previously accumulated value and the collection element.

The difference between the two functions is that `fold()` takes an initial value and uses it as the accumulated value on the first step, whereas the first step of `reduce()` uses the first and the second elements as operation arguments on the first step.

```
fun main() {
    val numbers = listOf(5, 2, 10, 4)

    val simpleSum = numbers.reduce { sum, element -> sum + element }
    println(simpleSum)
    val sumDoubled = numbers.fold(0) { sum, element -> sum + element * 2 }
    println(sumDoubled)

    //incorrect: the first element isn't doubled in the result
    //val sumDoubledReduce = numbers.reduce { sum, element -> sum + element * 2 }
    //println(sumDoubledReduce)
}
```

The example above shows the difference: `fold()` is used for calculating the sum of doubled elements. If you pass the same function to `reduce()`, it will return another result because it uses the list's first and second elements as arguments on the first step, so the first element won't be doubled.

To apply a function to elements in the reverse order, use functions `reduceRight()` and `foldRight()`. They work in a way similar to `fold()` and `reduce()` but start from the last element and then continue to previous. Note that when folding or reducing right, the operation arguments change their order: first goes the element, and then the accumulated value.

```
fun main() {
    val numbers = listOf(5, 2, 10, 4)
    val sumDoubledRight = numbers.foldRight(0) { element, sum -> sum + element * 2 }
    println(sumDoubledRight)
}
```

You can also apply operations that take element indices as parameters. For this purpose, use functions `reduceIndexed()` and `foldIndexed()` passing element index as the first argument of the operation.

Finally, there are functions that apply such operations to collection elements from right to left - `reduceRightIndexed()` and `foldRightIndexed()`.

```
fun main() {
    val numbers = listOf(5, 2, 10, 4)
    val sumEven = numbers.foldIndexed(0) { idx, sum, element -> if (idx % 2 == 0) sum + element else
sum }
    println(sumEven)

    val sumEvenRight = numbers.foldRightIndexed(0) { idx, element, sum -> if (idx % 2 == 0) sum +
element else sum }
    println(sumEvenRight)
}
```

All reduce operations throw an exception on empty collections. To receive `null` instead, use their `*OrNull()` counterparts:

- `reduceOrNull()`
- `reduceRightOrNull()`
- `reduceIndexedOrNull()`
- `reduceRightIndexedOrNull()`

For cases where you want to save intermediate accumulator values, there are functions `runningFold()` (or its synonym `scan()`) and `runningReduce()`.

```
fun main() {
    val numbers = listOf(0, 1, 2, 3, 4, 5)
    val runningReduceSum = numbers.runningReduce { sum, item -> sum + item }
    val runningFoldSum = numbers.runningFold(10) { sum, item -> sum + item }
    val transform = { index: Int, element: Int -> "N = ${index + 1}: $element" }
    println(runningReduceSum.mapIndexed(transform).joinToString("\n", "Sum of first N elements with
runningReduce:\n"))
    println(runningFoldSum.mapIndexed(transform).joinToString("\n", "Sum of first N elements with
runningFold:\n"))
}
```

If you need an index in the operation parameter, use `runningFoldIndexed()` or `runningReduceIndexed()`.

Collection write operations

Mutable collections support operations for changing the collection contents, for example, adding or removing elements. On this page, we'll describe write operations available for all implementations of `MutableCollection`. For more specific operations available for `List` and `Map`, see List-specific Operations and Map Specific Operations respectively.

Adding elements

To add a single element to a list or a set, use the `add()` function. The specified object is appended to the end of the collection.

```
fun main() {
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.add(5)
    println(numbers)
}
```

`addAll()` adds every element of the argument object to a list or a set. The argument can be an `Iterable`, a `Sequence`, or an `Array`. The types of the receiver and the argument may be different, for example, you can add all items from a `Set` to a `List`.

When called on lists, `addAll()` adds new elements in the same order as they go in the argument. You can also call `addAll()` specifying an element position as the first argument. The first element of the argument collection will be inserted at this position. Other elements of the argument collection will follow it, shifting the receiver elements to the end.

```
fun main() {
    val numbers = mutableListOf(1, 2, 5, 6)
    numbers.addAll(arrayOf(7, 8))
    println(numbers)
    numbers.addAll(2, setOf(3, 4))
    println(numbers)
}
```

You can also add elements using the in-place version of the `plus` operator - `plusAssign` (`+=`) When applied to a mutable collection, `+=` appends the second operand (an element or another collection) to the end of the collection.

```
fun main() {
    val numbers = mutableListOf("one", "two")
    numbers += "three"
    println(numbers)
    numbers += listOf("four", "five")
    println(numbers)
}
```

Removing elements

To remove an element from a mutable collection, use the `remove()` function. `remove()` accepts the element value and removes one occurrence of this value.

```
fun main() {
    val numbers = mutableListOf(1, 2, 3, 4, 3)
    numbers.remove(3)                    // removes the first `3`
    println(numbers)
    numbers.remove(5)                    // removes nothing
    println(numbers)
}
```

For removing multiple elements at once, there are the following functions :

- `removeAll()` removes all elements that are present in the argument collection. Alternatively, you can call it with a predicate as an argument; in this case, the function removes all elements for which the predicate yields `true` .
- `retainAll()` is the opposite of `removeAll()` : it removes all elements except the ones from the argument collection. When used with a predicate, it leaves only elements that match it.
- `clear()` removes all elements from a list and leaves it empty.

```
fun main() {
    val numbers = mutableListOf(1, 2, 3, 4)
    println(numbers)
    numbers.retainAll { it >= 3 }
    println(numbers)
    numbers.clear()
    println(numbers)

    val numbersSet = mutableSetOf("one", "two", "three", "four")
    numbersSet.removeAll(setOf("one", "two"))
    println(numbersSet)
}
```

Another way to remove elements from a collection is with the `minusAssign` (`-=`) operator – the in-place version of `minus` . The second argument can be a single instance of the element type or another collection. With a single element on the right-hand side, `-=` removes the *first* occurrence of it. In turn, if it's a collection, *all* occurrences of its elements are removed. For example, if a list contains duplicate elements, they are removed at once. The second operand can contain elements that are not present in the collection. Such elements don't affect the operation execution.

```
fun main() {
    val numbers = mutableListOf("one", "two", "three", "three", "four")
    numbers -= "three"
    println(numbers)
    numbers -= listOf("four", "five")
    //numbers -= listOf("four")    // does the same as above
    println(numbers)
}
```

Updating elements

Lists and maps also provide operations for updating elements. They are described in List-specific Operations and Map Specific Operations. For sets, updating doesn't make sense since it's actually removing an element and adding another one.

List-specific operations

`List` is the most popular type of built-in collection in Kotlin. Index access to the elements of lists provides a powerful set of operations for lists.

Retrieve elements by index

Lists support all common operations for element retrieval: `elementAt()`, `first()`, `last()`, and others listed in Retrieve single elements. What is specific for lists is index access to the elements, so the simplest way to read an element is retrieving it by index. That is done with the `get()` function with the index passed in the argument or the shorthand `[index]` syntax.

If the list size is less than the specified index, an exception is thrown. There are two other functions that help you avoid such exceptions:

- `getOrElse()` lets you provide the function for calculating the default value to return if the index isn't present in the collection.
- `getOrNull()` returns `null` as the default value.

```
fun main() {
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.get(0))
    println(numbers[0])
    //numbers.get(5)                        // exception!
    println(numbers.getOrNull(5))             // null
    println(numbers.getOrElse(5, {it}))       // 5
}
```

Retrieve list parts

In addition to common operations for Retrieving Collection Parts, lists provide the `subList()` function that returns a view of the specified elements range as a list. Thus, if an element of the original collection changes, it also changes in the previously created sublists and vice versa.

```
fun main() {
    val numbers = (0..13).toList()
    println(numbers.subList(3, 6))
}
```

Find element positions

Linear search

In any lists, you can find the position of an element using the functions `indexOf()` and `lastIndexOf()`. They return the first and the last position of an element equal to the given argument in the list. If there are no such elements, both functions return `-1`.

```
fun main() {
    val numbers = listOf(1, 2, 3, 4, 2, 5)
    println(numbers.indexOf(2))
    println(numbers.lastIndexOf(2))
}
```

There is also a pair of functions that take a predicate and search for elements matching it:

- `indexOfFirst()` returns the *index of the first* element matching the predicate or `-1` if there are no such elements.
- `indexOfLast()` returns the *index of the last* element matching the predicate or `-1` if there are no such elements.

```
fun main() {
    val numbers = mutableListOf(1, 2, 3, 4)
    println(numbers.indexOfFirst { it > 2})
    println(numbers.indexOfLast { it % 2 == 1})
}
```

Binary search in sorted lists

There is one more way to search elements in lists – binary search. It works significantly faster than other built-in search functions but *requires the list to be sorted* in ascending order according to a certain ordering: natural or another one provided in the function parameter. Otherwise, the result is undefined.

To search an element in a sorted list, call the `binarySearch()` function passing the value as an argument. If such an element exists, the function returns its index; otherwise, it returns `(-insertionPoint - 1)` where `insertionPoint` is the index where this element should be inserted so that the list remains sorted. If there is more than one element with the given value, the search can return any of their indices.

You can also specify an index range to search in: in this case, the function searches only between two provided indices.

```
fun main() {
    val numbers = mutableListOf("one", "two", "three", "four")
    numbers.sort()
    println(numbers)
    println(numbers.binarySearch("two"))  // 3
    println(numbers.binarySearch("z")) // -5
    println(numbers.binarySearch("two", 0, 2))  // -3
}
```

Comparator binary search

When list elements aren't `Comparable`, you should provide a `Comparator` to use in the binary search. The list must be sorted in ascending order according to this `Comparator`. Let's have a look at an example:

```
data class Product(val name: String, val price: Double)

fun main() {
    val productList = listOf(
        Product("WebStorm", 49.0),
        Product("AppCode", 99.0),
        Product("DotTrace", 129.0),
        Product("ReSharper", 149.0))

    println(productList.binarySearch(Product("AppCode", 99.0), compareBy<Product> { it.price }.thenBy
{ it.name }))
}
```

Here's a list of `Product` instances that aren't `Comparable` and a `Comparator` that defines the order: product `p1` precedes product `p2` if `p1`'s price is less than `p2`'s price. So, having a list sorted ascending according to this order, we use `binarySearch()` to find the index of the specified `Product`.

Custom comparators are also handy when a list uses an order different from natural one, for example, a case-insensitive order for `String` elements.

```
fun main() {
    val colors = listOf("Blue", "green", "ORANGE", "Red", "yellow")
    println(colors.binarySearch("RED", String.CASE_INSENSITIVE_ORDER)) // 3
}
```

Comparison binary search

Binary search with *comparison* function lets you find elements without providing explicit search values. Instead, it takes a comparison function mapping elements to `Int` values and searches for the element where the function returns zero. The list must be sorted in the ascending order according to the provided function; in other words, the return values of comparison must grow from one list element to the next one.

```
import kotlin.math.sign
data class Product(val name: String, val price: Double)

fun priceComparison(product: Product, price: Double) = sign(product.price - price).toInt()

fun main() {
    val productList = listOf(
        Product("WebStorm", 49.0),
        Product("AppCode", 99.0),
        Product("DotTrace", 129.0),
        Product("ReSharper", 149.0))

    println(productList.binarySearch { priceComparison(it, 99.0) })
}
```

Both comparator and comparison binary search can be performed for list ranges as well.

List write operations

In addition to the collection modification operations described in Collection write operations, mutable lists support specific write operations. Such operations use the index to access elements to broaden the list modification capabilities.

Add

To add elements to a specific position in a list, use `add()` and `addAll()` providing the position for element insertion as an additional argument. All elements that come after the position shift to the right.

```
fun main() {
    val numbers = mutableListOf("one", "five", "six")
    numbers.add(1, "two")
    numbers.addAll(2, listOf("three", "four"))
    println(numbers)
}
```

Update

Lists also offer a function to replace an element at a given position - `set()` and its operator form `[]`. `set()` doesn't change the indexes of other elements.

```
fun main() {
    val numbers = mutableListOf("one", "five", "three")
    numbers[1] =  "two"
    println(numbers)
}
```

`fill()` simply replaces all the collection elements with the specified value.

```
fun main() {
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.fill(3)
    println(numbers)
}
```

Remove

To remove an element at a specific position from a list, use the `removeAt()` function providing the position as an argument. All indices of elements that come after the element being removed will decrease by one.

```
fun main() {
    val numbers = mutableListOf(1, 2, 3, 4, 3)
    numbers.removeAt(1)
    println(numbers)
}
```

Sort

In Collection Ordering, we describe operations that retrieve collection elements in specific orders. For mutable lists, the standard library offers similar extension functions that perform the same ordering operations in place. When you apply such an operation to a list instance, it changes the order of elements in that exact instance.

The in-place sorting functions have similar names to the functions that apply to read-only lists, but without the `ed/d` suffix:

- `sort*` instead of `sorted*` in the names of all sorting functions: `sort()`, `sortDescending()`, `sortBy()`, and so on.
- `shuffle()` instead of `shuffled()`.

- `reverse()` instead of `reversed()`.

`asReversed()` called on a mutable list returns another mutable list which is a reversed view of the original list. Changes in that view are reflected in the original list. The following example shows sorting functions for mutable lists:

```
fun main() {
    val numbers = mutableListOf("one", "two", "three", "four")

    numbers.sort()
    println("Sort into ascending: $numbers")
    numbers.sortDescending()
    println("Sort into descending: $numbers")

    numbers.sortBy { it.length }
    println("Sort into ascending by length: $numbers")
    numbers.sortByDescending { it.last() }
    println("Sort into descending by the last letter: $numbers")

    numbers.sortWith(compareBy<String> { it.length }.thenBy { it })
    println("Sort by Comparator: $numbers")

    numbers.shuffle()
    println("Shuffle: $numbers")

    numbers.reverse()
    println("Reverse: $numbers")
}
```

Set-specific operations

The Kotlin collections package contains extension functions for popular operations on sets: finding intersections, merging, or subtracting collections from each other.

To merge two collections into one, use the `union()` function. It can be used in the infix form `a union b`. Note that for ordered collections, the order of the operands is important. In the resulting collection, the elements of the first operand go before the elements of the second:

```
fun main() {
    val numbers = setOf("one", "two", "three")

    // output according to the order
    println(numbers union setOf("four", "five"))
    // [one, two, three, four, five]
    println(setOf("four", "five") union numbers)
    // [four, five, one, two, three]
}
```

To find an intersection between two collections (elements present in both of them), use the `intersect()` function. To find collection elements not present in another collection, use the `subtract()` function. Both these functions can be called in the infix form as well, for example, `a intersect b`:

```
fun main() {
    val numbers = setOf("one", "two", "three")

    // same output
    println(numbers intersect setOf("two", "one"))
    // [one, two]
    println(numbers subtract setOf("three", "four"))
    // [one, two]
    println(numbers subtract setOf("four", "three"))
    // [one, two]
}
```

To find the elements present in either one of the two collections but not in their intersection, you can also use the `union()` function. For this operation (known as symmetric difference), calculate the differences between the two collections and merge the results:

```
fun main() {
    val numbers = setOf("one", "two", "three")
    val numbers2 = setOf("three", "four")

    // merge differences
    println((numbers - numbers2) union (numbers2 - numbers))
    // [one, two, four]
}
```

You can also apply `union()`, `intersect()`, and `subtract()` functions to lists. However, their result is *always* a `Set`. In this result, all the duplicate elements are merged into one and the index access is not available:

```
fun main() {
    val list1 = listOf(1, 1, 2, 3, 5, 8, -1)
    val list2 = listOf(1, 1, 2, 2, 3, 5)

    // result of intersecting two lists is a Set
    println(list1 intersect list2)
    // [1, 2, 3, 5]

    // equal elements are merged into one
    println(list1 union list2)
    // [1, 2, 3, 5, 8, -1]
}
```

Map-specific operations

In maps, types of both keys and values are user-defined. Key-based access to map entries enables various map-specific processing capabilities from getting a value by key to separate filtering of keys and values. On this page, we provide descriptions of the map processing functions from the standard library.

Retrieve keys and values

For retrieving a value from a map, you must provide its key as an argument of the `get()` function. The shorthand `[key]` syntax is also supported. If the given key is not found, it returns `null`. There is also the function `getValue()` which has slightly different behavior: it throws an exception if the key is not found in the map. Additionally, you have more options to handle the key absence:

- `getOrElse()` works the same way as for lists: the values for non-existent keys are returned from the given lambda function.

- `getOrDefault()` returns the specified default value if the key is not found.

For maps with nullable values, use the following functions instead, which handle missing keys and `null` values explicitly:

- `getOrElseIfNull()` returns the result of the specified default value if the key is missing or has a `null` value.
- `getOrElseIfMissing()` returns the result of the specified default value if the key is missing.

Here's an example that shows the difference between these functions:

```
@OptIn(ExperimentalStdlibApi::class)
fun main() {
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap.get("one"))
    // 1

    println(numbersMap["one"])
    // 1

    println(numbersMap.getOrDefault("four", 10))
    // 10

    println(numbersMap["five"])
    // null

    val nullableMap = mapOf("one" to 1, "two" to null)
    println(nullableMap.getOrElseIfNull("two") { 0 })
    // 0

    println(nullableMap.getOrElseIfMissing("two") { 0 })
    // null

    // Throws an exception because "six" is missing from the map
    // numbersMap.getValue("six")

}
```

To perform operations on all keys or all values of a map, you can retrieve them from the properties `keys` and `values` accordingly. `keys` is a set of all map keys and `values` is a collection of all map values.

```
fun main() {
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap.keys)
    println(numbersMap.values)
}
```

Filter

You can filter maps with the `filter()` function as well as other collections. When calling `filter()` on a map, pass to it a predicate with a `Pair` as an argument. This enables you to use both the key and the value in the filtering predicate.

```
fun main() {
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
    println(filteredMap)
}
```

There are also two specific ways for filtering maps: by keys and by values. For each way, there is a function: `filterKeys()` and `filterValues()`. Both return a new map of entries which match the given predicate. The predicate for `filterKeys()` checks only the element keys, the one for `filterValues()` checks only values.

```
fun main() {
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredKeysMap = numbersMap.filterKeys { it.endsWith("1") }
    val filteredValuesMap = numbersMap.filterValues { it < 10 }

    println(filteredKeysMap)
    println(filteredValuesMap)
}
```

Plus and minus operators

Due to the key access to elements, `plus` (`+`) and `minus` (`-`) operators work for maps differently than for other collections. `plus` returns a `Map` that contains elements of its both operands: a `Map` on the left and a `Pair` or another `Map` on the right. When the right-hand side operand contains entries with keys present in the left-hand side `Map`, the result map contains the entries from the right side.

```
fun main() {
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap + Pair("four", 4))
    println(numbersMap + Pair("one", 10))
    println(numbersMap + mapOf("five" to 5, "one" to 11))
}
```

`minus` creates a `Map` from entries of a `Map` on the left except those with keys from the right-hand side operand. So, the right-hand side operand can be either a single key or a collection of keys: list, set, and so on.

```
fun main() {
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap - "one")
    println(numbersMap - listOf("two", "four"))
}
```

For details on using `plusAssign` (`+=`) and `minusAssign` (`-=`) operators on mutable maps, see Map write operations below.

Map write operations

Mutable maps offer map-specific write operations. These operations let you change the map content using the key-based access to the values.

There are certain rules that define write operations on maps:

- Values can be updated. In turn, keys never change: once you add an entry, its key is constant.
- For each key, there is always a single value associated with it. You can add and remove whole entries.

Below are descriptions of the standard library functions for write operations available on mutable maps.

Add and update entries

To add a new key-value pair to a mutable map, use `put()`. When a new entry is put into a `LinkedHashMap` (the default map implementation), it is added so that it comes last when iterating the map. In sorted maps, the positions of new elements are defined by the order of their keys.

```
fun main() {
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap.put("three", 3)
    println(numbersMap)
}
```

To add multiple entries at a time, use `putAll()`. Its argument can be a `Map` or a group of `Pair`s: `Iterable`, `Sequence`, or `Array`.

```
fun main() {
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
    numbersMap.putAll(setOf("four" to 4, "five" to 5))
    println(numbersMap)
}
```

Both `put()` and `putAll()` overwrite the values if the given keys already exist in the map. Thus, you can use them to update the values of map entries.

```
fun main() {
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    val previousValue = numbersMap.put("one", 11)
    println("value associated with 'one', before: $previousValue, after: ${numbersMap["one"]}")
    println(numbersMap)
}
```

You can also add new entries to maps using the shorthand operator form. There are two ways:

- `plusAssign` (`+=`) operator.
- the `[]` operator alias for `set()`.

```
fun main() {
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap["three"] = 3     // calls numbersMap.put("three", 3)
    numbersMap += mapOf("four" to 4, "five" to 5)
    println(numbersMap)
}
```

When called with the key present in the map, operators overwrite the values of the corresponding entries.

Add default values for missing entries

To return an existing value or add a default value when no value is available, use the `.getOrPut()` extension function. If the key is missing or has a `null` value, `.getOrPut()` stores the default value and returns it.

For maps with nullable values, you can use the `.getOrPutIfNull()` and `.getOrPutIfMissing()` functions to control how `null` values are treated:

- `getOrPutIfNull()` behaves like `getOrPut()` and uses the default value if the key is missing or has a `null` value.
- `getOrPutIfMissing()` uses the default value only if the key is missing.

The `getOrPutIfNull()` and `getOrPutIfMissing()` functions are Experimental. To opt in, use the `@OptIn(ExperimentalStdlibApi::class)` annotation.

Here's an example:

```
@OptIn(ExperimentalStdlibApi::class)
fun main() {
    val mapForNull = mutableMapOf<String, Int?>("one" to null)
    val mapForMissing = mutableMapOf<String, Int?>("one" to null)

    // Replaces the value if "one" has a null value
    mapForNull.getOrPutIfNull("one") { 1 }

    println(mapForNull)
    // {one=1}

    // Keeps the null value because "one" exists in the map
    mapForMissing.getOrPutIfMissing("one") { 1 }

    println(mapForMissing)
    // {one=null}
}
```

Remove entries

To remove an entry from a mutable map, use the `remove()` function. When calling `remove()`, you can pass either a key or a whole key-value-pair. If you specify both the key and value, the element with this key will be removed only if its value matches the second argument.

```
fun main() {
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
    numbersMap.remove("one")
    println(numbersMap)
    numbersMap.remove("three", 4)            //doesn't remove anything
    println(numbersMap)
}
```

You can also remove entries from a mutable map by their keys or values. To do this, call `remove()` on the map's keys or values providing the key or the value of an entry. When called on values, `remove()` removes only the first entry with the given value.

```
fun main() {
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3, "threeAgain" to 3)
    numbersMap.keys.remove("one")
    println(numbersMap)
    numbersMap.values.remove(3)
    println(numbersMap)
}
```

The `minusAssign` (`-=`) operator is also available for mutable maps.

```
fun main() {
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
    numbersMap -= "two"
    println(numbersMap)
    numbersMap -= "five"             //doesn't remove anything
    println(numbersMap)
}
```

Plus and minus operators

In Kotlin, `plus` (`+`) and `minus` (`-`) operators are defined for collections. They take a collection as the first operand; the second operand can be either an element or another collection. The return value is a new read-only collection:

- The result of `plus` contains the elements from the original collection *and* from the second operand.
- The result of `minus` contains the elements of the original collection *except* the elements from the second operand. If it's an element, `minus` removes its *first* occurrence; if it's a collection, *all* occurrences of its elements are removed.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")

    val plusList = numbers + "five"
    val minusList = numbers - listOf("three", "four")
    println(plusList)
    println(minusList)
}
```

For the details on `plus` and `minus` operators for maps, see Map specific operations. The augmented assignment operators `plusAssign` (`+=`) and `minusAssign` (`-=`) are also defined for collections. However, for read-only collections, they actually use the `plus` or `minus` operators and try to assign the result to the same variable. Thus, they are available only on `var` read-only collections. For mutable collections, they modify the collection if it's a `val` . For more details, see Collection write operations.

Sequences

Along with collections, the Kotlin standard library contains another type – *sequences* (`Sequence<T>`). Unlike collections, sequences don't contain elements, they produce them while iterating. Sequences offer the same functions as `Iterable` but implement another approach to multistep collection processing.

When the processing of an `Iterable` includes multiple steps, they are executed eagerly: each processing step completes and returns its result – an intermediate collection. The following step executes on this collection. In turn, multistep processing of sequences is executed lazily when possible: actual computing happens only when the result of the whole processing chain is requested.

The order of operations execution is different as well: `Sequence` performs all the processing steps one-by-one for every single element. In turn, `Iterable` completes each step for the whole collection and then proceeds to the next step.

So, the sequences let you avoid building results of intermediate steps, therefore improving the performance of the whole collection processing chain. However, the lazy nature of sequences adds some overhead which may be significant when processing smaller collections or doing simpler computations. Hence, you should consider both `Sequence` and `Iterable` and decide which one is better for your case.

Construct

From elements

To create a sequence, call the `sequenceOf()` function listing the elements as its arguments.

```
val numbersSequence = sequenceOf("four", "three", "two", "one")
```

From an Iterable

If you already have an `Iterable` object (such as a `List` or a `Set`), you can create a sequence from it by calling `asSequence()`.

```
val numbers = listOf("one", "two", "three", "four")
val numbersSequence = numbers.asSequence()
```

From a function

One more way to create a sequence is by building it with a function that calculates its elements. To build a sequence based on a function, call `generateSequence()` with this function as an argument. Optionally, you can specify the first element as an explicit value or a result of a function call. The sequence generation stops when the provided function returns `null`. So, the sequence in the example below is infinite.

```
fun main() {
    val oddNumbers = generateSequence(1) { it + 2 } // `it` is the previous element
    println(oddNumbers.take(5).toList())
    //println(oddNumbers.count())     // error: the sequence is infinite
}
```

To create a finite sequence with `generateSequence()`, provide a function that returns `null` after the last element you need.

```
fun main() {
    val oddNumbersLessThan10 = generateSequence(1) { if (it < 8) it + 2 else null }
    println(oddNumbersLessThan10.count())
}
```

From chunks

Finally, there is a function that lets you produce sequence elements one by one or by chunks of arbitrary sizes – the `sequence()` function. This function takes a lambda expression containing calls of `yield()` and `yieldAll()` functions. They return an element to the sequence consumer and suspend the execution of `sequence()` until the next element is requested by the consumer. `yield()` takes a single element as an argument; `yieldAll()` can take an `Iterable` object, an `Iterator`, or another `Sequence`. A `Sequence` argument of `yieldAll()` can be infinite. However, such a call must be the last: all subsequent calls will never be executed.

```
fun main() {
    val oddNumbers = sequence {
        yield(1)
        yieldAll(listOf(3, 5))
        yieldAll(generateSequence(7) { it + 2 })
    }
    println(oddNumbers.take(5).toList())
}
```

Sequence operations

The sequence operations can be classified into the following groups regarding their state requirements:

- *Stateless* operations require no state and process each element independently, for example, `map()` or `filter()`. Stateless operations can also require a small constant amount of state to process an element, for example, `take()` or `drop()`.
- *Stateful* operations require a significant amount of state, usually proportional to the number of elements in a sequence.

If a sequence operation returns another sequence, which is produced lazily, it's called *intermediate*. Otherwise, the operation is *terminal*. Examples of terminal operations are `toList()` or `sum()`. Sequence elements can be retrieved only with terminal operations.

Sequences can be iterated multiple times; however, some sequence implementations might constrain themselves to be iterated only once. That is mentioned specifically in their documentation.

Sequence processing example

Let's take a look at the difference between `Iterable` and `Sequence` with an example.

Iterable

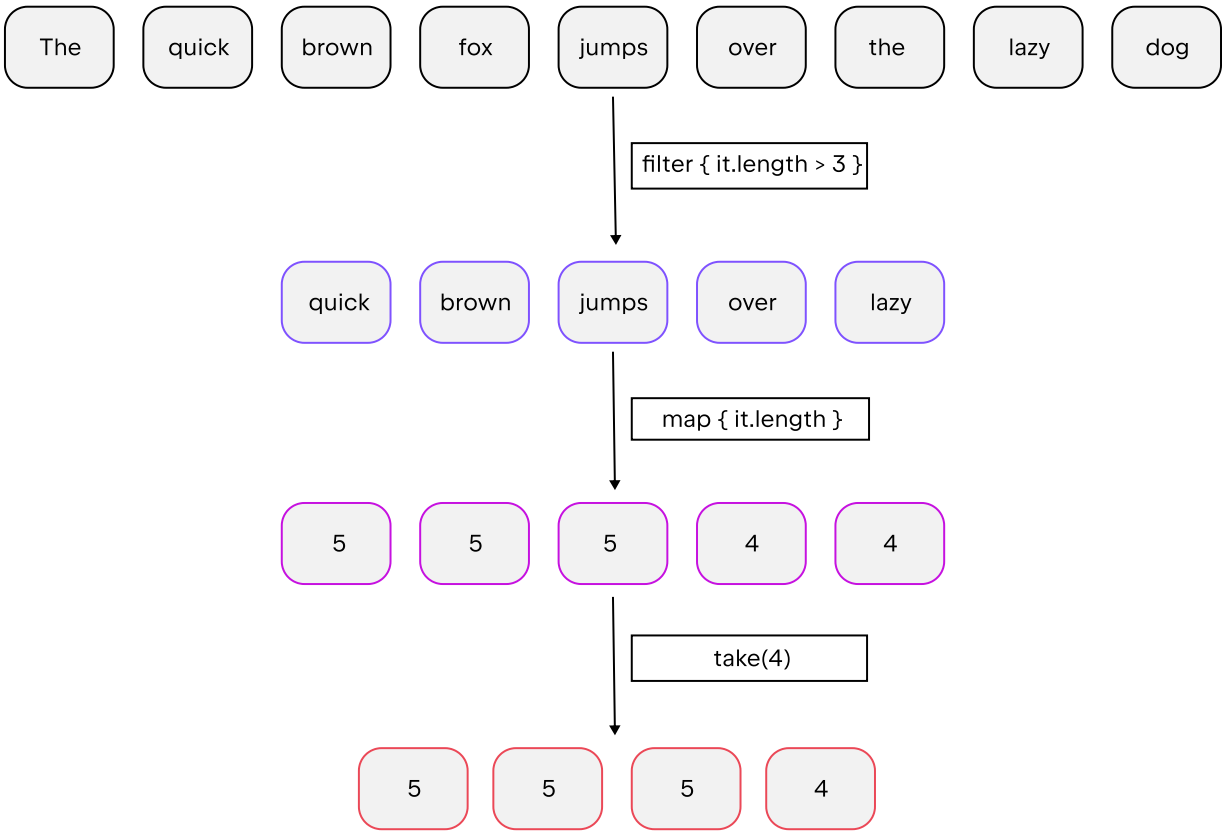
Assume that you have a list of words. The code below filters the words longer than three characters and prints the lengths of first four such words.

```
fun main() {
    val words = "The quick brown fox jumps over the lazy dog".split(" ")
    val lengthsList = words.filter { println("filter: $it"); it.length > 3 }
        .map { println("length: ${it.length}"); it.length }
        .take(4)

    println("Lengths of first 4 words longer than 3 chars:")
    println(lengthsList)
}
```

When you run this code, you'll see that the `filter()` and `map()` functions are executed in the same order as they appear in the code. First, you see `filter:` for all elements, then `length:` for the elements left after filtering, and then the output of the two last lines.

This is how the list processing goes:



List processing

Sequence

Now let's write the same with sequences:

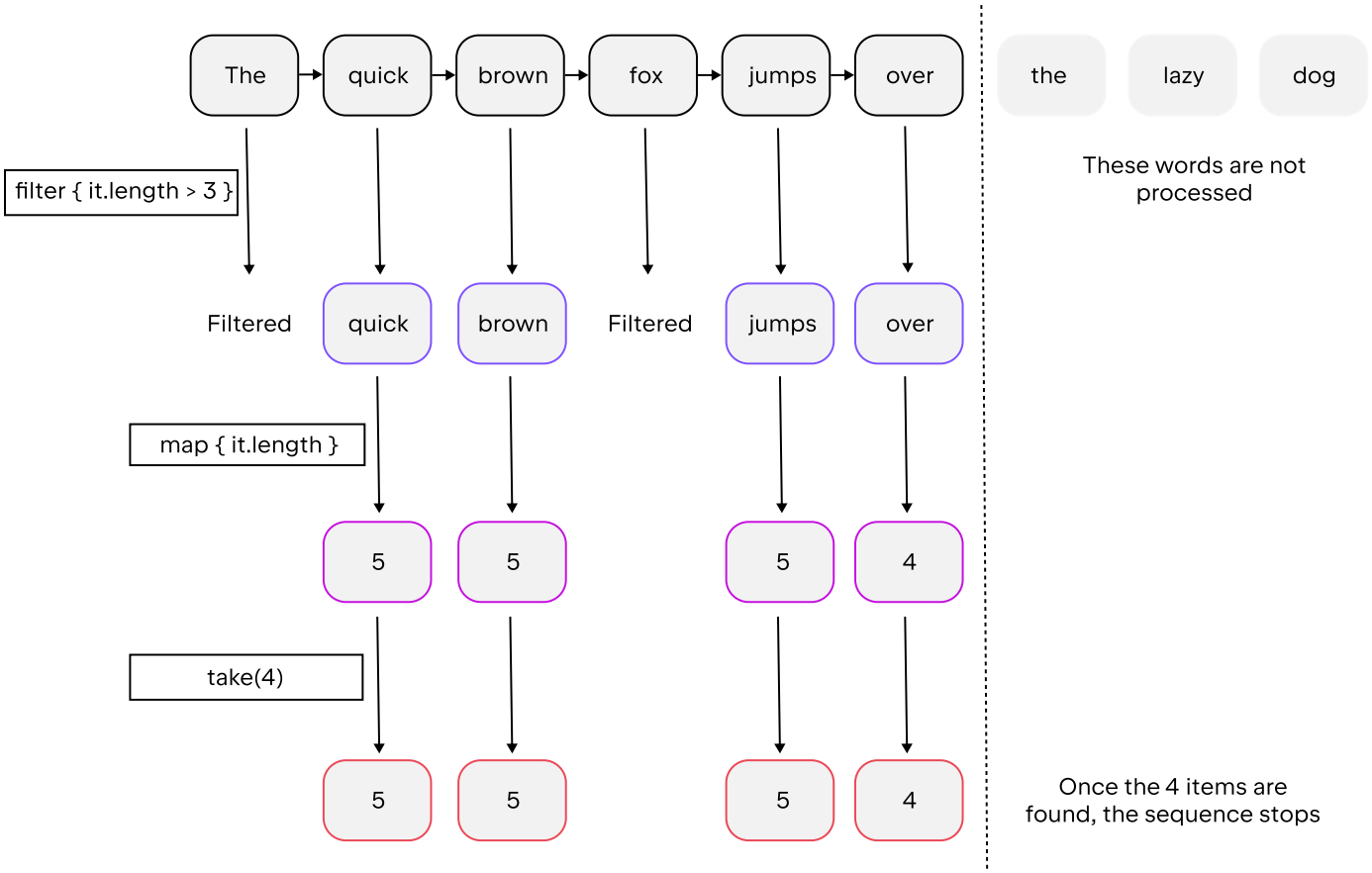
```
fun main() {
    val words = "The quick brown fox jumps over the lazy dog".split(" ")
    //convert the List to a Sequence
    val wordsSequence = words.asSequence()

    val lengthsSequence = wordsSequence.filter { println("filter: $it"); it.length > 3 }
        .map { println("length: ${it.length}"); it.length }
        .take(4)

    println("Lengths of first 4 words longer than 3 chars")
    // terminal operation: obtaining the result as a List
    println(lengthsSequence.toList())
}
```

The output of this code shows that the `filter()` and `map()` functions are called only when building the result list. So, you first see the line of text `"Lengths of.."` and then the sequence processing starts. Note that for elements left after filtering, the map executes before filtering the next element. When the result size reaches 4, the processing stops because it's the largest possible size that `take(4)` can return.

The sequence processing goes like this:



Sequences processing

In this example, the lazy processing of elements and stopping after finding four items reduces the number of operations compared to using a list approach.

Ranges and progressions

Ranges and progressions define sequences of values in Kotlin, supporting range operators, iteration, custom step values, and arithmetic progressions.

Ranges

Kotlin lets you easily create ranges of values using the `.rangeTo()` and `.rangeUntil()` functions from the `kotlin.ranges` package.

A range represents an ordered set of values with a defined start and end. By default, it increments by 1 at each step. For example, `1..4` represents the numbers 1, 2, 3, and 4.

To create:

- a closed-ended range, call the `.rangeTo()` function with the `..` operator. This includes both the start and end values.
- an open-ended range, call the `.rangeUntil()` function with the `..<` operator. This includes the start value but excludes the end value.

For example:

```
fun main() {
    // Closed-ended range: includes both 1 and 4
    println(4 in 1..4)
    // true

    // Open-ended range: includes 1, excludes 4
    println(4 in 1..<4)
    // false
}
```

Ranges are particularly useful for iterating over `for` loops:

```
fun main() {
    for (i in 1..4) print(i)
    // 1234
}
```

To iterate numbers in reverse order, use the `downTo` function instead of `..`.

```
fun main() {
    for (i in 4 downTo 1) print(i)
    // 4321
}
```

You can also iterate over numbers with a custom step using the `step()` function, instead of the default increment of 1:

```
fun main() {
    for (i in 0..8 step 2) print(i)
    println()
    // 02468
    for (i in 0..<8 step 2) print(i)
    println()
    // 0246
    for (i in 8 downTo 0 step 2) print(i)
    // 86420
}
```

Progressions

The ranges of integral types, such as `Int`, `Long`, and `Char`, can be treated as arithmetic progressions. In Kotlin, these progressions are defined by special types: `IntProgression`, `LongProgression`, and `CharProgression`.

Progressions have three essential properties: the `first` element, the `last` element, and a non-zero `step`. The first element is `first`, subsequent elements are the previous element plus a `step`. Iteration over a progression with a positive step is equivalent to an indexed `for` loop in Java/JavaScript.

```
for (int i = first; i ⩽ last; i += step) {
  // ...
}
```

When you create a progression implicitly by iterating a range, this progression's `first` and `last` elements are the range's endpoints, and the `step` is 1.

```
fun main() {
    for (i in 1..10) print(i)
    // 12345678910
}
```

To define a custom progression step, use the `step` function on a range.

```
fun main() {
    for (i in 1..8 step 2) print(i)
    // 1357
}
```

The `last` element of the progression is calculated this way:

- For a positive step: the maximum value not greater than the end value such that `(last - first) % step == 0`.
- For a negative step: the minimum value not less than the end value such that `(last - first) % step == 0`.

Thus, the `last` element is not always the same as the specified end value.

```
fun main() {
    for (i in 1..9 step 3) print(i) // the last element is 7
    // 147
}
```

Progressions implement `Iterable<N>`, where `N` is `Int`, `Long`, or `Char` respectively, so you can use them in various collection functions like `map`, `filter`, and other.

```
fun main() {
    println((1..10).filter { it % 2 == 0 })
    // [2, 4, 6, 8, 10]
}
```

Read standard input

Use the `readln()` function to read data from the standard input. It reads the whole line as a string:

```
// Reads and stores the user input in a variable. For example: Hi there!
val myInput = readln()

println(myInput)
// Hi there!

// Reads and prints the user input without storing it in a variable. For example: Hi, Kotlin!
println(readln())
// Hi, Kotlin!
```

You can also read multiline input as one string. Use the `joinToString()` function to combine the lines:

```
val input = generateSequence(::readlnOrNull)
  .joinToString("\n")
```

Use `readlnOrNull()` instead of `readln()` to read input until end-of-file without throwing an exception.

To work with data types other than strings, you can convert the input using conversion functions like `.toInt()`, `.toLong()`, `.toDouble()`, `.toFloat()`, or `.toBoolean()`. It is possible to read multiple inputs of different data types and store each input in a variable:

```
// Converts the input from a string to an integer value. For example: 12
val myNumber = readln().toInt()
println(myNumber)
// 12

// Converts the input from a string to a double value. For example: 345
val myDouble = readln().toDouble()
println(myDouble)
// 345.0

// Converts the input from a string to a boolean value. For example: true
val myBoolean = readln().toBoolean()
println(myBoolean)
// true
```

These conversion functions assume the user enters a valid representation of the target data type. For example, converting "hello" to an integer using `.toInt()` would result in an exception as the function expects a number in the string input.

To read several input elements separated by a delimiter, use the `.split()` function specifying the delimiter. The following code sample reads from the standard input, splits the input into a list of elements based on the delimiter, and converts each element of the list into a specific type:

```
// Reads the input, assuming the elements are separated by spaces, and converts them into integers.
For example: 1 2 3
val numbers = readln().split(' ').map { it.toInt() }
println(numbers)
//[1, 2, 3]

// Reads the input, assuming the elements are separated by commas, and converts them into doubles. For
example: 4,5,6
val doubles = readln().split(',').map { it.toDouble() }
println(doubles)
//[4.0, 5.0, 6.0]
```

For an alternative way to read user input in Kotlin/JVM, see Standard input with Java Scanner.

Handle standard input safely

You can use the `.toIntOrNull()` function to safely convert user input from a string to an integer. This function returns an integer if the conversion is successful. However, if the input is not a valid representation of an integer, it returns `null`:

```
// Returns null if the input is invalid. For example: Hello!
val wrongInt = readln().toIntOrNull()
println(wrongInt)
// null

// Converts a valid input from a string to an integer. For example: 13
val correctInt = readln().toIntOrNull()
println(correctInt)
// 13
```

The `readlnOrNull()` function is also helpful in safely handling the user input. The `readlnOrNull()` function reads from the standard input and returns null if the end of the input is reached, whereas `readln()` throws an exception in such a case.

Opt-in requirements

The Kotlin standard library provides a mechanism for requiring and giving explicit consent to use certain API elements. This mechanism allows library authors to inform users about specific conditions that require opt-in, such as when an API is in an experimental state and is likely to change in the future.

To protect users, the compiler warns about these conditions and requires them to opt in before the API can be used.

Opt in to API

If a library author marks a declaration from their library's API as requiring opt-in, you must give explicit consent before you can use it in your code. There are several ways to opt in. We recommend choosing the approach that best suits your situation.

Opt in locally

To opt in to a specific API element when you use it in your code, use the `@OptIn` annotation with a reference to the experimental API marker. For example, suppose you want to use the `DateProvider` class, which requires an opt-in:

```
// Library code
@RequiresOptIn(message = "This API is experimental. It could change in the future without notice.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime

@MyDateTime
// A class requiring opt-in
class DateProvider
```

In your code, before declaring a function that uses the `DateProvider` class, add the `@OptIn` annotation with a reference to the `MyDateTime` annotation class:

```
// Client code
@OptIn(MyDateTime::class)

// Uses DateProvider
fun getDate(): Date {
    val dateProvider: DateProvider
    // ...
}
```

It's important to note that with this approach, if the `getDate()` function is called elsewhere in your code or used by another developer, no opt-in is required:

```
// Client code
@OptIn(MyDateTime::class)

// Uses DateProvider
fun getDate(): Date {
    val dateProvider: DateProvider
    // ...
}

fun displayDate() {
    // OK: No opt-in is required
    println(getDate())
}
```

The opt-in requirement is not propagated, which means others might unknowingly use experimental APIs. To avoid this, it is safer to propagate the opt-in requirements.

Propagate opt-in requirements

When you use API in your code that's intended for third-party use, such as in a library, you can propagate its opt-in requirement to your API as well. To do this, mark your declaration with the same opt-in requirement annotation used by the library.

For example, before declaring a function that uses the `DateProvider` class, add the `@MyDateTime` annotation:

```
// Client code
@MyDateTime
fun getDate(): Date {
    // OK: the function requires opt-in as well
    val dateProvider: DateProvider
    // ...
}

fun displayDate() {
    println(getDate())
    // Error: getDate() requires opt-in
}
```

As you can see in this example, the annotated function appears to be a part of the `@MyDateTime` API. The opt-in propagates the opt-in requirement to users of the `getDate()` function.

If the signature of an API element includes a type that requires opt-in, the signature itself must also require opt-in. Otherwise, if an API element doesn't require opt-in, but its signature includes a type that does, using it triggers an error.

```
// Client code
@MyDateTime
fun getDate(dateProvider: DateProvider = DateProvider()): Date

@MyDateTime
fun displayDate() {
    // OK: the function requires opt-in as well
    println(getDate())
}
```

Similarly, if you apply `@OptIn` to a declaration whose signature includes a type that requires opt-in, the opt-in requirement still propagates:

```
// Client code
@OptIn(MyDateTime::class)
// Propagates opt-in due to DateProvider in the signature
fun getDate(dateProvider: DateProvider = DateProvider()): Date

fun displayDate() {
    println(getDate())
    // Error: getDate() requires opt-in
}
```

When propagating opt-in requirements, it's important to understand that if an API element becomes stable and no longer has an opt-in requirement, any other API elements that still have the opt-in requirement remain experimental. For example, suppose a library author removes the opt-in requirement for the `getDate()` function because it's now stable:

```
// Library code
// No opt-in requirement
fun getDate(): Date {
    val dateProvider: DateProvider
    // ...
}
```

If you use the `displayDate()` function without removing the opt-in annotation, it remains experimental even though the opt-in is no longer needed:

```
// Client code

// Still experimental!
@MyDateTime
fun displayDate() {
    // Uses a stable library function
    println(getDate())
}
```

Opt in to multiple APIs

To opt in to multiple APIs, mark the declaration with all their opt-in requirement annotations. For example:

```
@ExperimentalCoroutinesApi
@FlowPreview
```

Or alternatively with `@OptIn`:

```
@OptIn(ExperimentalCoroutinesApi::class, FlowPreview::class)
```

Opt in a file

To use an API that requires opt-in for all functions and classes in a file, add the file-level annotation `@file:OptIn` to the top of the file before the package specification and imports.

```
// Client code
@file:OptIn(MyDateTime::class)
```

Opt in a module

> The `-opt-in` compiler option is available since Kotlin 1.6.0. For earlier Kotlin versions, use `-Xopt-in`.

If you don't want to annotate every usage of APIs that require opt-in, you can opt in to them for your whole module. To opt in to using an API in a module, compile it with the argument `-opt-in`, specifying the fully qualified name of the opt-in requirement annotation of the API you use: `-opt-in=org.mylibrary.OptInAnnotation`. Compiling with this argument has the same effect as if every declaration in the module has the annotation `@OptIn(OptInAnnotation::class)`.

If you build your module with Gradle, you can add arguments like this:

Kotlin

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named<KotlinCompilationTask<*>>("compileKotlin").configure {
    compilerOptions.optIn.add("org.mylibrary.OptInAnnotation")
}
```

Groovy

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        optIn.add('org.mylibrary.OptInAnnotation')
    }
}
```

If your Gradle module is a multiplatform module, use the `optIn` method:

Kotlin

```
kotlin {
    compilerOptions {
        optIn.add("org.mylibrary.OptInAnnotation")
    }
}
```

Groovy

```
kotlin {
    compilerOptions {
        optIn.add('org.mylibrary.OptInAnnotation')
    }
}
```

For Maven, use the following:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <executions>...</executions>
            <configuration>
                <args>
                    <arg>-opt-in=org.mylibrary.OptInAnnotation</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

To opt in to multiple APIs on the module level, add one of the described arguments for each opt-in requirement marker used in your module.

Opt in to inherit from a class or interface

Sometimes, a library author provides an API but wants to require users to explicitly opt in before they can extend it. For example, the library API may be stable for use but not for inheritance, as it might be extended in the future with new abstract functions. Library authors can enforce this by marking open or abstract classes and non-functional interfaces with the `@SubclassOptInRequired` annotation.

To opt in to use such an API element and extend it in your code, use the `@SubclassOptInRequired` annotation with a reference to the annotation class. For example, suppose you want to use the `CoreLibraryApi` interface, which requires an opt-in:

```
// Library code
@RequiresOptIn(
 level = RequiresOptIn.Level.WARNING,
 message = "Interfaces in this library are experimental"
)
annotation class UnstableApi()

@SubclassOptInRequired(UnstableApi::class)
// An interface requiring opt-in to extend
interface CoreLibraryApi
```

In your code, before creating a new interface that inherits from the `CoreLibraryApi` interface, add the `@SubclassOptInRequired` annotation with a reference to the `UnstableApi` annotation class:

```
// Client code
@SubclassOptInRequired(UnstableApi::class)
interface SomeImplementation : CoreLibraryApi
```

Note that when you use the `@SubclassOptInRequired` annotation on a class, the opt-in requirement is not propagated to any inner or nested classes:

```
// Library code
@RequiresOptIn
annotation class ExperimentalFeature

@SubclassOptInRequired(ExperimentalFeature::class)
open class FileSystem {
    open class File
}

// Client code

// Opt-in is required
class NetworkFileSystem : FileSystem()

// Nested class
// No opt-in required
class TextFile : FileSystem.File()
```

Alternatively, you can opt in by using the `@OptIn` annotation. You can also use an experimental marker annotation to propagate the requirement further to any uses of the class in your code:

```
// Client code
// With @OptIn annotation
@OptInRequired(UnstableApi::class)
interface SomeImplementation : CoreLibraryApi

// With annotation referencing annotation class
// Propagates the opt-in requirement further
@UnstableApi
interface SomeImplementation : CoreLibraryApi
```

Require opt-in to use API

You can require users of your library to opt in before they are able to use your API. Additionally, you can inform users about any special conditions for using your API until you decide to remove the opt-in requirement.

Create opt-in requirement annotations

To require opt-in to use your module's API, create an annotation class to use as an **opt-in requirement annotation**. This class must be annotated with `@RequiresOptIn`:

```
@RequiresOptIn
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime
```

Opt-in requirement annotations must meet several requirements. They must have:

- `BINARY` or `RUNTIME` retention.
- No `EXPRESSION`, `FILE`, `TYPE`, or `TYPE_PARAMETER` as a target.
- No parameters.

An opt-in requirement can have one of two severity levels:

- `RequiresOptIn.Level.ERROR`. Opt-in is mandatory. Otherwise, the code that uses marked API won't compile. This is the default level.
- `RequiresOptIn.Level.WARNING`. Opt-in is not mandatory, but advisable. Without it, the compiler raises a warning.

To set the desired level, specify the `level` parameter of the `@RequiresOptIn` annotation.

Additionally, you can provide a `message` to API users. The compiler shows this message to users that try to use the API without opt-in:

```
@RequiresOptIn(level = RequiresOptIn.Level.WARNING, message = "This API is experimental. It can be
incompatibly changed in the future.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime
```

If you publish multiple independent features that require opt-in, declare an annotation for each. This makes using your API safer for your clients because they can use only the features that they explicitly accept. This also means that you can remove the opt-in requirements from features independently, which makes your API easier to maintain.

Mark API elements

To require an opt-in to use an API element, annotate its declaration with an opt-in requirement annotation:

```
@MyDateTime
class DateProvider

@MyDateTime
fun getTime(): Time {}
```

Note that for some language elements, an opt-in requirement annotation is not applicable:

- You can't annotate a backing field or a getter of a property, just the property itself.
- You can't annotate a local variable or a value parameter.

Require opt-in to extend API

There may be times when you want more granular control over which specific parts of your API can be used and extended. For example, when you have some API that is stable to use but:

- **Unstable to implement** due to ongoing evolution, such as when you have a family of interfaces where you expect to add new abstract functions without default implementations.
- **Delicate or fragile to implement**, such as individual functions that need to behave in a coordinated manner.
- **Has a contract that may be weakened in the future** in a backward-incompatible way for external implementations, such as changing an input parameter `T` to a nullable version `T?` where the code didn't previously consider `null` values.

In such cases, you can require users to opt in to your API before they can extend it further. Users can extend your API by inheriting from the API or implementing abstract functions. By using the `@SubclassOptInRequired` annotation, you can enforce this requirement to opt-in for open or abstract classes and non-functional interfaces.

To add the opt-in requirement to an API element, use the `@SubclassOptInRequired` annotation with a reference to the annotation class:

```
@RequiresOptIn(
 level = RequiresOptIn.Level.WARNING,
 message = "Interfaces in this library are experimental"
)
annotation class UnstableApi()

@SubclassOptInRequired(UnstableApi::class)
// An interface requiring opt-in to extend
interface CoreLibraryApi
```

Note that when you use the `@SubclassOptInRequired` annotation to require opt-in, the requirement is not propagated to any inner or nested classes.

For a real-world example of how to use the `@SubclassOptInRequired` annotation in your API, check out the `SharedFlow` interface in the `kotlinx.coroutines` library.

Opt-in requirements for pre-stable APIs

If you use opt-in requirements for features that are not stable yet, carefully handle the API graduation to avoid breaking client code.

Once your pre-stable API graduates and is released in a stable state, remove opt-in requirement annotations from your declarations. The clients can then use them without restriction. However, you should leave the annotation classes in modules so that the existing client code remains compatible.

To encourage API users to update their modules by removing any annotations from their code and recompiling, mark the annotations as `@Deprecated` and provide an explanation in the deprecation message.

```
@Deprecated("This opt-in requirement is not used anymore. Remove its usages from your code.")
@RequiresOptIn
annotation class ExperimentalDateTime
```

Scope functions

The Kotlin standard library contains several functions whose sole purpose is to execute a block of code within the context of an object. When you call such a function on an object with a lambda expression provided, it forms a temporary scope. In this scope, you can access the object without its name. Such functions are called *scope functions*. There are five of them: `let`, `run`, `with`, `apply`, and `also`.

Basically, these functions all perform the same action: execute a block of code on an object. What's different is how this object becomes available inside the block and what the result of the whole expression is.

Here's a typical example of how to use a scope function:

```
data class Person(var name: String, var age: Int, var city: String) {
    fun moveTo(newCity: String) { city = newCity }
    fun incrementAge() { age++ }
}

fun main() {
    Person("Alice", 20, "Amsterdam").let {
        println(it)
        it.moveTo("London")
        it.incrementAge()
        println(it)
    }
}
```

If you write the same without `let`, you'll have to introduce a new variable and repeat its name whenever you use it.

```
data class Person(var name: String, var age: Int, var city: String) {
    fun moveTo(newCity: String) { city = newCity }
    fun incrementAge() { age++ }
}

fun main() {
    val alice = Person("Alice", 20, "Amsterdam")
    println(alice)
    alice.moveTo("London")
    alice.incrementAge()
    println(alice)
}
```

Scope functions don't introduce any new technical capabilities, but they can make your code more concise and readable.

Due to the many similarities between scope functions, choosing the right one for your use case can be tricky. The choice mainly depends on your intent and the consistency of use in your project. Below, we provide detailed descriptions of the differences between scope functions and their conventions.

Function selection

To help you choose the right scope function for your purpose, we provide this table that summarizes the key differences between them.

Function	Object reference	Return value	Is extension function
`let`	`it`	Lambda result	Yes
`run`	`this`	Lambda result	Yes
`run`	-	Lambda result	No: called without the context object
`with`	`this`	Lambda result	No: takes the context object as an argument.
`apply`	`this`	Context object	Yes
`also`	`it`	Context object	Yes

Detailed information about these functions is provided in the dedicated sections below.

Here is a short guide for choosing scope functions depending on the intended purpose:

- Executing a lambda on non-nullable objects: `let`
- Introducing an expression as a variable in local scope: `let`
- Object configuration: `apply`
- Object configuration and computing the result: `run`
- Running statements where an expression is required: non-extension `run`
- Additional effects: `also`
- Grouping function calls on an object: `with`

The use cases of different scope functions overlap, so you can choose which functions to use based on the specific conventions used in your project or team.

Although scope functions can make your code more concise, avoid overusing them: it can make your code hard to read and lead to errors. We also recommend that you avoid nesting scope functions and be careful when chaining them because it's easy to get confused about the current context object and value of `this` or `it`.

Distinctions

Because scope functions are similar in nature, it's important to understand the differences between them. There are two main differences between each scope function:

- The way they refer to the context object.
- Their return value.

Context object: this or it

Inside the lambda passed to a scope function, the context object is available by a short reference instead of its actual name. Each scope function uses one of two ways to reference the context object: as a lambda receiver

(`this`) or as a lambda argument (`it`). Both provide the same capabilities, so we describe the pros and cons of each for different use cases and provide recommendations for their use.

```
fun main() {
    val str = "Hello"
    // this
    str.run {
        println("The string's length: $length")
        //println("The string's length: ${this.length}") // does the same
    }

    // it
    str.let {
        println("The string's length is ${it.length}")
    }
}
```

this

`run` , `with` , and `apply` reference the context object as a lambda receiver - by keyword `this` . Hence, in their lambdas, the object is available as it would be in ordinary class functions.

In most cases, you can omit `this` when accessing the members of the receiver object, making the code shorter. On the other hand, if `this` is omitted, it can be hard to distinguish between the receiver members and external objects or functions. So having the context object as a receiver (`this`) is recommended for lambdas that mainly operate on the object's members by calling its functions or assigning values to properties.

```
data class Person(var name: String, var age: Int = 0, var city: String = "")

fun main() {
    val adam = Person("Adam").apply {
        age = 20                       // same as this.age = 20
        city = "London"
    }
    println(adam)
}
```

it

In turn, `let` and `also` reference the context object as a lambda argument. If the argument name is not specified, the object is accessed by the implicit default name `it` . `it` is shorter than `this` and expressions with `it` are usually easier to read.

However, when calling the object's functions or properties, you don't have the object available implicitly like `this` . Hence, accessing the context object via `it` is better when the object is mostly used as an argument in function calls. `it` is also better if you use multiple variables in the code block.

```
import kotlin.random.Random

fun writeToLog(message: String) {
    println("INFO: $message")
}

fun main() {
    fun getRandomInt(): Int {
        return Random.nextInt(100).also {
            writeToLog("getRandomInt() generated value $it")
        }
    }

    val i = getRandomInt()
    println(i)
}
```

The example below demonstrates referencing the context object as a lambda argument with argument name: `value`.

```
import kotlin.random.Random

fun writeToLog(message: String) {
    println("INFO: $message")
}

fun main() {
    fun getRandomInt(): Int {
        return Random.nextInt(100).also { value ->
            writeToLog("getRandomInt() generated value $value")
        }
    }

    val i = getRandomInt()
    println(i)
}
```

Return value

Scope functions differ by the result they return:

- `apply` and `also` return the context object.
- `let`, `run`, and `with` return the lambda result.

You should consider carefully what return value you want based on what you want to do next in your code. This helps you to choose the best scope function to use.

Context object

The return value of `apply` and `also` is the context object itself. Hence, they can be included into call chains as *side steps*: you can continue chaining function calls on the same object, one after another.

```
fun main() {
    val numberList = mutableListOf<Double>()
    numberList.also { println("Populating the list") }
        .apply {
            add(2.71)
            add(3.14)
            add(1.0)
        }
        .also { println("Sorting the list") }
        .sort()
    println(numberList)
}
```

They also can be used in return statements of functions returning the context object.

```
import kotlin.random.Random

fun writeToLog(message: String) {
    println("INFO: $message")
}

fun main() {
    fun getRandomInt(): Int {
        return Random.nextInt(100).also {
            writeToLog("getRandomInt() generated value $it")
        }
    }

    val i = getRandomInt()
}
```

Lambda result

`let`, `run`, and `with` return the lambda result. So you can use them when assigning the result to a variable, chaining operations on the result, and so on.

```
fun main() {
    val numbers = mutableListOf("one", "two", "three")
    val countEndsWithE = numbers.run {
        add("four")
        add("five")
        count { it.endsWith("e") }
    }
    println("There are $countEndsWithE elements that end with e.")
}
```

Additionally, you can ignore the return value and use a scope function to create a temporary scope for local variables.

```
fun main() {
    val numbers = mutableListOf("one", "two", "three")
    with(numbers) {
        val firstItem = first()
        val lastItem = last()
        println("First item: $firstItem, last item: $lastItem")
    }
}
```

Functions

To help you choose the right scope function for your use case, we describe them in detail and provide recommendations for use. Technically, scope functions are interchangeable in many cases, so the examples show conventions for using them.

let

- **The context object** is available as an argument (`it`).
- **The return value** is the lambda result.

`let` can be used to invoke one or more functions on results of call chains. For example, the following code prints the results of two operations on a collection:

```
fun main() {
    val numbers = mutableListOf("one", "two", "three", "four", "five")
    val resultList = numbers.map { it.length }.filter { it > 3 }
    println(resultList)
}
```

With `let` , you can rewrite the above example so that you're not assigning the result of the list operations to a variable:

```
fun main() {
    val numbers = mutableListOf("one", "two", "three", "four", "five")
    numbers.map { it.length }.filter { it > 3 }.let {
        println(it)
        // and more function calls if needed
    }
}
```

If the code block passed to `let` contains a single function with `it` as an argument, you can use the method reference (`::`) instead of the lambda argument:

```
fun main() {
    val numbers = mutableListOf("one", "two", "three", "four", "five")
    numbers.map { it.length }.filter { it > 3 }.let(::println)
}
```

`let` is often used to execute a code block containing non-null values. To perform actions on a nullable object, use the safe call operator `?.` on it and call `let` with the actions in its lambda.

```
fun processNonNullString(str: String) {}

fun main() {
    val str: String? = "Hello"
    //processNonNullString(str)       // compilation error: str can be null
    val length = str?.let {
        println("let() called on $it")
        processNonNullString(it)      // OK: 'it' is not null inside '?.let { }'
        it.length
    }
}
```

You can also use `let` to introduce local variables with a limited scope to make your code easier to read. To define a new variable for the context object, provide its name as the lambda argument so that it can be used instead of the default `it`.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    val modifiedFirstItem = numbers.first().let { firstItem →
        println("The first item of the list is '$firstItem'")
        if (firstItem.length ≥ 5) firstItem else "!" + firstItem + "!"
    }.uppercase()
    println("First item after modifications: '$modifiedFirstItem'")
}
```

with

- **The context object** is available as a receiver (`this`).
- **The return value** is the lambda result.

As `with` is not an extension function: the context object is passed as an argument, but inside the lambda, it's available as a receiver (`this`).

We recommend using `with` for calling functions on the context object when you don't need to use the returned result. In code, `with` can be read as " *with this object, do the following.* "

```
fun main() {
    val numbers = mutableListOf("one", "two", "three")
    with(numbers) {
        println("'with' is called with argument $this")
        println("It contains $size elements")
    }
}
```

You can also use `with` to introduce a helper object whose properties or functions are used for calculating a value.

```
fun main() {
    val numbers = mutableListOf("one", "two", "three")
    val firstAndLast = with(numbers) {
        "The first element is ${first()}," +
        " the last element is ${last()}"
    }
    println(firstAndLast)
}
```

run

- **The context object** is available as a receiver (`this`).
- **The return value** is the lambda result.

`run` does the same as `with` but it is implemented as an extension function. So like `let`, you can call it on the context object using dot notation.

`run` is useful when your lambda both initializes objects and computes the return value.

```
class MultiportService(var url: String, var port: Int) {
    fun prepareRequest(): String = "Default request"
    fun query(request: String): String = "Result for query '$request'"
}

fun main() {
    val service = MultiportService("https://example.kotlinlang.org", 80)

    val result = service.run {
        port = 8080
        query(prepareRequest() + " to port $port")
    }

    // the same code written with let() function:
    val letResult = service.let {
        it.port = 8080
        it.query(it.prepareRequest() + " to port ${it.port}")
    }
    println(result)
    println(letResult)
}
```

You can also invoke `run` as a non-extension function. The non-extension variant of `run` has no context object, but it still returns the lambda result. Non-extension `run` lets you execute a block of several statements where an expression is required. In code, non-extension `run` can be read as " *run the code block and compute the result.* "

```
fun main() {
    val hexNumberRegex = run {
        val digits = "0-9"
        val hexDigits = "A-Fa-f"
        val sign = "+-"

        Regex("[$sign]?[$digits$hexDigits]+")
    }

    for (match in hexNumberRegex.findAll("+123 -FFFF !%*& 88 XYZ")) {
        println(match.value)
    }
}
```

apply

- **The context object** is available as a receiver (`this`).
- **The return value** is the object itself.

As `apply` returns the context object itself, we recommend that you use it for code blocks that don't return a value and that mainly operate on the members of the receiver object. The most common use case for `apply` is for object configuration. Such calls can be read as " *apply the following assignments to the object.* "

```
data class Person(var name: String, var age: Int = 0, var city: String = "")

fun main() {
    val adam = Person("Adam").apply {
        age = 32
        city = "London"
    }
    println(adam)
}
```

Another use case for `apply` is to include `apply` in multiple call chains for more complex processing.

also

- **The context object** is available as an argument (`it`).
- **The return value** is the object itself.

`also` is useful for performing some actions that take the context object as an argument. Use `also` for actions that need a reference to the object rather than its properties and functions, or when you don't want to shadow the `this` reference from an outer scope.

When you see `also` in code, you can read it as " *and also do the following with the object.* "

```
fun main() {
    val numbers = mutableListOf("one", "two", "three")
    numbers
        .also { println("The list elements before adding new one: $it") }
        .add("four")
}
```

takeIf and takeUnless

In addition to scope functions, the standard library contains the functions `takeIf` and `takeUnless` . These functions let you embed checks of an object's state in call chains.

When called on an object along with a predicate, `takeIf` returns this object if it satisfies the given predicate. Otherwise, it returns `null` . So, `takeIf` is a filtering function for a single object.

`takeUnless` has the opposite logic of `takeIf` . When called on an object along with a predicate, `takeUnless` returns `null` if it satisfies the given predicate. Otherwise, it returns the object.

When using `takeIf` or `takeUnless` , the object is available as a lambda argument (`it`).

```
import kotlin.random.*

fun main() {
    val number = Random.nextInt(100)

    val evenOrNull = number.takeIf { it % 2 == 0 }
    val oddOrNull = number.takeUnless { it % 2 == 0 }
    println("even: $evenOrNull, odd: $oddOrNull")
}
```

When chaining other functions after `takeIf` and `takeUnless` , don't forget to perform a null check or use a safe call (`?.`) because their return value is nullable.

```
fun main() {
    val str = "Hello"
    val caps = str.takeIf { it.isNotEmpty() }?.uppercase()
   //val caps = str.takeIf { it.isNotEmpty() }.uppercase() //compilation error
    println(caps)
}
```

`takeIf` and `takeUnless` are especially useful in combination with scope functions. For example, you can chain `takeIf` and `takeUnless` with `let` to run a code block on objects that match the given predicate. To do this, call `takeIf` on the object and then call `let` with a safe call (`?`). For objects that don't match the predicate, `takeIf` returns `null` and `let` isn't invoked.

```
fun main() {
    fun displaySubstringPosition(input: String, sub: String) {
        input.indexOf(sub).takeIf { it ≥ 0 }?.let {
            println("The substring $sub is found in $input.")
            println("Its start position is $it.")
        }
    }

    displaySubstringPosition("010000011", "11")
    displaySubstringPosition("010000011", "12")
}
```

For comparison, below is an example of how the same function can be written without using `takeIf` or scope functions:

```
fun main() {
    fun displaySubstringPosition(input: String, sub: String) {
        val index = input.indexOf(sub)
        if (index ≥ 0) {
            println("The substring $sub is found in $input.")
            println("Its start position is $index.")
        }
    }

    displaySubstringPosition("010000011", "11")
    displaySubstringPosition("010000011", "12")
}
```

Time measurement

The Kotlin standard library gives you the tools to calculate and measure time in different units. Accurate time measurement is important for activities like:

- Managing threads or processes
- Collecting statistics
- Detecting timeouts
- Debugging

By default, time is measured using a monotonic time source, but other time sources can be configured. For more information, see Create time source.

Calculate duration

To represent an amount of time, the standard library has the `Duration` class. A `Duration` can be expressed in the following units from the `DurationUnit` enum class:

- `NANOSECONDS`
- `MICROSECONDS`

- `MILLISECONDS`
- `SECONDS`
- `MINUTES`
- `HOURS`
- `DAYS`

A `Duration` can be positive, negative, zero, positive infinity, or negative infinity.

Create duration

To create a `Duration`, use the extension properties available for `Int`, `Long`, and `Double` types: `nanoseconds`, `microseconds`, `milliseconds`, `seconds`, `minutes`, `hours`, and `days`.

> Days refer to periods of 24 hours. They are not calendar days.

For example:

```
import kotlin.time.*
import kotlin.time.Duration.Companion.nanoseconds
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.Duration.Companion.seconds
import kotlin.time.Duration.Companion.minutes
import kotlin.time.Duration.Companion.days

fun main() {
    val fiveHundredMilliseconds: Duration = 500.milliseconds
    val zeroSeconds: Duration = 0.seconds
    val tenMinutes: Duration = 10.minutes
    val negativeNanosecond: Duration = (-1).nanoseconds
    val infiniteDays: Duration = Double.POSITIVE_INFINITY.days
    val negativeInfiniteDays: Duration = Double.NEGATIVE_INFINITY.days

    println(fiveHundredMilliseconds) // 500ms
    println(zeroSeconds)             // 0s
    println(tenMinutes)              // 10m
    println(negativeNanosecond)      // -1ns
    println(infiniteDays)            // Infinity
    println(negativeInfiniteDays)    // -Infinity
}
```

You can also perform basic arithmetic with `Duration` objects:

```
import kotlin.time.*
import kotlin.time.Duration.Companion.seconds

fun main() {
    val fiveSeconds: Duration = 5.seconds
    val thirtySeconds: Duration = 30.seconds

    println(fiveSeconds + thirtySeconds)
    // 35s
    println(thirtySeconds - fiveSeconds)
    // 25s
    println(fiveSeconds * 2)
    // 10s
    println(thirtySeconds / 2)
    // 15s
    println(thirtySeconds / fiveSeconds)
    // 6.0
    println(-thirtySeconds)
    // -30s
    println((-thirtySeconds).absoluteValue)
    // 30s
}
```

Get string representation

It can be useful to have a string representation of a `Duration` so that you can print, serialize, transfer, or store it.

To get a string representation, use the `.toString()` function. By default, the time is reported using each unit that is present. For example: `1h 0m 45.677s` or `-(6d 5h 5m 28.284s)`

To configure the output, use the `.toString()` function with your desired `DurationUnit` and number of decimal places as function parameters:

```
import kotlin.time.Duration
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.DurationUnit

fun main() {
    // Print in seconds with 2 decimal places
    println(5887.milliseconds.toString(DurationUnit.SECONDS, 2))
    // 5.89s
}
```

To get an ISO-8601-compatible string, use the `toIsoString()` function:

```
import kotlin.time.Duration.Companion.seconds

fun main() {
    println(86420.seconds.toIsoString()) // PT24H0M20S
}
```

Convert duration

To convert your `Duration` into a different `DurationUnit`, use the following properties:

- `inWholeNanoseconds`
- `inWholeMicroseconds`

- `inWholeSeconds`
- `inWholeMinutes`
- `inWholeHours`
- `inWholeDays`

For example:

```
import kotlin.time.Duration
import kotlin.time.Duration.Companion.minutes

fun main() {
    val thirtyMinutes: Duration = 30.minutes
    println(thirtyMinutes.inWholeSeconds)
    // 1800
}
```

Alternatively, you can use your desired `DurationUnit` as a function parameter in the following extension functions:

- `.toInt()`
- `.toDouble()`
- `.toLong()`

For example:

```
import kotlin.time.Duration.Companion.seconds
import kotlin.time.DurationUnit

fun main() {
    println(270.seconds.toDouble(DurationUnit.MINUTES))
    // 4.5
}
```

Compare duration

To check if `Duration` objects are equal, use the equality operator (`==`):

```
import kotlin.time.Duration
import kotlin.time.Duration.Companion.hours
import kotlin.time.Duration.Companion.minutes

fun main() {
    val thirtyMinutes: Duration = 30.minutes
    val halfHour: Duration = 0.5.hours
    println(thirtyMinutes == halfHour)
    // true
}
```

To compare `Duration` objects, use the comparison operators (`<` , `>`):

```
import kotlin.time.Duration.Companion.microseconds
import kotlin.time.Duration.Companion.nanoseconds

fun main() {
    println(3000.microseconds < 25000.nanoseconds)
    // false
}
```

Break duration into components

To break down a `Duration` into its time components and perform a further action, use the overload of the `toComponents()` function. Add your desired action as a function or lambda expression as a function parameter.

For example:

```
import kotlin.time.Duration
import kotlin.time.Duration.Companion.minutes

fun main() {
    val thirtyMinutes: Duration = 30.minutes
    println(thirtyMinutes.toComponents { hours, minutes, _, _ -> "${hours}h:${minutes}m" })
    // 0h:30m
}
```

In this example, the lambda expression has `hours` and `minutes` as function parameters with underscores (`_`) for the unused `seconds` and `nanoseconds` parameters. The expression returns a concatenated string using string templates to get the desired output format of `hours` and `minutes` .

Measure time

To track the passage of time, the standard library provides tools so that you can easily:

- Measure the time taken to execute some code with your desired time unit.
- Mark a moment in time.
- Compare and subtract two moments in time.
- Check how much time has passed since a specific moment in time.
- Check whether the current time has passed a specific moment in time.

Measure code execution time

To measure the time taken to execute a block of code, use the `measureTime` inline function:

```
import kotlin.time.measureTime

fun main() {
    val timeTaken = measureTime {
        Thread.sleep(100)
    }
    println(timeTaken) // e.g. 103 ms
}
```

To measure the time taken to execute a block of code **and** return the value of the block of code, use inline function `measureTimedValue` .

For example:

```
import kotlin.time.measureTimedValue

fun main() {
    val (value, timeTaken) = measureTimedValue {
        Thread.sleep(100)
        42
    }
    println(value)     // 42
    println(timeTaken) // e.g. 103 ms
}
```

By default, both functions use a monotonic time source.

Mark moments in time

To mark a specific moment in time, use the `TimeSource` interface and the `markNow()` function to create a `TimeMark`:

```
import kotlin.time.*

fun main() {
   val timeSource = TimeSource.Monotonic
   val mark = timeSource.markNow()
}
```

Measure differences in time

To measure differences between `TimeMark` objects from the same time source, use the subtraction operator (`-`).

To compare `TimeMark` objects from the same time source, use the comparison operators (`<`, `>`).

For example:

```
import kotlin.time.*

fun main() {
   val timeSource = TimeSource.Monotonic
   val mark1 = timeSource.markNow()
   Thread.sleep(500) // Sleep 0.5 seconds.
   val mark2 = timeSource.markNow()

   repeat(4) { n ->
       val mark3 = timeSource.markNow()
       val elapsed1 = mark3 - mark1
       val elapsed2 = mark3 - mark2

       println("Measurement 1.${n + 1}: elapsed1=$elapsed1, elapsed2=$elapsed2, diff=${elapsed1 -
elapsed2}")
   }

   println(mark2 > mark1) // This is true, as mark2 was captured later than mark1.
   // true
}
```

To check if a deadline has passed or a timeout has been reached, use the `hasPassedNow()` and `hasNotPassedNow()` extension functions:

```
import kotlin.time.*
import kotlin.time.Duration.Companion.seconds

fun main() {
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    val fiveSeconds: Duration = 5.seconds
    val mark2 = mark1 + fiveSeconds

    // It hasn't been 5 seconds yet
    println(mark2.hasPassedNow())
    // false

    // Wait six seconds
    Thread.sleep(6000)
    println(mark2.hasPassedNow())
    // true

}
```

Time sources

By default, time is measured using a monotonic time source. Monotonic time sources only move forward and are not affected by variations like timezones. An alternative to monotonic time is elapsed real time, which is also known as wall-clock time. Elapsed real time is measured relative to another point in time.

Default time sources per platform

This table explains the default source of monotonic time for each platform:

Platform	Source
Kotlin/JVM	`System.nanoTime()`
Kotlin/JS (Node.js)	`process.hrtime()`
Kotlin/JS (browser)	`window.performance.now()` or `Date.now()`
Kotlin/Native	`std::chrono::high_resolution_clock` or `std::chrono::steady_clock`

Create time source

There are some cases where you might want to use a different time source. For example, in Android, `System.nanoTime()` only counts time while the device is active. It loses track of time when the device enters deep sleep. To keep track of time while the device is in deep sleep, you can create a time source that uses `SystemClock.elapsedRealtimeNanos()`:

```
object RealtimeMonotonicTimeSource : AbstractLongTimeSource(DurationUnit.NANOSECONDS) {
    override fun read(): Long = SystemClock.elapsedRealtimeNanos()
}
```

Then you can use your time source to make time measurements:

```
fun main() {
    val elapsed: Duration = RealtimeMonotonicTimeSource.measureTime {
        Thread.sleep(100)
    }
    println(elapsed) // e.g. 103 ms
}
```

For more information about the `kotlin.time` package, see our standard library API reference.

UUIDs

The `Uuid` class represents Universally Unique Identifiers (UUIDs), also known as Globally Unique Identifiers (GUIDs).

A `Uuid` is a 128-bit value used to uniquely identify an entity without relying on a central system that assigns IDs. This makes UUIDs useful in distributed applications, databases, client-generated records, or Kotlin Multiplatform applications.

Use the `Uuid` class to work with UUID values. Unlike plain strings, a dedicated UUID type makes your code more explicit and prevents accidental use of invalid values.

To use UUIDs in your project, import the `Uuid` class from the `kotlin.uuid` package:

```
import kotlin.uuid.Uuid
```

Generate UUIDs

To generate a random version 4 UUID for regular identifiers, such as user or database IDs, use the `Uuid.random()` function:

```
import kotlin.uuid.Uuid

fun main() {
//sampleStart
    val id = Uuid.random()
    println(id)
}
```

You can also generate specific versions of UUIDs with the following Experimental functions:

- The `Uuid.generateV4()` function generates the same type of UUID as the `Uuid.random()` function but explicitly states that the value is a version 4 UUID.
- The `Uuid.generateV7()` function generates a version 7 UUID with a timestamp that you can use for UUID sorting.
- The `Uuid.generateV7NonMonotonicAt()` function generates a version 7 UUID for a specific moment in time.

These UUID generation functions are Experimental. To opt in, use the `@OptIn(ExperimentalUuidApi::class)` annotation or add the following compiler option to your build file:

Gradle

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-opt-in=kotlin.uuid.ExperimentalUuidApi")
    }
}
```

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-opt-in=kotlin.uuid.ExperimentalUuidApi</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Here's an example that generates version-specific UUIDs:

```
import kotlin.time.Instant
import kotlin.time.ExperimentalTime
import kotlin.uuid.Uuid

@OptIn(kotlin.uuid.ExperimentalUuidApi::class, ExperimentalTime::class)
fun main() {
    // Generates a version 4 UUID
    val idVersion4 = Uuid.generateV4()
    println(idVersion4)

    // Generates a version 7 UUID
    val idVersion7 = Uuid.generateV7()
    println(idVersion7)

    // Generates a version 7 UUID for the specified timestamp
    val timestamp = Instant.fromEpochMilliseconds(1757440583000L)
    val idVersion7SpecificTime = Uuid.generateV7NonMonotonicAt(timestamp)
    println(idVersion7SpecificTime)
}
```

Parse UUIDs

UUID values are often represented as strings, such as in URL parameters or database records.

To convert a `String` value to a `Uuid` value, use the `Uuid.parse()` function:

```
import kotlin.uuid.Uuid

fun main() {
    val id = Uuid.parse("de2bc56c-ea73-4f3c-8a37-5a46fdb2d79a")
    println(id)
}
```

The `Uuid.parse()` function accepts both the standard hex-and-dash format and the hexadecimal format without dashes.

If the input is invalid, the `Uuid.parse()` function throws an `IllegalArgumentException`:

```
import kotlin.uuid.Uuid

fun main() {
//sampleStart
    val id = Uuid.parse("10")
    println(id)
}
```

If your application accepts only one representation, use the format-specific functions:

- `Uuid.parseHexDash()` for the hex-and-dash string representation.
- `Uuid.parseHex()` for the hexadecimal string representation without dashes.

For example:

```
import kotlin.uuid.Uuid

fun main() {
//sampleStart
    val standard = Uuid.parseHexDash("de2bc56c-ea73-4f3c-8a37-5a46fdb2d79a")
    val compact = Uuid.parseHex("de2bc56cea734f3c8a375a46fdb2d79a")

    println(standard)
    println(compact)
}
```

If you have UUIDs from external sources and must handle invalid input safely, use `Uuid.parseOrNull()`, `Uuid.parseHexDashOrNull()`, or `Uuid.parseHexOrNull()`. These functions return `null` if the input is invalid:

```
fun parseId(input: String): Uuid? {
    return Uuid.parseOrNull(input)
}
```

Convert UUIDs to strings

You can convert a `Uuid` value to a `String` value with the following functions:

- `toString()` for the standard string representation
- `toHexDashString()` for the hex-and-dash format
- `toHexString()` for the hexadecimal format without dashes

For example:

```
import kotlin.uuid.Uuid

fun main() {
//sampleStart
    val id = Uuid.parse("de2bc56c-ea73-4f3c-8a37-5a46fdb2d79a")

    println(id.toString())
    // de2bc56c-ea73-4f3c-8a37-5a46fdb2d79a
    println(id.toHexDashString())
    // de2bc56c-ea73-4f3c-8a37-5a46fdb2d79a
    println(id.toHexString())
    // de2bc56cea734f3c8a375a46fdb2d79a
}
```

Compare UUIDs

You can check whether `Uuid` values are equal using the `==` operator.

Kotlin compares values according to the UUID value, not to the textual representation. For example, two values in different forms are equal if they represent the same 128-bit value:

```
import kotlin.uuid.Uuid

fun main() {
//sampleStart
    val first = Uuid.parse("de2bc56c-ea73-4f3c-8a37-5a46fdb2d79a")
    val second = Uuid.parse("de2bc56cea734f3c8a375a46fdb2d79a")

    println(first == second)
    // true
}
```

This makes the `Uuid` comparison more reliable than comparing strings, which treats the same value in different formats as different values. `Uuid` comparison checks the actual identifier value.

`Uuid` implements the `Comparable<Uuid>` interface, so UUID values can be sorted with standard collection functions such as `sorted()`. In that case, Kotlin compares values lexicographically (from the most to the least significant bit):

```
import kotlin.uuid.Uuid

fun main() {
//sampleStart
    val first = Uuid.generateV7()
    val second = Uuid.generateV7()

    val sorted = listOf(first, second).sorted()
    println(sorted)
}
```

Work with binary representations

Some APIs, storage formats, and binary protocols do not represent UUIDs as strings. Instead, they store the 128-bit UUID value as either:

- A 16-byte array

- Two 64-bit values

Use these representations when you need to exchange UUIDs with systems that expect binary UUID data.

To convert between a UUID and a 16-byte representation, use the `.toByteArray()` and `Uuid.fromByteArray()` functions:

```
import kotlin.uuid.Uuid

fun main() {
//sampleStart
    val id = Uuid.random()

    val bytes = id.toByteArray()
    val original = Uuid.fromByteArray(bytes)

    println(id)

    println(bytes)
    println(original)

    println(id == original)
    // true
}
```

You can also represent the same 128-bit UUID value as two `Long` values. This is useful because Kotlin doesn't provide a built-in 128-bit integer type. The two `Long` values store the UUID in two parts:

- The `mostSignificantBits` parameter for the first 64 bits of a UUID.
- The `leastSignificantBits` parameter for the last 64 bits of a UUID.

To create a `Uuid` value from two `Long` values, use the `Uuid.fromLongs()` function:

```
import kotlin.uuid.Uuid

fun main() {
//sampleStart
    val id = Uuid.fromLongs(
        mostSignificantBits = -4653685776373167443,
        leastSignificantBits = -6288180676521310383.toLong()
    )
    println(id)
    // bf6ac971-52fd-4aad-a8bb-e4fdac78c751
}
```

To extract the two parts from an existing `Uuid` value, use the `Uuid.toLongs()` function:

```
import kotlin.uuid.Uuid

fun main() {
//sampleStart
    val id = Uuid.random()

    id.toLongs { mostSignificantBits, leastSignificantBits ->
        println(mostSignificantBits)
        println(leastSignificantBits)
    }
}
```

Serialize UUIDs

Kotlin supports serialization for `Uuid` values. Use it to store or transfer a UUID value outside Kotlin code, for example, in JSON APIs or configuration files.

To serialize a `Uuid` value, represent it as a string unless your application requires another format. The `kotlinx.serialization` library uses the hex-and-dash format:

```
//sampleStart
import kotlin.uuid.Uuid
import kotlinx.serialization.Serializable
import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

@Serializable
data class User(
    val id: Uuid,
    val name: String
)

fun main() {
    val user = User(
        id = Uuid.parse("de2bc56cea734f3c8a375a46fdb2d79a"),
        name = "Kotlin"
    )

    println(Json.encodeToString(user))
    // {"id":"de2bc56c-ea73-4f3c-8a37-5a46fdb2d79a","name":"Kotlin"}
}
```

Use UUIDs with Java APIs

Java uses the `java.util.UUID` class to represent UUIDs. On the JVM, Java APIs may accept or return this type. Although `java.util.UUID` and `kotlin.uuid.Uuid` both represent UUIDs, they are two distinct types.

Convert values explicitly to pass UUIDs between Kotlin and Java:

- Convert a Java UUID to Kotlin with the `.toKotlinUuid()` extension function:

  ```
  import kotlin.uuid.toKotlinUuid

  val kotlinId: Uuid = javaId.toKotlinUuid()
  ```

- Convert a Kotlin UUID to Java with the `.toJavaUuid()` extension function:

  ```
  import kotlin.uuid.toJavaUuid

  val javaId: java.util.UUID = kotlinId.toJavaUuid()
  ```

These functions allow you to represent your UUID values using `Uuid` at JVM interoperability boundaries.

The `java.util.UUID` and `kotlin.uuid.Uuid` classes are comparable, but the ordering may differ. Make sure to check your code that depends on UUID ordering before migrating from the Java API to the Kotlin API.

Kotlin also provides support for working with Java buffers. Use JVM-specific functions to work with UUIDs in a `ByteBuffer`:

- Use the `.getUuid()` function to read a UUID from a buffer.
- Use the `.putUuid()` function to write a UUID to a buffer.

Coroutines guide

Kotlin provides only minimal low-level APIs in its standard library to enable other libraries to utilize coroutines. Unlike many other languages with similar capabilities, `async` and `await` are not keywords in Kotlin and are not even part of its standard library. Moreover, Kotlin's concept of *suspending function* provides a safer and less error-prone abstraction for asynchronous operations than futures and promises.

`kotlinx.coroutines` is a rich library for coroutines developed by JetBrains. It contains a number of high-level coroutine-enabled primitives that this guide covers, including `launch`, `async`, and others.

This is a guide about the core features of `kotlinx.coroutines` with a series of examples, divided up into different topics.

In order to use coroutines as well as follow the examples in this guide, you need to add a dependency on the `kotlinx-coroutines-core` module as explained in the project README.

Table of contents

- Coroutines basics
- Cancellation and timeouts
- Flows
- Tutorial: Intro to coroutines and channels
- Composing suspending functions
- Coroutine context and dispatchers
- Channels
- Coroutine exceptions handling
- Shared mutable state and concurrency
- Select expression (experimental)
- Tutorial: Debug coroutines using IntelliJ IDEA
- Tutorial: Debug Kotlin Flow using IntelliJ IDEA

Additional references

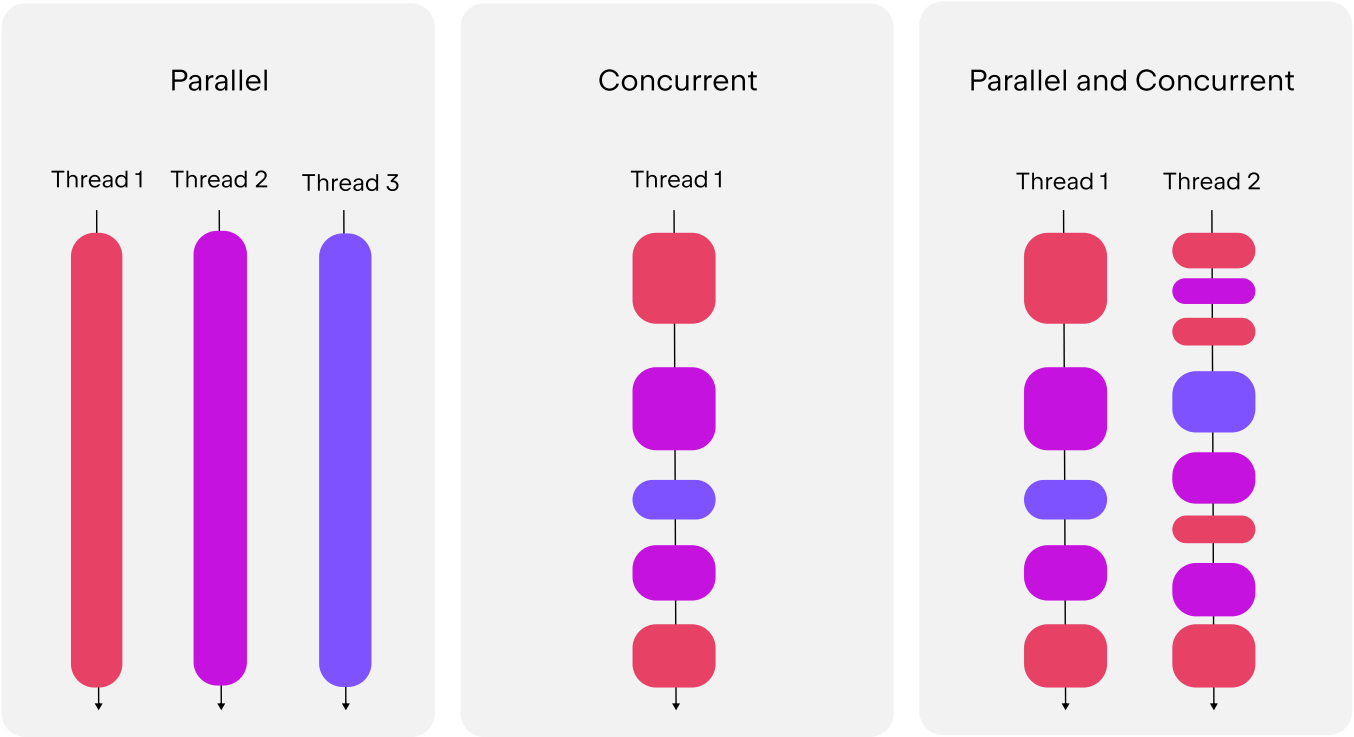
- Guide to UI programming with coroutines
- Coroutines design document (KEEP)
- Full kotlinx.coroutines API reference
- Best practices for coroutines in Android

- Additional Android resources for Kotlin coroutines and flow

Coroutines basics

To create applications that perform multiple tasks at once, a concept known as concurrency, Kotlin uses *coroutines*. A coroutine is a suspendable computation that lets you write concurrent code in a clear, sequential style. Coroutines can run concurrently with other coroutines and potentially in parallel.

On the JVM and in Kotlin/Native, all concurrent code, such as coroutines, runs on *threads*, managed by the operating system. Coroutines can suspend their execution instead of blocking a thread. This allows one coroutine to suspend while waiting for some data to arrive and another coroutine to run on the same thread, ensuring effective resource utilization.



Comparing parallel and concurrent threads

For more information about the differences between coroutines and threads, see Comparing coroutines and JVM threads.

Suspending functions

The most basic building block of coroutines is the *suspending function*. It allows a running operation to pause and resume later without affecting the structure of your code.

To declare a suspending function, use the `suspend` keyword:

```
suspend fun greet() {
    println("Hello world from a suspending function")
}
```

You can only call a suspending function from another suspending function. To call suspending functions at the entry point of a Kotlin application, mark the `main()` function with the `suspend` keyword:

```
suspend fun main() {
    showUserInfo()
}

suspend fun showUserInfo() {
    println("Loading user...")
    greet()
    println("User: John Smith")
}

suspend fun greet() {
    println("Hello world from a suspending function")
}
```

This example doesn't use concurrency yet, but by marking the functions with the `suspend` keyword, you allow them to call other suspending functions and run concurrent code inside.

While the `suspend` keyword is part of the core Kotlin language, most coroutine features are available through the `kotlinx.coroutines` library.

Add the kotlinx.coroutines library to your project

To include the `kotlinx.coroutines` library in your project, add the corresponding dependency configuration based on your build tool:

Kotlin

```
// build.gradle.kts
repositories {
    mavenCentral()
}

dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0")
}
```

Groovy

```
// build.gradle
repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0'
}
```

Maven

```
<!-- pom.xml -->
<project>
    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlinx</groupId>
            <artifactId>kotlinx-coroutines-core</artifactId>
            <version>1.11.0</version>
        </dependency>
    </dependencies>
    ...
</project>
```

Create your first coroutines

The examples on this page use the explicit `this` expression with the coroutine builder functions `CoroutineScope.launch()` and `CoroutineScope.async()`. These coroutine builders are extension functions on `CoroutineScope`, and the `this` expression refers to the current `CoroutineScope` as the receiver.

For a practical example, see Extract coroutine builders from the coroutine scope.

To create a coroutine in Kotlin, you need the following:

- A suspending function.
- A coroutine scope in which it can run, for example inside the `withContext()` function.
- A coroutine builder like `CoroutineScope.launch()` to start it.
- A dispatcher to control which threads it uses.

Let's look at an example that uses multiple coroutines in a multithreaded environment:

1. Import the `kotlinx.coroutines` library:

```
import kotlinx.coroutines.*
```

2. Mark functions that can pause and resume with the `suspend` keyword:

```
suspend fun greet() {
    println("The greet() on the thread: ${Thread.currentThread().name}")
}

suspend fun main() {}
```

While you can mark the `main()` function as `suspend` in some projects, it may not be possible when integrating with existing code or using a framework. In that case, check the framework's documentation to see if it supports calling suspending functions. If not, use `runBlocking()` to call them by blocking the current thread.

3. Add the `delay()` function to simulate a suspending task, such as fetching data or writing to a database:

```
suspend fun greet() {
    println("The greet() on the thread: ${Thread.currentThread().name}")
    delay(1000L)
}
```

4. Use `withContext(Dispatchers.Default)` to define an entry point for multithreaded concurrent code that runs on a shared thread pool:

```
suspend fun main() {
    withContext(Dispatchers.Default) {
        // Add the coroutine builders here
    }
}
```

> The suspending `withContext()` function is typically used for context switching, but in this example, it also defines a non-blocking entry point for concurrent code. It uses the `Dispatchers.Default` dispatcher to run code on a shared thread pool for multithreaded execution. By default, this pool uses up to as many threads as there are CPU cores available at runtime, with a minimum of two threads.
>
> The coroutines launched inside the `withContext()` block share the same coroutine scope, which ensures structured concurrency.

5. Use a coroutine builder function like `CoroutineScope.launch()` to start the coroutine:

```
suspend fun main() {
    withContext(Dispatchers.Default) { // this: CoroutineScope
        // Starts a coroutine inside the scope with CoroutineScope.launch()
        this.launch { greet() }
        println("The withContext() on the thread: ${Thread.currentThread().name}")
    }
}
```

6. Combine these pieces to run multiple coroutines at the same time on a shared pool of threads:

```
// Imports the coroutines library
import kotlinx.coroutines.*

// Imports the kotlin.time.Duration to express duration in seconds
import kotlin.time.Duration.Companion.seconds

// Defines a suspending function
suspend fun greet() {
    println("The greet() on the thread: ${Thread.currentThread().name}")
    // Suspends for 1 second and releases the thread
    delay(1.seconds)
    // The delay() function simulates a suspending API call here
    // You can add suspending API calls here like a network request
}

suspend fun main() {
    // Runs the code inside this block on a shared thread pool
    withContext(Dispatchers.Default) { // this: CoroutineScope
        this.launch() {
            greet()
        }

        // Starts another coroutine
        this.launch() {
            println("The CoroutineScope.launch() on the thread: ${Thread.currentThread().name}")
            delay(1.seconds)
            // The delay function simulates a suspending API call here
            // You can add suspending API calls here like a network request
        }

        println("The withContext() on the thread: ${Thread.currentThread().name}")
    }
}
```

Try running the example multiple times. You may notice that the output order and thread names may change each time you run the program, because the OS decides when threads run.

> You can display coroutine names next to thread names in the output of your code for additional information. To do so, pass the `-Dkotlinx.coroutines.debug` VM option in your build tool or IDE run configuration.
>
> See Debug coroutines for more information.

Coroutine scope and structured concurrency

When you run many coroutines in an application, you need a way to manage them as groups. Kotlin coroutines rely on a principle called *structured concurrency* to provide this structure.

According to this principle, coroutines form a tree hierarchy of parent and child tasks with linked lifecycles. A coroutine's lifecycle is the sequence of states from its creation until completion, failure, or cancellation.

A parent coroutine waits for its children to complete before it finishes. If the parent coroutine fails or gets canceled, all its child coroutines are recursively canceled too. Keeping coroutines connected this way makes cancellation and error handling predictable and safe.

To maintain structured concurrency, new coroutines can only be launched in a `CoroutineScope` that defines and manages their lifecycle. The `CoroutineScope` includes the *coroutine context*, which defines the dispatcher and

other execution properties. When you start a coroutine inside another coroutine, it automatically becomes a child of its parent scope.

Calling a coroutine builder function, such as `CoroutineScope.launch()` on a `CoroutineScope`, starts a child coroutine of the coroutine associated with that scope. Inside the builder's block, the receiver is a nested `CoroutineScope`, so any coroutines you launch there become its children.

Create a coroutine scope with the `coroutineScope()` function

To create a new coroutine scope with the current coroutine context, use the `coroutineScope()` function. This function creates a root coroutine of the coroutine subtree. It's the direct parent of coroutines launched inside the block and the indirect parent of any coroutines they launch. `coroutineScope()` executes the suspending block and waits until the block and any coroutines launched in it complete.

Here's an example:

```
// Imports the kotlin.time.Duration to express duration in seconds
import kotlin.time.Duration.Companion.seconds

import kotlinx.coroutines.*

// If the coroutine context doesn't specify a dispatcher,
// CoroutineScope.launch() uses Dispatchers.Default
suspend fun main() {
    // Root of the coroutine subtree
    coroutineScope { // this: CoroutineScope
        this.launch {
            this.launch {
                delay(2.seconds)
                println("Child of the enclosing coroutine completed")
            }
            println("Child coroutine 1 completed")
        }
        this.launch {
            delay(1.seconds)
            println("Child coroutine 2 completed")
        }
    }
    // Runs only after all children in the coroutineScope have completed
    println("Coroutine scope completed")
}
```

Since no dispatcher is specified in this example, the `CoroutineScope.launch()` builder functions in the `coroutineScope()` block inherit the current context. If that context doesn't have a specified dispatcher, `CoroutineScope.launch()` uses `Dispatchers.Default`, which runs on a shared pool of threads.

Extract coroutine builders from the coroutine scope

In some cases, you may want to extract coroutine builder calls, such as `CoroutineScope.launch()`, into separate functions.

Consider the following example:

```
suspend fun main() {
    coroutineScope { // this: CoroutineScope
        // Calls CoroutineScope.launch() where CoroutineScope is the receiver
        this.launch { println("1") }
        this.launch { println("2") }
    }
}
```

You can also write `this.launch` without the explicit `this` expression, as `launch`. These examples use explicit `this` expressions to highlight that it's an extension function on `CoroutineScope`.

For more information on how lambdas with receivers work in Kotlin, see Function literals with receiver.

The `coroutineScope()` function takes a lambda with a `CoroutineScope` receiver. Inside this lambda, the implicit receiver is a `CoroutineScope`, so builder functions like `CoroutineScope.launch()` and `CoroutineScope.async()` resolve as extension functions on that receiver.

To extract the coroutine builders into another function, that function must declare a `CoroutineScope` receiver, otherwise a compilation error occurs:

```
import kotlinx.coroutines.*
suspend fun main() {
    coroutineScope {
        launchAll()
    }
}

fun CoroutineScope.launchAll() { // this: CoroutineScope
    // Calls .launch() on CoroutineScope
    this.launch { println("1") }
    this.launch { println("2") }
}
/* -- Calling launch without declaring CoroutineScope as the receiver results in a compilation error -
-

fun launchAll() {
    // Compilation error: this is not defined
    this.launch { println("1") }
    this.launch { println("2") }
}
 */
```

In this example, the `launchAll()` function doesn't need the `suspend` keyword, because it only starts the coroutines in the current `CoroutineScope` and then returns immediately. Only mark functions with `suspend` when it makes sense for them to pause and resume before returning.

Coroutine builder functions

A coroutine builder function is a function that accepts a `suspend` lambda that defines a coroutine to run. Here are some examples:

- `CoroutineScope.launch()`
- `CoroutineScope.async()`
- `runBlocking()`

- `withContext()`
- `coroutineScope()`

Coroutine builder functions require a `CoroutineScope` to run in. This can be an existing scope or one you create with helper functions such as `coroutineScope()`, `runBlocking()`, or `withContext()`. Each builder defines how the coroutine starts and how you interact with its result.

`CoroutineScope.launch()`

The `CoroutineScope.launch()` coroutine builder function is an extension function on `CoroutineScope`. It starts a new coroutine without blocking the rest of the scope, inside an existing coroutine scope.

Use `CoroutineScope.launch()` to run a task alongside other work when the result isn't needed or you don't want to wait for it:

```
// Imports the kotlin.time.Duration to enable expressing duration in milliseconds
import kotlin.time.Duration.Companion.milliseconds

import kotlinx.coroutines.*

suspend fun main() {
    withContext(Dispatchers.Default) {
        performBackgroundWork()
    }
}

suspend fun performBackgroundWork() = coroutineScope { // this: CoroutineScope
    // Starts a coroutine that runs without blocking the scope
    this.launch {
        // Suspends to simulate background work
        delay(100.milliseconds)
        println("Sending notification in background")
    }

    // Main coroutine continues while a previous one suspends
    println("Scope continues")
}
```

After running this example, you can see that the `main()` function isn't blocked by `CoroutineScope.launch()` and keeps running other code while the coroutine works in the background.

> The `CoroutineScope.launch()` function returns a `Job` handle. Use this handle to wait for the launched coroutine to complete. For more information, see Cancellation and timeouts.

`CoroutineScope.async()`

The `CoroutineScope.async()` coroutine builder function is an extension function on `CoroutineScope`. It starts a concurrent computation inside an existing coroutine scope and returns a `Deferred` handle that represents an eventual result. Use the `.await()` function to suspend the code until the result is ready:

```
// Imports the kotlin.time.Duration to enable expressing duration in milliseconds
import kotlin.time.Duration.Companion.milliseconds

import kotlinx.coroutines.*

suspend fun main() = withContext(Dispatchers.Default) { // this: CoroutineScope
    // Starts downloading the first page
    val firstPage = this.async {
        delay(50.milliseconds)
        "First page"
    }

    // Starts downloading the second page in parallel
    val secondPage = this.async {
        delay(100.milliseconds)
        "Second page"
    }

    // Awaits both results and compares them
    val pagesAreEqual = firstPage.await() == secondPage.await()
    println("Pages are equal: $pagesAreEqual")
}
```

`runBlocking()`

The `runBlocking()` coroutine builder function creates a coroutine scope and blocks the current thread until the coroutines launched in that scope finish.

Use `runBlocking()` only when there is no other option to call suspending code from non-suspending code:

```
import kotlin.time.Duration.Companion.milliseconds
import kotlinx.coroutines.*

// A third-party interface you can't change
interface Repository {
    fun readItem(): Int
}

object MyRepository : Repository {
    override fun readItem(): Int {
        // Bridges to a suspending function
        return runBlocking {
            myReadItem()
        }
    }
}

suspend fun myReadItem(): Int {
    delay(100.milliseconds)
    return 4
}
```

Coroutine dispatchers

A *coroutine dispatcher* controls which thread or thread pool coroutines use for their execution. Coroutines aren't always tied to a single thread. They can pause on one thread and resume on another, depending on the dispatcher. This lets you run many coroutines at the same time without allocating a separate thread for every coroutine.

Even though coroutines can suspend and resume on different threads, values written before the coroutine suspends are still guaranteed to be available within the same coroutine when it resumes.

A dispatcher works together with the coroutine scope to define when coroutines run and where they run. While the coroutine scope controls the coroutine's lifecycle, the dispatcher controls which threads are used for execution.

You don't have to specify a dispatcher for every coroutine. By default, coroutines inherit the dispatcher from their parent scope. You can specify a dispatcher to run a coroutine in a different context.

If the coroutine context doesn't include a dispatcher, coroutine builders use `Dispatchers.Default`.

The `kotlinx.coroutines` library includes different dispatchers for different use cases. For example, `Dispatchers.Default` runs coroutines on a shared pool of threads, performing work in the background, separate from the main thread. This makes it an ideal choice for CPU-intensive operations like data processing.

To specify a dispatcher for a coroutine builder like `CoroutineScope.launch()`, pass it as an argument:

```
suspend fun runWithDispatcher() = coroutineScope { // this: CoroutineScope
    this.launch(Dispatchers.Default) {
        println("Running on ${Thread.currentThread().name}")
    }
}
```

Alternatively, you can use a `withContext()` block to run all code in it on a specified dispatcher:

```
// Imports the kotlin.time.Duration to enable expressing duration in milliseconds
import kotlin.time.Duration.Companion.milliseconds

import kotlinx.coroutines.*

suspend fun main() = withContext(Dispatchers.Default) { // this: CoroutineScope
    println("Running withContext block on ${Thread.currentThread().name}")

    val one = this.async {
        println("First calculation starting on ${Thread.currentThread().name}")
        val sum = (1L..500_000L).sum()
        delay(200L)
        println("First calculation done on ${Thread.currentThread().name}")
        sum
    }

    val two = this.async {
        println("Second calculation starting on ${Thread.currentThread().name}")
        val sum = (500_001L..1_000_000L).sum()
        println("Second calculation done on ${Thread.currentThread().name}")
        sum
    }

    // Waits for both calculations and prints the result
    println("Combined total: ${one.await() + two.await()}")
}
```

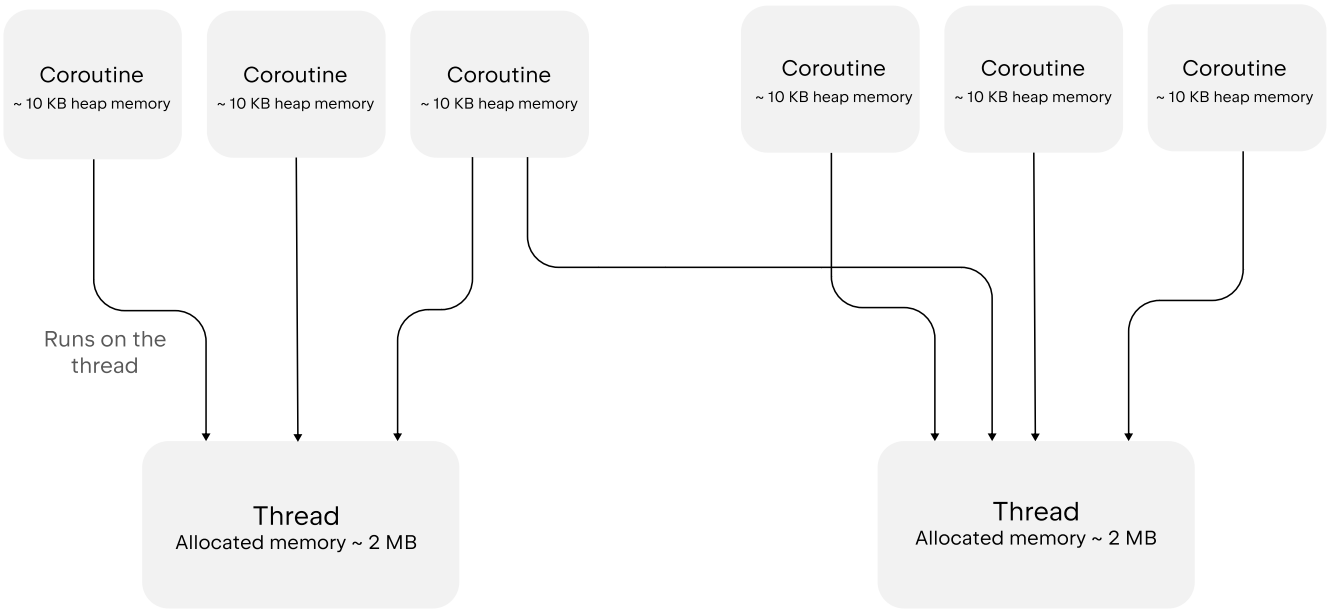
To learn more about coroutine dispatchers and their uses, including other dispatchers like `Dispatchers.IO` and `Dispatchers.Main`, see Coroutine context and dispatchers.

Comparing coroutines and JVM threads

While coroutines are suspendable computations that run code concurrently like threads on the JVM, they work differently under the hood.

A *thread* is managed by the operating system. Threads can run tasks in parallel on multiple CPU cores and represent a standard approach to concurrency on the JVM. When you create a thread, the operating system allocates memory for its stack and uses the kernel to switch between threads. This makes threads powerful but also resource-intensive. Each thread usually needs a few megabytes of memory, and typically the JVM can only handle a few thousand threads at once.

On the other hand, a coroutine isn't bound to a specific thread. It can suspend on one thread and resume on another, so many coroutines can share the same thread pool. When a coroutine suspends, the thread isn't blocked and remains free to run other tasks. This makes coroutines much lighter than threads and allows running millions of them in one process without exhausting system resources.



Comparing coroutines and threads

Let's look at an example where 50,000 coroutines each wait five seconds and then print a period (`.`):

```
import kotlin.time.Duration.Companion.seconds
import kotlinx.coroutines.*

suspend fun main() {
    withContext(Dispatchers.Default) {
        // Launches 50,000 coroutines that each wait five seconds, then print a period
        printPeriods()
    }
}

suspend fun printPeriods() = coroutineScope { // this: CoroutineScope
    // Launches 50,000 coroutines that each wait five seconds, then print a period
    repeat(50_000) {
        this.launch {
            delay(5.seconds)
            print(".")
        }
    }
}
```

Now let's look at the same example using JVM threads:

```
import kotlin.concurrent.thread

fun main() {
    repeat(50_000) {
        thread {
            Thread.sleep(5000L)
            print(".")
        }
    }
}
```

Running this version uses much more memory because each thread needs its own memory stack. For 50,000 threads, that can be up to 100 GB, compared to roughly 500 MB for the same number of coroutines.

Depending on your operating system, JDK version, and settings, the JVM thread version may throw an out-of-memory error or slow down thread creation to avoid running too many threads at once.

What's next

- Discover more about combining suspending functions in Composing suspending functions.
- Learn how to cancel coroutines and handle timeouts in Cancellation and timeouts.
- Dive deeper into coroutine execution and thread management in Coroutine context and dispatchers.
- Learn how to return multiple asynchronously computed values in Flows.

Cancellation and timeouts

Cancellation lets you request to stop a coroutine before it completes. It stops work that's no longer needed, such as when a user closes a window or navigates away in a user interface while a coroutine is still running.

You can use cancellation to release resources early and to stop a coroutine from accessing objects past their disposal. You can also use it to stop long-running coroutines that perform repeated work, such as:

- Sending heartbeats.

- Running scheduled tasks.
- Updating a state to reflect the newest reading, such as in a clock UI.

Cancellation works through the `Job` handle, which represents the lifecycle of a coroutine and its parent-child relationships. `Job` allows you to check whether the coroutine is active and allows you to cancel it, along with its children, as defined by structured concurrency.

Cancel coroutines

A coroutine is canceled when the `cancel()` function is invoked on its `Job` handle. Coroutine builder functions such as `.launch()` return a `Job`. The `.async()` function returns a `Deferred`, which implements `Job` and supports the same cancellation behavior.

You can call the `cancel()` function manually, or it can be invoked automatically through cancellation propagation when a parent coroutine is canceled.

When a coroutine is canceled, it throws a `CancellationException` the next time it checks for cancellation. Suspending functions in the `kotlinx.coroutines` library, such as the `delay()` function, check for cancellation when they suspend.

You can use the `awaitCancellation()` function to suspend a coroutine until it's canceled. It's equivalent to calling `delay(Duration.INFINITE)`.

> For more information about how and when coroutines check for cancellation, see Suspension points and cancellation.

Here's an example of how to manually cancel a coroutine:

```
import kotlinx.coroutines.*

suspend fun main() {
withContext(Dispatchers.Default) {
    // Used as a signal that the coroutine has started running
    val childStarted = CompletableDeferred<Unit>()

    val childJob: Job = launch {
        println("The coroutine has started")

        // Completes the CompletableDeferred,
        // signaling that the coroutine has started running
        childStarted.complete(Unit)
        try {
            // Suspends indefinitely
            // This call will never return unless the coroutine is canceled
            awaitCancellation()
        } catch (e: CancellationException) {
            println("The coroutine was canceled: $e")

            // Always rethrow cancellation exceptions!
            throw e
        }
        println("This line will never be executed")
    }

    // Waits for the coroutine to start before canceling it
    childStarted.await()

    // Cancels the coroutine,
    // so awaitCancellation() throws a CancellationException
    childJob.cancel()
}
// Coroutine builders such as withContext() or coroutineScope()
// wait for all child coroutines to complete,
// even when the children are canceled
println("All coroutines have completed")
}
```

In this example, `CompletableDeferred` is used as a signal that the coroutine has started running. The coroutine calls `complete()` when it starts executing, and `await()` only returns once that `CompletableDeferred` is completed. You don't need this check to cancel a coroutine. It's included here to make the example reproducible by ensuring that the coroutine starts and prints its messages before it's canceled.

Because `Deferred` implements `Job`, cancellation works the same way for coroutines created by the `async()` coroutine builder function:

```
val deferred = async { /* ... */ }
deferred.cancel()
```

Catching `CancellationException` can break the cancellation propagation. If you must catch it, rethrow it to let the cancellation propagate correctly through the coroutine hierarchy.

For more information, see Coroutine exceptions handling.

Cancellation propagation

Structured concurrency ensures that canceling a coroutine also cancels all of its children. This prevents child coroutines from continuing their work after their parent coroutine is canceled.

Here's an example:

```
import kotlinx.coroutines.*

suspend fun main() {
    withContext(Dispatchers.Default) {
// Used as a signal that the child coroutines have been launched
val childrenLaunched = CompletableDeferred<Unit>()

// Launches two child coroutines
val parentJob = launch {
    launch {
        println("Child coroutine 1 has started running")
        try {
            awaitCancellation()
        } finally {
            println("Child coroutine 1 has been canceled")
        }
    }
    launch {
        println("Child coroutine 2 has started running")
        try {
            awaitCancellation()
        } finally {
            println("Child coroutine 2 has been canceled")
        }
    }
    // Completes the CompletableDeferred,
    // signaling that the child coroutines have been launched
    childrenLaunched.complete(Unit)
}
// Waits for the parent coroutine to signal that it has launched
// all of its children
childrenLaunched.await()

// Cancels the parent coroutine, which cancels all its children
parentJob.cancel()
    }
}
```

In this example, each child coroutine uses a `finally` block, so the code inside it runs when the coroutine is canceled. Here, `CompletableDeferred` signals that the child coroutines are launched before they're canceled, but it doesn't guarantee that they start running. If they're canceled first, nothing is printed.

Make coroutines react to cancellation

In Kotlin, coroutine cancellation is *cooperative*. Coroutines react to cancellation only when they cooperate by suspending or checking for cancellation explicitly.

In this section, you can learn how adding suspension points, such as calls to the yield() function, lets coroutines react to cancellation.

Suspension points and cancellation

When a coroutine is canceled, it continues running until it reaches a point in the code where it may suspend, also known as a *suspension point*. If the coroutine suspends there, the suspending function checks whether it has been canceled. If it has, the coroutine stops and throws `CancellationException`.

A call to a `suspend` function is a suspension point, but it doesn't always suspend. For example, when awaiting a `Deferred` result, the coroutine only suspends if that `Deferred` isn't completed yet.

Here's an example using common suspending functions that suspend, enabling the coroutine to check and stop when it's canceled:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.sync.Mutex
import kotlinx.coroutines.channels.Channel
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.Duration

suspend fun main() {
withContext(Dispatchers.Default) {
    val childJobs = listOf(
        launch {
            // Suspends until canceled
            awaitCancellation()
        },
        launch {
            // Suspends until canceled
            delay(Duration.INFINITE)
        },
        launch {
            val channel = Channel<Int>()
            // Suspends while waiting for a value that's never sent
            channel.receive()
        },
        launch {
            val deferred = CompletableDeferred<Int>()
            // Suspends while waiting for a value that's never completed
            deferred.await()
        },
        launch {
            val mutex = Mutex(locked = true)
            // Suspends while waiting for a mutex that remains locked indefinitely
            mutex.lock()
        }
    )

    // Gives the child coroutines time to start and suspend
    delay(100.milliseconds)

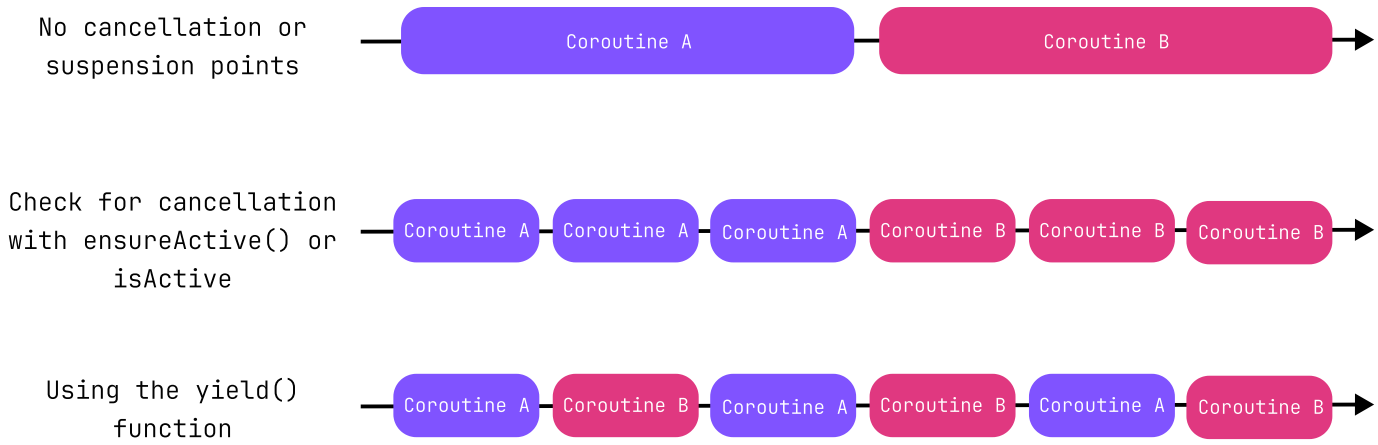
    // Cancels all child coroutines
    childJobs.forEach { it.cancel() }
}
println("All child jobs completed!")
}
```

All suspending functions in the `kotlinx.coroutines` library cooperate with cancellation because they use `suspendCancellableCoroutine()` internally, which checks for cancellation when the coroutine suspends. In contrast, custom suspending functions that use `suspendCoroutine()` don't react to cancellation.

The `yield()` suspending function

If a coroutine doesn't suspend, other coroutines can't run on the same thread until it completes. As a result, coroutines that don't suspend run sequentially on that thread. If a coroutine doesn't suspend for a long time, it also doesn't stop when it's canceled.

In CPU-intensive computations and other code that runs for a long time without suspending, call the `yield()` function periodically. This function releases the current thread and gives other coroutines a chance to run on it. It also ensures that the coroutine checks for cancellation regularly. If the coroutine is canceled, the `yield()` function throws a `CancellationException`.



Comparison of coroutine cancellation handling without checks, with ensureActive() or isActive, and with yield()

Here's an example:

```
import kotlinx.coroutines.*

fun main() {
// runBlocking uses the current thread for running all coroutines
runBlocking {
    val coroutineCount = 5
    repeat(coroutineCount) { coroutineIndex ->
        launch {
            val id = coroutineIndex + 1
            repeat(5) { iterationIndex ->
                val iteration = iterationIndex + 1
                // Suspends temporarily to give other coroutines a chance to run
                // Without this, the coroutines run sequentially
                yield()
                // Prints the coroutine index and iteration index
                println("$id * $iteration = ${id * iteration}")
            }
        }
    }
}
}
```

In this example, each coroutine uses `yield()` to let other coroutines run between iterations.

Check for cancellation explicitly

You can check for cancellation explicitly, which lets long-running code react to cancellation without suspending. A long-running coroutine that doesn't suspend can prevent other coroutines on the same thread from running until it completes. Unless this behavior is intentional for your use case, use the `yield()` function instead.

Depending on the API, the check either returns a Boolean value or throws an exception:

- The `isActive` property returns `false` when the coroutine is canceled.
- The `ensureActive()` function throws a `CancellationException` when the coroutine is canceled.

Interrupt blocking code when coroutines are canceled

On the JVM, some blocking functions, such as `Thread.sleep()` or `BlockingQueue.take()`, can block the current thread. These blocking functions can be interrupted, which stops them prematurely. However, when you call them from a coroutine, cancellation doesn't interrupt the thread.

To interrupt the thread when canceling a coroutine, wrap the blocking code in the `runInterruptible()` function:

```
import kotlinx.coroutines.*

suspend fun main() {
withContext(Dispatchers.Default) {
    val childStarted = CompletableDeferred<Unit>()
    val childJob = launch {
        try {
            // Cancellation triggers a thread interruption
            runInterruptible {
                childStarted.complete(Unit)
                try {
                    // Blocks the current thread for a very long time
                    Thread.sleep(Long.MAX_VALUE)
                } catch (e: InterruptedException) {
                    println("Thread interrupted (Java): $e")
                    throw e
                }
            }
        } catch (e: CancellationException) {
            println("Coroutine canceled (Kotlin): $e")
            throw e
        }
    }
    childStarted.await()

    // Cancels the coroutine and interrupts the thread executing Thread.sleep()
    childJob.cancel()
}
}
```

Handle values safely when canceling coroutines

When a suspended coroutine is canceled, it resumes with a `CancellationException` instead of returning any values, even if those values are already available. This behavior is called *prompt cancellation*. It prevents your code from continuing in a canceled coroutine's scope, such as updating a screen that's already closed.

Here's an example:

```
// Defines a coroutine scope that uses the UI thread
class ScreenWithButtons(private val scope: CoroutineScope) {
    fun loadAndUpdateButtons(filename: String) {
        scope.launch {
            // withContext() checks for cancellation before entering the block
            // and after the block returns
            val buttonNames = withContext(Dispatchers.IO) {
                // This is a blocking call that doesn't react to cancellation
                readLines(filename)
            }

            // It's safe to call updateUi()
            // because withContext() doesn't return if the coroutine is canceled,
            // and no code running on the UI thread can dispose of the buttons before this call
            updateUi(buttonNames)
        }
    }

    // Call this function only from the UI thread because it accesses the buttons
    // Throws an exception if called after the buttons are disposed
    private fun updateUi(buttonNames: List<String>) {
        // Placeholder code that updates the buttons with specified names
    }

    // Call this function only from the UI thread
    fun leaveScreen() {
        // Cancels the scope when leaving the screen
        // You can no longer update the UI
        scope.cancel()
    }
}

// UI controller code
setHandler(Event.ScreenClosed) {
    // Runs on the UI thread
    screenWithButtons.leaveScreen()
    buttons.dispose()
}
```

In this example, `withContext(Dispatchers.IO)` cooperates with cancellation and prevents `updateUi()` from running if the `leaveScreen()` function cancels the coroutine before `withContext(Dispatchers.IO)` returns the button names.

While prompt cancellation prevents using values after they're no longer valid, it can also stop your code while an important value is still in use, which might lead to losing that value. This can happen when a coroutine receives a value, such as an `AutoCloseable` resource, but is canceled before it can reach the part of the code that closes it. To prevent this, keep cleanup logic in a place that's guaranteed to run even when the coroutine receiving the value is canceled.

Here's an example:

```
import java.nio.file.*
import java.nio.charset.*
import kotlinx.coroutines.*
import java.io.*

// Uses a scope that runs its coroutines on the UI thread
class ScreenWithFileContents(private val scope: CoroutineScope) {
    fun displayFile(path: Path) {
        scope.launch {
            // Stores the reader in a variable, so the finally block can close it
            var reader: BufferedReader? = null

            try {
                withContext(Dispatchers.IO) {
                    reader = Files.newBufferedReader(
                        path, Charset.forName("US-ASCII")
                    )
                }
                // Uses the stored reader after withContext() completes
                updateUi(reader!!)
            } finally {
                // Ensures the reader is closed even when the coroutine is canceled
                reader?.close()
            }
        }
    }

    private suspend fun updateUi(reader: BufferedReader) {
        // Shows the file contents
        while (true) {
            val line = withContext(Dispatchers.IO) {
                reader.readLine()
            }
            if (line == null)
                break
            addOneLineToUi(line)
        }
    }

    private fun addOneLineToUi(line: String) {
        // Placeholder for code that adds one line to the UI
    }

    // Only callable from the UI thread
    fun leaveScreen() {
        // Cancels the scope and prevents its coroutines from updating the UI
        scope.cancel()
    }
}
```

In this example, storing the `BufferedReader` in a variable and closing it in the `finally` block ensures the resource is released even if the coroutine is canceled.

Run non-cancelable blocks

You can prevent cancellation from affecting certain parts of a coroutine. To do so, pass `NonCancellable` as an argument to the `withContext()` coroutine builder function.

Avoid using `NonCancellable` with other coroutine builders like `.launch()` or `.async()`. Doing so disrupts structured concurrency by breaking the parent-child relationship.

`NonCancellable` is useful when you need to ensure that certain operations, such as closing resources with a suspending `close()` function, complete even if the coroutine is canceled before they finish.

Here's an example:

```
import kotlinx.coroutines.*
import kotlin.time.Duration.Companion.milliseconds

val serviceStarted = CompletableDeferred<Unit>()

fun startService() {
    println("Starting the service...")
    serviceStarted.complete(Unit)
}

suspend fun shutdownServiceAndWait() {
    println("Shutting down...")
    delay(100.milliseconds)
    println("Successfully shut down!")
}

suspend fun main() {
    withContext(Dispatchers.Default) {
        val childJob = launch {
            startService()
            try {
                awaitCancellation()
            } finally {
                withContext(NonCancellable) {
                    // Without withContext(NonCancellable),
                    // this function doesn't complete because the coroutine is canceled
                    shutdownServiceAndWait()
                }
            }
        }
        serviceStarted.await()
        childJob.cancel()
    }
    println("Exiting the program")
}
```

Timeout

A timeout allows you to automatically cancel a coroutine after a specified duration. You can use it to stop operations that take too long.

For example, if a request to download a picture from a server times out, you can retry or fallback to the local cache.

To specify a timeout, use the `withTimeoutOrNull()` function with a `Duration`:

```
import kotlinx.coroutines.*
import kotlin.time.Duration.Companion.milliseconds

suspend fun slowOperation(): String {
    try {
        delay(300.milliseconds)
        return "A"
    } catch (e: CancellationException) {
        println("The slow operation has been canceled: $e")
        throw e
    }
}

suspend fun fastOperation(): String {
    try {
        delay(15.milliseconds)
        return "B"
    } catch (e: CancellationException) {
        println("The fast operation has been canceled: $e")
        throw e
    }
}

suspend fun main() {
    withContext(Dispatchers.Default) {
        val slow = withTimeoutOrNull(100.milliseconds) {
            slowOperation()
        }
        println("The slow operation finished with $slow")
        val fast = withTimeoutOrNull(100.milliseconds) {
            fastOperation()
        }
        println("The fast operation finished with $fast")
    }
}
```

If the timeout exceeds the specified `Duration`, `withTimeoutOrNull()` returns `null`.

Flows

A flow represents a sequential stream of values that can be produced asynchronously. Unlike a suspending function, which returns one value, you can use flows to work with multiple sequential values over time.

You can use flows to create *flow pipelines* that load data progressively, react to event streams, and model subscription-style APIs.

A flow pipeline is a sequence of operations that involves the following roles:

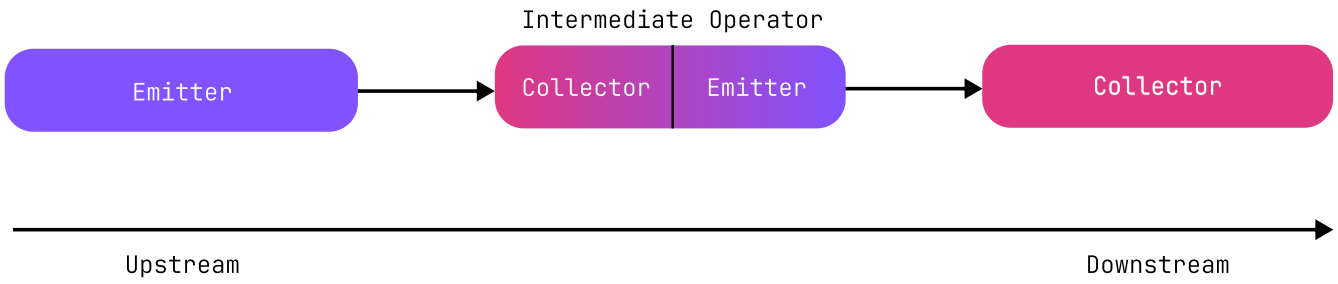
- **Emitter**: produces values.
- **Intermediate operators (optional)**: consume values from a flow, apply an operation to them, and return another flow.
- **Collector**: consumes values from a flow.

Here's a simple example that shows how these pipeline roles work together:

```kotlin
import kotlinx.coroutines.flow.*

suspend fun main() {
    // The emitter produces values
    flowOf(0x4B, 0x6F, 0x74, 0x6C, 0x69, 0x6E)
        // The intermediate operator consumes values,
        // applies an operation, and returns another flow
        .map { value -> value.toChar() }
        // The collector consumes the transformed values
        .collect { updatedValue ->
            println("Say '$updatedValue'!")
        }
}
```

In a flow, values move from an emitter toward a collector, from *upstream* to *downstream*. Intermediate operators collect an upstream flow, apply an operation to its values, and return a new downstream flow. That downstream flow can become the upstream flow for the next collector.



Parts of a flow: emitter, intermediate operator (optional), collector. Values move from upstream to downstream.

Kotlin provides the following flow types:

- **Cold flows** start producing values when collected. Each collector triggers a new, independent execution of the flow.
- **Hot flows** emit values independently of collectors and share the same stream of values with all collectors.

> You can use the Turbine library to test Kotlin flows. It simplifies collecting and asserting flow emissions in unit tests, including completion and failure cases.

Cold flows

Like sequences, cold flows are lazy.

The code block of a cold flow builder doesn't run until a collector collects it. Each new collector starts a new execution of the flow.

Create a cold flow

To create a cold flow, use the `flow()` builder function. Inside its block, use the `emit()` function to emit values to collectors:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() {
    // Creates a flow
    val pageFlow = flow {
        for (page in 1..3) {
            println("Loading page $page...")

            // Emits each page as it is loaded
            emit("Page $page")
        }
    }
    println("Creating a cold flow doesn't run it!")
}
```

In this example, the `flow()` builder function returns a `Flow<T>`, but it doesn't start executing its block. A cold flow is like a recipe: it defines how to produce values, but it only starts producing values when you collect it.

You can also create cold flows with the following functions:

- `flowOf()`: creates a flow from provided values.
- `.asFlow()`: converts an existing iterable, such as a range, into a flow.

Here's an example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() {
    // Creates a flow from provided values
    val predefinedPageFlow = flowOf("Page 1", "Page 2", "Page 3")
    // Creates a flow from a range
    val generatedPageFlow = (1..3).asFlow()
}
```

Collect a cold flow

To collect a cold flow, use the `collect()` function, which triggers the emissions from the upstream flow. If you pass a lambda to `collect()`, it receives each emitted value:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds

suspend fun main() {
    withContext(Dispatchers.Default) {
        val pageFlow = flow {
            for (page in 1..3) {
                println("Loading page $page...")
                emit("Page $page")
            }
        }
        // Collects the flow with a lambda that receives each emitted page
        pageFlow.collect { page ->
            println("Processing $page...")
            delay(100.milliseconds)
            println("Done processing $page.")
        }
    }
}
```

Each call to `collect()` runs the entire cold flow from the beginning. If multiple collectors collect the same cold flow, each collector triggers its own collection:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds

suspend fun main() {
    val pageFlow = flow {
        // Reads the name of the current coroutine
        val coroutineName = currentCoroutineContext()[CoroutineName]?.name

        println("Starting emissions in $coroutineName")
        for (page in 1..3) {
            println("Loading page $page in $coroutineName")
            emit("Page $page")
        }
        println("Done emitting in $coroutineName")
    }

    withContext(Dispatchers.Default) {
        // Launches a collector that processes each page slowly
        launch(CoroutineName("a slow coroutine")) {
            pageFlow.collect {
                println("Processing $it slowly")
                delay(100.milliseconds)
                println("Done processing $it slowly")
            }
        }

        // Launches a collector that processes each page quickly
        launch(CoroutineName("a fast coroutine")) {
            pageFlow.collect {
                println("Processing $it quickly")
                delay(10.milliseconds)
                println("Done processing $it quickly")
            }
        }
    }
}
```

In this example, `CoroutineName` adds a name to each coroutine. You can use `CoroutineName` for debugging. Here, it helps show which collector runs each collection.

Intermediate flow operators

Intermediate operators apply operations to an upstream flow and return a new downstream flow. They are cold, so the returned flow doesn't start processing values until it's collected, even when the upstream flow is hot.

The kotlinx.coroutines library offers a wide range of intermediate flow operators for transforming and processing flows. You can also define custom operators yourself when you need behavior that the built-in operators don't provide.

Here's an example of a simplified custom `.map()` operator that applies a transformation to each emitted value:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

// A simplified custom implementation of the default .map() operator
fun <T, R> Flow<T>.myMap(transform: suspend (value: T) -> R): Flow<R> = flow {
    // Collects values from the upstream flow
    this@myMap.collect { value ->
        // Transforms each collected value and emits the result
        emit(transform(value))
    }
}

suspend fun main() {
    // Creates a flow, applies the custom map operator, and collects the transformed values
    flowOf(1, 2, 3).myMap { 2 * it }.collect {
        println("Collecting $it")
    }
}
```

Call suspending functions inside a flow builder

Unlike in sequences, you can call suspending functions inside a `flow()` builder function:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

suspend fun loadPage(): Int {
    delay(100)
    return 3
}

suspend fun main() {
    flow {
        emit(loadPage())
    }.collect {
        println(it)
        // 3
    }
}
```

However, a `flow()` builder function must emit values from the same coroutine context where it runs. You can't start another coroutine that calls `emit()` in its block, and you can't change the coroutine context with `withContext()`:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

suspend fun main() {
    // This fails with an exception!
    flow {
        // Changes the coroutine context with withContext()
        withContext(Dispatchers.IO) {
            emit('a')
        }
    }.collect {
        println("This never prints")
    }
}
```

This restriction applies to the `flow()` builder function.

If you need the upstream flow to run in a different coroutine context, you can change it with the `.flowOn()` operator.

Alternatively, you can use `channelFlow()` to emit values from multiple coroutines.

Change the coroutine context of a cold flow with `.flowOn()`

By default, a cold flow runs in the same coroutine context as the collector.

If you want the flow to run in a different coroutine context, use the `.flowOn()` operator. This operator is *context-preserving*. It changes only the coroutine context of the upstream flow while keeping the downstream flow in the caller's context.

Here's an example of a cold flow that emits values in one coroutine context and collects them in another:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

suspend fun main() {
    withContext(Dispatchers.Default + CoroutineName("downstream")) {
        flow {
            val coroutineName = currentCoroutineContext()[CoroutineName]?.name

            // Emits in the coroutine context applied with .flowOn()
            println("Emitting '1' in $coroutineName")
            // Emitting '1' in upstream
            emit(1)

        // Changes the coroutine context of the upstream flow
        }.flowOn(Dispatchers.IO + CoroutineName("upstream"))
            .collect {
            val coroutineName = currentCoroutineContext()[CoroutineName]?.name

            // Collects in the caller's coroutine context
            println("Collecting '$it' in $coroutineName")
            // Collecting '1' in downstream
        }
    }
}
```

Handle exceptions in flows

Both emitters and collectors can throw exceptions.

If you don't handle an exception during flow collection, it propagates upstream from the collector and is thrown to the caller of the `collect()` function.

You can handle such exceptions by wrapping the `collect()` function in a `try-catch` block, for example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

class MyFlowException(message: String) : Exception(message)

suspend fun main() {
    val myFlow = flow {
        try {
            // The emit() function calls the lambda passed to collect()
            emit('a')
        } catch (e: MyFlowException) {
            println("Collector threw $e")

            // Rethrows the downstream exception
            throw e
        }
    }
    // Wraps flow collection in try-catch
    try {
        myFlow.collect {
            // Throws an exception from the collect() lambda
            throw MyFlowException("Can't process '$it'!")
        }
    } catch (e: MyFlowException) {
        println("Flow collection failed with $e")
        // Rethrows the exception to the caller
        throw e
    }
}
```

In this example, the collector throws an exception when it receives a value from the `emit()` function. The `flow()` builder function catches this downstream exception.

When you catch an exception thrown by the collector inside a flow builder function, rethrow it. This preserves exception transparency and lets the caller of `collect()` handle the exception.

Use the `.catch()` operator to handle upstream exceptions

To handle exceptions before they reach the collector, use the `.catch()` operator.

You can use the `.catch()` operator to handle exceptions from the upstream flow, for example by using the `emit()` function to emit a fallback value downstream:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

suspend fun main() {
    flow {
        emit("a")
        emit("b")

        // Throws an exception from the upstream flow
        throw UnsupportedOperationException(
            "I am tired of listing letters"
        )
    }.catch { upstreamException ->
        println("Upstream completed with $upstreamException!")

        // Emits a fallback value downstream
        emit("Upstream terminated with an exception!")
    }.collect {
        println("Got '$it'")
    }
}
```

In this example, the upstream flow emits values before throwing an exception. The `.catch()` operator handles the exception and emits `"Upstream terminated with an exception!"` as a fallback value.

When a flow is expected to throw some exceptions during normal operation, handle the recoverable exceptions in `.catch()` and rethrow any unexpected exceptions.

Here's an example where a flow loads data and reports its progress:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.random.*
import java.io.IOException
import kotlin.time.Duration.Companion.milliseconds

sealed interface LoadingState {
    sealed interface Terminal: LoadingState
    object Started: LoadingState
    data class Percentage(val percents: Int): LoadingState
    object Failed: Terminal
    object Done: Terminal
}

fun loadBlob(url: String) = flow {
    emit(LoadingState.Started)

    val failureChancePerStep = 1 - java.lang.Math.pow(0.99, 10.0)

    repeat(10) { step ->
        if (Random.nextDouble() < failureChancePerStep)
            throw IOException("Failed to load!")
        emit(LoadingState.Percentage((step + 1) * 10))
        delay(10.milliseconds)
    }
    emit(LoadingState.Done)
}.catch { e ->
    println("Loading data failed with $e")
    if (e is IOException) {
        // Handles an expected exception
        emit(LoadingState.Failed)
    } else {
        // Rethrows unexpected exceptions, so the collect() fails with them
        throw e
    }
}

suspend fun main() {
    loadBlob("https://example.com/").collect {
        println("Got '$it'")
    }
}
```

In this example, if loading fails with an expected exception, the `.catch()` operator uses the `emit()` function to emit a fallback state. For unexpected exceptions, rethrow them in the `.catch()` operator. This lets the caller of the `collect()` function receive exceptions that the flow doesn't handle.

The `.catch()` operator doesn't handle exceptions thrown by the collector. If the lambda passed to `collect()` throws an exception, handle it with a `try-catch` block around the `collect()` function:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

suspend fun main() {
    val myFlow = flow {
        for (char in listOf('a', 'o', '5', 'c')) {
            try {
                emit(char)
            } catch (e: IllegalArgumentException) {
                println("Collector doesn't support character '$char': $e")

                // Rethrows the downstream exception
                throw e
            }
        }
    }.catch { e ->
        // Doesn't run because the exception happens downstream
        println("Upstream threw an exception: $e")
    }

    try {
        myFlow.collect {
            require(!it.isDigit()) { "Digits are not allowed!" }
        }
    } catch (e: IllegalArgumentException) {
        // Handles the exception from the collect() lambda
        println("Flow collection failed with $e")
    }
}
```

Since the `collect()` lambda runs after `.catch()`, you can't use `.catch()` to handle exceptions thrown from it. To handle exceptions from code that runs for each emitted value with `.catch()`, place that code in .onEach() before `.catch()`.

The `.onEach()` operator runs its lambda before each value is emitted downstream. If `.catch()` handles an exception from `.onEach()`, the flow completes and doesn't emit the next value:

Here's an example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.random.*
import java.io.IOException
import kotlin.time.Duration.Companion.milliseconds

suspend fun main() {
    flowOf('a', 'o', '5', 'c')
        // Runs before each value is emitted downstream
        .onEach {
            require(!it.isDigit()) { "Digits are not allowed!" }
            println("Got '$it'")
        }
        .catch { e ->
            println("Caught an exception: $e")
        }
        .collect()
}
```

In this example, the `.onEach()` operator runs upstream from `.catch()`, so the `.catch()` operator handles the exception when the `require()` check fails for `'5'`.

Restart the upstream flow after an exception

Some operations may temporarily fail, such as a network request that loses its connection. For these cases, you can use the `.retry()` operator to restart the upstream flow after an exception.

The `.retry()` operator receives an exception and restarts collection when its lambda returns `true`, up to the specified number of retries. For example, `.retry(3)` retries the upstream flow up to three times after the first failed attempt.

If the lambda returns `false`, `.retry()` stops retrying and rethrows the exception.

> For more control over the retry logic, use the `.retryWhen()` operator. Like `.retry()`, it receives the exception, but it also receives the current attempt number and can emit values before retrying.

Here's an example that retries loading up to three times after an `IOException`:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.random.*
import java.io.IOException
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.Duration.Companion.seconds

sealed interface LoadingState {
    sealed interface Terminal: LoadingState
    object Started: LoadingState
    data class Percentage(val percents: Int): LoadingState
    object Failed: Terminal
    object Done: Terminal
}

fun loadBlob(url: String) = flow {
    emit(LoadingState.Started)

    val failureChancePerStep = 1 - java.lang.Math.pow(0.99, 10.0)

    repeat(10) { step ->
        if (Random.nextDouble() < failureChancePerStep)
            throw IOException("Failed to load!")
        emit(LoadingState.Percentage((step + 1) * 10))
        delay(10.milliseconds)
    }
    emit(LoadingState.Done)
}.retry(3) { e ->
    if (e is IOException) {
        // This is an expected error
        // Waits for one second before retrying
        delay(1.seconds)
        true
    } else {
        // Stops retrying and rethrows unexpected exceptions
        false
    }
}

suspend fun main() {
    loadBlob("https://example.org/").collect {
        println("Got $it")
    }
}
```

Flow cancellation

Flow cancellation stops collection when the result is no longer needed, such as when a request times out.

Flow collection is tied to the coroutine that calls the `collect()` function. When that coroutine is canceled, the collection stops, and the upstream flow is canceled too.

To cancel flow collection, call the `cancel()` function on the `Job` of the collecting coroutine:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.random.*
import java.io.IOException
import kotlin.time.Duration.Companion.milliseconds

val myFlow = flow {
    var i = 0
    try {
        while (true) {
            println("Emitting $i")
            emit(i)
            println("Emitted $i")
            ++i
            delay(10.milliseconds)
        }
    } catch (e: Throwable) {
        println("Upstream finished with $e")
        throw e
    }
}

suspend fun main() {
    coroutineScope {
        val job = launch {
            try {
                myFlow.collect {
                    println("Processing $it")
                    delay(5.milliseconds)
                }
            } catch (e: Throwable) {
                println("Collection finished with $e")
                throw e
            }
        }
        delay(100.milliseconds)

        // Cancels the coroutine that collects the flow
        job.cancel()
    }
}
```

The collector can also cancel the upstream flow while the collecting coroutine stays active. To do this, throw a `CancellationException` from the collector.

The `.take()` operator uses this behavior to stop collecting after a fixed number of values. For example, `.take(3)` collects only the first three values from the upstream flow, then cancels it.

Here's an example that uses a simplified version of the `.take()` operator:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.random.*
import java.io.IOException
import kotlin.time.Duration.Companion.milliseconds

// Defines a simplified version of the default .take() operator
fun <T> Flow<T>.myTake(count: Int): Flow<T> = flow {
    require(count > 0)
    val cancellationException = CancellationException()
    var elementsRemaining = count
    try {
        this@myTake.collect {
            emit(it)
            --elementsRemaining
            if (elementsRemaining == 0) {
                // Cancels the upstream flow after the requested number of values
                throw cancellationException
            }
        }
    } catch (e: Throwable) {
        if (e === cancellationException) {
            // Handles the CancellationException used to cancel the upstream flow
            // Completes the flow after the set number of values in .myTake()
        } else {
            // Rethrows unexpected exceptions
            throw e
        }
    }
}

suspend fun main() {
    (0..1000).asFlow().myTake(3).collect {
        println("Got $it")
    }
}
```

In this example, the `.myTake()` function emits values from the upstream flow until all requested values are emitted. Then it throws a `CancellationException` to cancel the upstream flow.

Emit values concurrently with `channelFlow()`

The `flow()` builder function is simple and efficient for flows that emit values from one coroutine. If you want to emit values from multiple coroutines concurrently into the same flow, use the `channelFlow()` builder function. You can use it for concurrent work that reports results progressively, such as loading data from multiple sources.

The `channelFlow()` builder function creates a cold flow that uses a channel to send values from multiple coroutines. Inside the builder, use the `send()` function instead of the `emit()` function to produce values.

Here's an example that uses `channelFlow()` to collect two flows concurrently and re-emit their values with a simplified version of the `.merge()` operator:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.random.*
import java.io.IOException
import kotlin.time.Duration.Companion.milliseconds

// Defines a simplified version of the default .merge() operator
fun <T> Flow<T>.myMerge(other: Flow<T>): Flow<T> = channelFlow {
    // CoroutineScope and SendChannel are available as receivers here
    // Launches a coroutine that collects the receiver flow
    launch {
        // Collects the receiver flow
        this@myMerge.collect {
            send(it)
        }
    }
    launch {
        // Launches a coroutine that collects the other flow
        other.collect {
            // Calls SendChannel.send
            send(it)
        }
    }
}

suspend fun main() {
    val flow1 = (0..3).asFlow().onEach { delay(20.milliseconds) }
    val flow2 = (6..9).asFlow().onEach { delay(50.milliseconds) }
    flow1.myMerge(flow2).collect { println(it) }
}
```

The `channelFlow()` builder function uses a buffered channel, allowing producers to send values ahead of the collector until the buffer is full. By default, the buffer can hold up to 64 values. When the buffer is full, producers suspend until the buffer has free space.

You can change the buffer capacity with the `.buffer()` operator. For example, `.buffer(12)` lets producers send up to 12 values ahead of the collector, while `.buffer(0)` removes the buffer, so each value is sent only when the collector can receive it.

Here's an example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.random.*
import java.io.IOException
import kotlin.time.Duration.Companion.milliseconds

suspend fun main() {
    val oneHundredNumbers = channelFlow {
        repeat(100) {
            println("Sending $it")
            send(it)
        }
    }

    // Uses the default buffer capacity
    oneHundredNumbers.collect {
        println("Processing $it")
        delay(10.milliseconds)
    }

    // Removes the buffer so sending and processing interleave from the start
    oneHundredNumbers.buffer(0).collect {
        println("Processing $it")
        delay(10.milliseconds)
    }
}
```

In this example, `oneHundredNumbers` flow uses the default buffer capacity, while `oneHundredNumbers.buffer(0)` flow has no buffer.

With the default buffer capacity, the producer sends values quickly until the buffer becomes full. After that, `send()` suspends until the buffer has free space, so `Sending` and `Processing` messages start to interleave.

With `.buffer(0)`, each `send()` call waits until the collector can receive the value, so `Sending` and `Processing` interleave from the start.

Hot flows

Hot flows are shared streams that emit values independently of collectors. They keep emitting values even when no collector is active, and multiple collectors can collect the same emissions from an already active stream instead of starting a new execution.

A collector of a hot flow is called a *subscriber*.

You can use hot flows when multiple parts of an application need to react to the same stream of updates, such as incoming chat messages, user actions, or UI state changes.

Kotlin provides two hot flow types:

- `SharedFlow` broadcasts values to multiple subscribers. Use it when you need to broadcast events that happen over time, such as messages or notifications.
- `StateFlow` is a specialized `SharedFlow` that always holds the latest state value. Use it when you need to represent state that changes over time, such as UI state.

Create a `SharedFlow`

`SharedFlow` is a hot flow that broadcasts emitted values that happen over time to its subscribers.

You can create a `SharedFlow` with the `MutableSharedFlow()` function.

A `MutableSharedFlow` exposes functions for emitting values. If you expose it directly, code outside the class can emit values into the flow.

To prevent this, store the mutable flow in a private backing property and expose a read-only `SharedFlow` with the `.asSharedFlow()` function. To emit values to subscribers, use the `emit()` function on the `MutableSharedFlow`:

```
data class Message(
    val senderId: Int,
    val time: Instant,
    val text: String,
)

class Chatroom {
    // Stores the SharedFlow in a private backing property
    private val _messages = MutableSharedFlow<Message>()

    // Exposes a read-only SharedFlow to subscribers
    val messages: SharedFlow<Message>
        get() = _messages.asSharedFlow()

    suspend fun sendMessageToEveryone(message: Message) {
        // Emits the message to subscribers
        _messages.emit(message)
    }
}
```

Just like with cold flows, you can use the `collect()` function to collect values from a `SharedFlow`.

You can also configure a `SharedFlow` to immediately replay already emitted values to new subscribers. The replay cache works like a small history buffer, storing a fixed number of previous emissions.

To set how many previous emissions a new subscriber receives, use the `replay` parameter in `MutableSharedFlow()`:

```
// Sets the number of already emitted messages new subscribers receive on subscription
const val MESSAGES_TO_REMEMBER = 10

class Chatroom {
    private val _messages = MutableSharedFlow<Message>(

        // Replays the set amount of last emitted messages to new subscribers
        replay = MESSAGES_TO_REMEMBER
    )

    val messages: SharedFlow<Message>
        get() = _messages.asSharedFlow()

    suspend fun sendMessageToEveryone(message: Message) {
        // Emits the message to subscribers of the messages flow
        _messages.emit(message)
    }
}
```

Collecting a hot flow doesn't complete by itself, so you must cancel the collecting coroutines when you no longer need them.

A hot flow doesn't have a close or cancel operation. Canceling collection only stops the corresponding subscriber from collecting. To stop new emissions, cancel the coroutine or scope that produces values for the hot flow.

Let's look at an example that uses a `SharedFlow` to model a chat room, which sends each new message to active subscribers and replays recent messages to subscribers that join later:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.random.*
import java.io.IOException
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.*

data class Message(
    val senderId: Int,
    val time: Instant,
    val text: String,
)

// Sets the number of already emitted messages that new subscribers receive on subscription
const val MESSAGES_TO_REMEMBER = 10

class Chatroom {
    // Stores the SharedFlow in a private backing property
    private val _messages = MutableSharedFlow<Message>(

        // Replays the set amount of last emitted messages to new subscribers
        replay = MESSAGES_TO_REMEMBER
    )

    // Exposes a read-only SharedFlow to subscribers
    val messages: SharedFlow<Message>
        get() = _messages.asSharedFlow()


    // Emits the message to subscribers
    suspend fun sendMessageToEveryone(message: Message) {
        _messages.emit(message)
    }
}

suspend fun main() {
    val nUsers = 3
    val chatroom = Chatroom()
    withContext(Dispatchers.Default) {
        // Starts a message reader for each user
        val messageReaders = List(nUsers) { userId ->
            // Starts collection before messages are emitted
            launch(start = CoroutineStart.UNDISPATCHED) {
                chatroom.messages.collect { message ->
                    println("User $userId received $message")
                }
            }
        }
        // Sends a greeting from each user
        repeat(nUsers) { userId ->
            chatroom.sendMessageToEveryone(
                Message(
                    userId,
                    Clock.System.now(),
                    "Hello from $userId!"
                )
            )
        }
        // Delays to make sure people have enough time to chat
        delay(100.milliseconds)
        // Cancels readers because SharedFlow collection doesn't finish by itself
        messageReaders.forEach { it.cancel() }
    }
```

```
}
```

In this example, `CoroutineStart.UNDISPATCHED` starts each collecting coroutine immediately.

This ensures that each coroutine reaches `collect()`, subscribes to `messages`, and suspends before `sendMessageToEveryone()` emits messages. Without it, a collecting coroutine might start later and miss earlier emissions if the replay cache is too small.

Experimental

Use explicit backing fields to expose hot flows

You can use explicit backing fields to expose a read-only `SharedFlow` while keeping a mutable backing field inside the class.

Explicit backing fields define the implementation type in the `field` declaration. Inside the class, the compiler smart casts the property to the backing field type, so you can call the `emit()` function without a separate private backing property.

> Explicit backing fields don't create the read-only wrapper that `.asSharedFlow()` provides. Use this pattern only when downcasting the exposed flow isn't a concern.

Here's an example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Clock
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.ExperimentalTime
import kotlin.time.Instant

data class Message(
    val senderId: Int,
    val time: Instant,
    val text: String,
)

const val MESSAGES_TO_REMEMBER = 10

class Chatroom {
    // Exposes a read-only SharedFlow with a mutable backing field
    val messages: SharedFlow<Message>
        field = MutableSharedFlow<Message>(
            replay = MESSAGES_TO_REMEMBER
        )

    suspend fun sendMessageToEveryone(message: Message) {
        // Emits through the mutable backing field inside Chatroom
        messages.emit(message)
    }
}

suspend fun main() {
    val nUsers = 3
    val chatroom = Chatroom()

    withContext(Dispatchers.Default) {
        val messageReaders = List(nUsers) { userId ->
            launch(start = CoroutineStart.UNDISPATCHED) {
                chatroom.messages.collect { message ->
                    println("User $userId received $message")
                }
            }
        }

        repeat(nUsers) { userId ->
            chatroom.sendMessageToEveryone(
                Message(
                    senderId = userId,
                    time = Clock.System.now(),
                    text = "Hello from $userId!"
                )
            )
        }

        delay(100.milliseconds)
        messageReaders.forEach { it.cancel() }
    }
}
```

Create a `StateFlow`

`StateFlow` is a hot flow that stores a single state value and emits updates when that value is replaced with a new one. New subscribers receive the current value as soon as they start collecting, and then receive new values each time the state updates.

You can use `StateFlow` to represent state that changes over time, such as loading progress, UI state, or the state of an object.

To create a `StateFlow`, use the `MutableStateFlow()` function with an initial value:

```
// Creates a MutableStateFlow with LoadingState.Started as the initial value
val result = MutableStateFlow<LoadingState>(LoadingState.Started)
```

To set the current state, use the `value` property:

```
fun loadBlob(url: String): StateFlow<LoadingState> {
    val result = MutableStateFlow<LoadingState>(LoadingState.Started)

    DownloadManager.startLoading(
        url,
        onPercentageLoaded = { percentage →
            // Replaces the current state with the latest progress
            result.value = LoadingState.Percentage(percentage)
        },
        onCompletion = {
            // Replaces the current state with the completion state
            result.value = LoadingState.Done
        },
        onFailure = {
            // Replaces the current state with the failure state
            result.value = LoadingState.Failed
        }
    )
}
```

Setting `value` is thread-safe and replaces the current state, but updating `value` based on its previous value isn't atomic. When a new state depends on the previous state, use `.update()` instead.

Similar to `MutableSharedFlow`, `MutableStateFlow` exposes APIs for emitting updates. If you expose it directly, any code that receives it can update the state by downcasting it to `MutableStateFlow`.

To prevent this, expose the mutable flow as a read-only `StateFlow` with the `.asStateFlow()` function:

```
fun loadBlob(url: String): StateFlow<LoadingState> {
    val result = MutableStateFlow<LoadingState>(LoadingState.Started)

    DownloadManager.startLoading(
        url,
        onPercentageLoaded = { percentage →
            result.value = LoadingState.Percentage(percentage)
        },
        onCompletion = {
            result.value = LoadingState.Done
        },
        onFailure = {
            result.value = LoadingState.Failed
        }
    )

    // Exposes the loading state as a read-only StateFlow
    return result.asStateFlow()
}
```

Here's an example that uses `StateFlow` to report loading progress from a callback-based API:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.*
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.Duration.Companion.seconds
import kotlin.io.encoding.*
import java.io.IOException
import kotlin.random.Random

sealed interface LoadingState {
    sealed interface Terminal: LoadingState
    object Started: LoadingState
    data class Percentage(val percents: Int): LoadingState
    object Failed: Terminal
    object Done: Terminal
}

fun loadBlob(url: String): StateFlow<LoadingState> {
    // Creates a mutable StateFlow with the initial loading state
    val result = MutableStateFlow<LoadingState>(LoadingState.Started)
    DownloadManager.startLoading(
        url,
        onPercentageLoaded = { percentage ->
            // Replaces the current state with the latest progress
            result.value = LoadingState.Percentage(percentage)
        },
        onCompletion = {
            // Replaces the current state with the completion state
            result.value = LoadingState.Done
        },
        onFailure = {
            // Replaces the current state with the failure state
            result.value = LoadingState.Failed
        }
    )
    // Exposes the loading state as a read-only StateFlow
    return result.asStateFlow()
}

// Defines a callback-based API that downloads data asynchronously
object DownloadManager {
    // Starts loading the url asynchronously
    fun startLoading(
        url: String,
        onPercentageLoaded: (Int) -> Unit,
        onCompletion: () -> Unit,
        onFailure: (Throwable) -> Unit
    ) {
        // Uses GlobalScope for illustrative purposes only,
        // to keep this example self-contained
        GlobalScope.launch {
            val failureChancePerStep = 1 - java.lang.Math.pow(0.99, 10.0)

            repeat(10) { step ->
                if (Random.nextDouble() < failureChancePerStep) {
                    onFailure(IOException("Failed to load!"))
                    return@launch
                }
                onPercentageLoaded((step + 1) * 10)
                delay(10.milliseconds)
            }
            onCompletion()
        }
```

```
    }
}

suspend fun main() {
    loadBlob("https://example.com/").onEach { state ->
        when (state) {
            is LoadingState.Started -> {
                // Waits for progress updates
            }
            is LoadingState.Percentage ->
                println("Loaded ${state.percents}...")
            is LoadingState.Failed ->
                println("Loading failed.")
            is LoadingState.Done ->
                println("Finished loading!")
        }
    }.takeWhile { it !is LoadingState.Terminal }.collect()
}
```

This example uses `GlobalScope` only to keep the callback-based API short. In your own applications, pass a `CoroutineScope` to the function that starts the work, such as `startLoading()` in this example, and launch the coroutine in that scope so callers can cancel the work when they no longer need it.

Since `StateFlow` is a hot flow, collection doesn't complete on its own. In this example, the `.takeWhile()` operator stops collection when loading reaches a terminal state.

A `StateFlow` emits an update only when the new value differs from the current value.

Avoid storing mutable objects in a `StateFlow`. Mutating the object itself doesn't replace the current value, so collectors don't receive an update.

You can also update a `StateFlow` by calculating the new state from the current state. Use the `.update()` function for these updates. The `.update()` function updates the value atomically, which helps when multiple coroutines update the same `MutableStateFlow`.

If you only need to update a shared value and don't need to observe state changes over time, use the Kotlin Atomics APIs, such as `AtomicInt` or `AtomicReference`.

Here's an example where likes are stored in a `StateFlow`, and each new state is calculated from the previous state:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.*
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.Duration.Companion.seconds
import kotlin.io.encoding.*
import java.io.IOException
import kotlin.random.Random

class Post(val id: Long) {
    // Stores the current number of likes as a StateFlow
    private val _numberOfLikes = MutableStateFlow<Int>(
        // Sets the initial number of likes
        0
    )

    // Exposes a read-only StateFlow with the current number of likes
    val numberOfLikes: StateFlow<Int>
        get() = _numberOfLikes.asStateFlow()


    // Adds a like
    fun like() {
        // Increments the number of likes atomically for concurrent and multithreaded calls
        _numberOfLikes.update { it + 1 }
    }
}

suspend fun drawUpdatedNumberOfLikes(likes: Int) {
    // Displays the latest number of likes
    println("${Clock.System.now()}: the number of likes is $likes")
}

suspend fun main() {
    withContext(Dispatchers.Default) {
        val post = Post(15)
        val notifyingJob = launch {
            post.numberOfLikes.collect {
                drawUpdatedNumberOfLikes(it)
            }
        }
        // Simulates users who like the post
        coroutineScope {
            repeat(10) {
                launch {
                    delay(Random.nextInt(100).milliseconds)
                    post.like()
                }
            }
        }
        // Cancels collection after all simulated users finish
        notifyingJob.cancelAndJoin()
    }
}
```

In this example, the `.update()` function increments the number of likes atomically. This prevents lost updates when multiple coroutines call the `like()` function at the same time.

Store accumulated state in a `StateFlow`

Sometimes you might want subscribers to receive the result of all previous emissions, not only the latest emitted value.

For example, a chat room can keep its message history as a single state value. When a new user joins the chat room, they receive the current message history first. Then they continue receiving updates when new messages arrive.

You can model this behavior with a `StateFlow`.

To do so, store the full message history as the current value with `StateFlow<List<Message>>`, instead of broadcasting each chat message as a separate event with `SharedFlow<Message>`:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.*
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.Duration.Companion.seconds
import kotlin.io.encoding.*
import java.io.IOException
import kotlin.random.Random

data class Message(
    val senderId: Int,
    val time: Instant,
    val text: String,
)

class Chatroom {
    // Stores the full message history
    private val _messageHistory = MutableStateFlow<List<Message>>(emptyList())

    // Exposes a read-only StateFlow with the current message history
    val messageHistory: StateFlow<List<Message>>
        get() = _messageHistory.asStateFlow()

    // Sends a message to all subscribers of the messageHistory flow
    suspend fun sendMessageToEveryone(message: Message) {
        // Adds the new message to the current history atomically
        _messageHistory.update {
            it + message
        }
    }
}

suspend fun main() {
    val nUsers = 3
    val chatroom = Chatroom()
    withContext(Dispatchers.Default) {
        // Starts a message reader for each user
        val messageReaders = List(nUsers) { userId ->
            launch(start = CoroutineStart.UNDISPATCHED) {
                chatroom.messageHistory.collect { currentHistory ->
                    println("User $userId sees the history as $currentHistory")
                }
            }
        }
        // Sends a greeting from each user
        repeat(nUsers) { userId ->
            chatroom.sendMessageToEveryone(
                Message(
                    userId,
                    Clock.System.now(),
                    "Hello from $userId!"
                )
            )
        }
        // Delays to make sure users have enough time to receive updates
        delay(100.milliseconds)
        // Cancels readers because StateFlow collection doesn't finish by itself
        messageReaders.forEach { it.cancel() }
    }
}
```

In this example, `messageHistory` stores the full list of previous messages as the current state. When a new message is sent, the `.update()` function creates a new list from the previous history and adds the new message atomically.

> Updating immutable collections by creating a new collection can take more time as the collection grows. You can create persistent collections with the Experimental `kotlinx.collections.immutable` library to make immutable collection updates more efficient.

Since `messageHistory` is a `StateFlow`, subscribers receive the current message history when they start collecting. After that, they receive a new list each time a message is sent, which changes the chat history.

Convert cold flows to hot flows

A cold flow runs its upstream operations separately for each collector. When multiple subscribers need emissions from the same upstream collection, you can convert the cold flow to a hot flow that shares that collection with its subscribers.

The following simplified version of `.shareIn()` demonstrates this idea by collecting a cold flow once, emitting its values into a `MutableSharedFlow`, and exposing it as a read-only `SharedFlow`:

```
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.*

fun <T> Flow<T>.simpleShareIn(scope: CoroutineScope): SharedFlow<T> {
    val sharedFlow = MutableSharedFlow<T>()
    scope.launch {
        this@simpleShareIn.collect {
            sharedFlow.emit(it)
        }
    }
    return sharedFlow.asSharedFlow()
}

suspend fun main() {

}
```

In this example, `simpleShareIn()` starts a new coroutine in the provided scope. To stop collecting from the upstream flow, cancel the scope that runs the collecting coroutine.

If the upstream flow throws an exception, this collecting coroutine fails. Use operators such as `.catch()` or `.retry()` before sharing the flow to handle upstream exceptions before the collecting coroutine fails.

The built-in `.shareIn()` function provides this pattern without requiring you to create a `MutableSharedFlow` yourself. It also adds options for controlling when upstream collection starts and stops, and how many previous emissions new subscribers receive.

To use the built-in `.shareIn()` function, provide the following arguments:

- The coroutine scope where the upstream flow is collected.
- The `SharingStarted` strategy that controls when upstream collection starts and stops. For example, `SharingStarted.Eagerly` starts upstream collection immediately in the provided scope, before any subscriber starts collecting.
- The optional `replay` value that controls how many previous emissions new subscribers receive.

The `.shareIn()` function collects the upstream flow in the provided coroutine scope and broadcasts its emissions to subscribers.

Here's an example where `.shareIn()` converts a cold flow to a hot flow that shares serialized chat messages with multiple subscribers:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.*
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.Duration.Companion.seconds
import kotlin.io.encoding.*
import java.io.IOException
import kotlin.random.Random

data class Message(
    val senderId: Int,
    val time: Instant,
    val text: String,
)

class Chatroom {
    // Stores the message flow
    private val _messages = MutableSharedFlow<Message>()

    // Exposes a read-only SharedFlow with emitted messages
    // New subscribers don't receive already emitted messages
    val messages: SharedFlow<Message>
        get() = _messages.asSharedFlow()

    // Sends a message to all subscribers of the messages flow
    suspend fun sendMessageToEveryone(message: Message) {
        _messages.emit(message)
    }
}

suspend fun main() {
    val nUsers = 3
    val chatroom = Chatroom()
    withContext(Dispatchers.Default) {
        // Creates a child scope of the currently running coroutine
        val derivedFlowsScope = CoroutineScope(
            currentCoroutineContext() + Job(currentCoroutineContext()[Job])
        )
        // Shares serialized messages between subscribers
        val serializedMessages: SharedFlow<String> =
            chatroom
                .messages
                .map {
                    // Serializes each message once for the shared flow
                    "senderId: ${it.senderId}, time: ${it.time}, text: " +
                        Base64.Default.encode(it.text.encodeToByteArray())
                }
                .shareIn(
                    // Starts the sharing coroutine in this scope.
                    // The upstream flow, including .map(), runs in that coroutine
                    derivedFlowsScope,

                    // Starts collecting the upstream flow immediately,
                    // before the first subscriber appears
                    SharingStarted.Eagerly,

                    // Doesn't replay previous serialized messages to new subscribers
                    replay = 0,
                )

        // Starts a message reader for each user
        val messageReaders = List(nUsers) { userId ->
            launch(start = CoroutineStart.UNDISPATCHED) {
```

```
                serializedMessages.collect { serializedMessage ->
                    println("User $userId observes the message $serializedMessage")
                }
            }
        }
        // Sends a greeting from each user
        repeat(nUsers) { userId ->
            chatroom.sendMessageToEveryone(
                Message(
                    userId,
                    Clock.System.now(),
                    "Hello from $userId!"
                )
            )
        }
        // Delays to make sure users have enough time to receive updates
        delay(100.milliseconds)
        // Cancels readers because SharedFlow collection doesn't finish by itself
        messageReaders.forEach { it.cancel() }
        // Cancels the scope that runs the derived hot flow
        derivedFlowsScope.cancel()
    }
}
```

In this example, the `.map()` operator creates a cold flow that serializes each message. Without the `.shareIn()` function, each collector runs that serialization separately. The `.shareIn()` function shares one upstream collection, so each message is serialized once and then shared with all subscribers.

Because `SharingStarted.Eagerly` starts upstream collection immediately, the derived hot flow starts collecting `chatroom.messages` as soon as `.shareIn()` is called.

Similarly, to convert a cold flow to a `StateFlow`, use the `.stateIn()` function.

Unlike `.shareIn()`, `.stateIn()` requires an initial value because a `StateFlow` must always have a current value.

For example:

```
val lastUpdateFlow: StateFlow<Instant?> =
    chatroom
        .messageHistory
        .map { currentHistory -> currentHistory.lastOrNull()?.time }
        .stateIn(
            // Starts the sharing coroutine in this scope
            // The upstream flow, including .map(), runs in that coroutine
            derivedFlowsScope,

            // Starts collecting when the first subscriber appears
            // and stops when the last subscriber disappears
            SharingStarted.WhileSubscribed(),
            // Sets the initial state before the first upstream emission
            null,
        )
```

Cancel hot flows

Hot flows don't stop when a subscriber is canceled.

When you cancel the coroutine that collects a hot flow, you only cancel that subscriber. The hot flow can still emit values to other subscribers, and the coroutine that produces those values can keep running.

Hot flows themselves don't have a cancellation operation. To cancel a hot flow, cancel the coroutine or scope that produces values for it.

Hot flows created with the `.shareIn()` or `.stateIn()` extension functions continue to collect the upstream flow until the sharing coroutine is canceled. To stop collection from the upstream flow, cancel the scope that runs the sharing coroutine.

> You can also stop upstream collection automatically when there are no subscribers with `SharingStarted.WhileSubscribed()`.

Here's an example where canceling the scope passed to `.stateIn()` stops a derived hot flow from collecting new values:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.*
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.Duration.Companion.seconds
import kotlin.io.encoding.*
import java.io.IOException
import kotlin.random.Random

data class Message(
    val senderId: Int,
    val time: Instant,
    val text: String,
)

class Chatroom {
    // Stores the message history
    private val _messageHistory = MutableStateFlow<List<Message>>(emptyList())

    // Exposes a read-only StateFlow with the current message history
    val messageHistory: StateFlow<List<Message>>
        get() = _messageHistory.asStateFlow()

    // Sends a message to all subscribers of the messageHistory flow
    suspend fun sendMessageToEveryone(message: Message) {
        _messageHistory.update {
            it + message
        }
    }
}

suspend fun main() {
    val chatroom = Chatroom()
    withContext(Dispatchers.Default) {
        // Creates a child scope of the currently running coroutine
        val derivedFlowsScope = CoroutineScope(
            currentCoroutineContext() + Job(currentCoroutineContext()[Job])
        )
        val totalMessages = chatroom.messageHistory
            .map { currentHistory ->
                currentHistory.size
            }.onEach {
                println("There are currently $it messages")
            }.stateIn(
                // Starts the sharing coroutine in this scope
                derivedFlowsScope
            )
        // Updates messageHistory
        chatroom.sendMessageToEveryone(
            Message(0, Clock.System.now(), "We are shutting down soon!")
        )
        delay(100.milliseconds)
        // Cancels the scope that runs the derived hot flow
        derivedFlowsScope.cancel()
        // Updates messageHistory, but totalMessages no longer receives the update
        chatroom.sendMessageToEveryone(
            Message(0, Clock.System.now(), "We have shut down.")
        )
        println("Last collected history size: ${totalMessages.value}")
        println("Actual history size: ${chatroom.messageHistory.value.size}")
    }
}
```

In this example, when you call the `derivedFlowsScope.cancel()` function, `totalMessages` stops collecting updates from `messageHistory`.

The `sendMessageToEveryone()` function still updates `messageHistory` because the coroutine that calls it wasn't canceled. As a result, `totalMessages.value` keeps the last collected size, while `chatroom.messageHistory.value.size` shows the actual number of messages.

Handle exceptions in hot flows

In cold flows, upstream exceptions propagate to the caller of `collect()` unless you use an operator such as `.catch()` to handle them first.

Hot flows don't propagate exceptions from producers to subscribers. If code that emits to a `MutableSharedFlow` or updates a `MutableStateFlow` throws an exception, handle it in the coroutine that runs that code. If a subscriber throws an exception while collecting, handle it in the collecting coroutine.

Hot flows created with the `.shareIn()` or `.stateIn()` extension functions collect from the upstream flow in a sharing coroutine. If the upstream flow throws an exception, the exception cancels the sharing coroutine:

```
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.*

suspend fun main() {
    withContext(Dispatchers.Default) {
        launch {
            flow<Int> {
                error("An upstream failure")
            }.stateIn(
                this@launch
            )
        }
    }
}
```

You can restart upstream collection after a failure. To do so, place the `.retry()` operator before `.shareIn()` or `.stateIn()`:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds

suspend fun main() {
    coroutineScope {
        launch {
            var currentAttempt = 0

            val stateFlow = flow {
                delay(10.milliseconds)

                if (currentAttempt++ < 5) {
                    println("An error happened!")
                    error("An upstream failure")
                } else {
                    println("Success.")
                    emit(10)
                }
            }
                // Restarts the upstream flow after recoverable failures
                .retry(retries = 5)
                .stateIn(
                    // Starts the sharing coroutine in this scope
                    this@launch
                )

            stateFlow.collect {
                println("Observed $it")

                // Cancels collection and the sharing coroutine
                this@launch.cancel()
            }
        }
    }
}
```

In this example, the flow fails five times before it emits a value. Because `.retry()` runs before `.stateIn()`, it handles each upstream failure before the failure reaches the sharing coroutine.

After the upstream flow emits `10`, the collecting coroutine receives the value and cancels itself. Since the same coroutine is also the parent of the sharing coroutine, this stops the derived hot flow.

Flow operators

Flow operators let you transform and process values in a flow pipeline. Kotlin provides two main kinds of flow operators:

- **Intermediate operators** return a new downstream flow that consumes values from upstream flows and applies an operation to them.
- **Terminal operators** trigger the execution of the flow pipeline by collecting an upstream flow. They may also return a result.

While the `kotlinx.coroutines` library offers a wide range of flow operators, you can also define custom ones when you need behavior that the built-in operators don't provide.

The following sections include examples for custom implementations alongside the corresponding built-in operators.

Intermediate operators

Intermediate operators return a new downstream flow that consumes values from upstream flows. You can chain several intermediate operators to build a flow pipeline before collecting the final result.

Intermediate operators can be classified by purpose into the following categories:

- **Transforming operators** transform values before emitting them downstream.
- **Filtering and size-limiting operators** control which upstream values continue downstream.
- **Concurrent processing operators** let emissions run separately from collection.
- **Combining operators** collect values from multiple upstream flows and emit them into one downstream flow.
- **Lifecycle operators** run actions in response to specific events during flow collection, such as when collection starts or when the upstream flow completes.

Transforming operators

Transforming operators transform the values emitted by an upstream flow. You can use them to convert values to another type, skip values, or emit additional values downstream.

Transforming operators accept suspending lambdas, so their lambdas can call suspending functions while processing each emitted value. They still process values sequentially unless the flow pipeline uses an operator that introduces concurrency.

The `.transform()` operator is a general transforming operator that you can use as the basis for more specific transforming operators, such as `.map()` and `.filter()`.

Here's an example that uses the `.transform()` operator to emit each upstream value as many times as its value:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

// A simplified custom implementation of the default .transform() operator
inline fun <T, R> Flow<T>.myTransform(
    // Accepts a suspending lambda that can emit values downstream
    crossinline transform: suspend FlowCollector<R>.(value: T) -> Unit
): Flow<R> = flow {
    // Collects values from the upstream flow
    this@myTransform.collect { value ->
        // Applies the transformation and emits values to the downstream flow
        this@flow.transform(value)
    }
}

// Uses the default .transform() operator
suspend fun main() = withContext(Dispatchers.Default) {
    val flow = (0..4).asFlow().transform { value ->
        // Emits each value as many times as its value
        repeat(value) {
            emit(value)
        }
    }
    println(flow.toList())
    // [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
}
```

You can use the `.map()` operator to transform each upstream value into one downstream value.

Here's an example that uses `.map()` to multiply each value by four:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

// A simplified custom implementation of the default .map() operator
inline fun <T, R> Flow<T>.myMap(
    crossinline transform: suspend (value: T) -> R
): Flow<R> = transform { value ->
    emit(transform(value))
}

suspend fun main() = withContext(Dispatchers.Default) {
    // Multiplies each upstream value by four
    val flow = (0..4).asFlow().map { it * 4 }
    println(flow.toList())
    // [0, 4, 8, 12, 16]
}
```

To emit only upstream values that match a condition, use the `.filter()` operator.

Here's an example that emits values where dividing by `3` leaves a remainder of `1`:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

// A simplified custom implementation of the default .filter() operator
inline fun <T> Flow<T>.myFilter(
    crossinline predicate: suspend (value: T) -> Boolean
): Flow<T> = transform { value ->
    // Emits only values that match the condition
    if (predicate(value))
        emit(value)
}

suspend fun main() = withContext(Dispatchers.Default) {
    // Emits only values where dividing by 3 leaves a remainder of 1
    val flow = (0..10).asFlow().filter { it % 3 == 1 }
    println(flow.toList())
    // [1, 4, 7, 10]
}
```

Some operators can combine the behavior of other transforming operators, such as `.map()` and `.filter()`, by transforming values and emitting only the results that match a condition.

For example, use `.mapNotNull()` to transform each upstream value and emit only non-null results:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

// A simplified custom implementation of the default .mapNotNull() operator
inline fun <T, R: Any> Flow<T>.myMapNotNull(
    crossinline transform: suspend (value: T) -> R?
): Flow<R> = transform { value ->
    transform(value)?.let { transformed ->
        emit(transformed)
    }
}

suspend fun main() = withContext(Dispatchers.Default) {
    // Converts each string to Double and skips values that can't be converted
    val flow = flowOf("1.2", "10", "11", "error", "0.000")
        .mapNotNull { it.toDoubleOrNull() }

    println(flow.toList())
    // [1.2, 10.0, 11.0, 0.0]
}
```

Filtering and size-limiting operators

Filtering and size-limiting operators control which values continue downstream from a flow. You can use them to remove repeated consecutive values, skip values from the beginning of a flow, or cancel collection after a specified number of values.

To ignore repeated consecutive values, use the `.distinctUntilChanged()` operator. It emits a value only when it differs from the previously emitted value:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

// A simplified custom version of the default .distinctUntilChanged() operator
fun <T> Flow<T>.myDistinctUntilChanged(): Flow<T> = flow {
    var lastEmitted: Any? = Any() // A value that's equal only to itself
    this@myDistinctUntilChanged.collect { value ->
        if (lastEmitted != value) {
            this@flow.emit(value)
            lastEmitted = value
        }
    }
}

suspend fun main() = withContext(Dispatchers.Default) {
    // Removes repeated consecutive values from the upstream flow
    val flow = flowOf(1, 2, 3, 3, 3, 4, 5, 5, 1).distinctUntilChanged()
    println(flow.toList())
    // [1, 2, 3, 4, 5, 1]
}
```

You can use the `.drop()` operator to skip the first values emitted by an upstream flow. For example, `.drop(2)` skips the first two values and emits the remaining values downstream:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

// A simplified custom version of the default .drop() operator
fun <T> Flow<T>.myDrop(count: Int): Flow<T> = flow {
    require(count >= 0)
    var elementsAlreadyDropped = 0
    this@myDrop.collect { value ->
        if (elementsAlreadyDropped == count) {
            this@flow.emit(value)
        } else {
            ++elementsAlreadyDropped
        }
    }
}

suspend fun main() = withContext(Dispatchers.Default) {
    // Skips the first two values from the upstream flow
    val flow = flowOf(1, 2, 3, 4, 5).drop(2)
    println(flow.toList())
    // [3, 4, 5]
}
```

To cancel collection after a fixed number of values, use the `.take()` operator. Here's an example that uses the `.take()` operator to collect only the first three values:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.random.*
import java.io.IOException
import kotlin.time.Duration.Companion.milliseconds

// A simplified custom version of the default .take() operator
fun <T> Flow<T>.myTake(count: Int): Flow<T> = flow {
    require(count > 0)
    val cancellationException = CancellationException()
    var elementsRemaining = count
    try {
        this@myTake.collect {
            emit(it)
            --elementsRemaining
            if (elementsRemaining == 0) {
                // Cancels the upstream flow after the requested number of values
                throw cancellationException
            }
        }
    } catch (e: Throwable) {
        if (e === cancellationException) {
            // Handles the CancellationException used to cancel the upstream flow
            // Completes the flow after the set number of values in .myTake()
        } else {
            // Rethrows unexpected exceptions
            throw e
        }
    }
}

suspend fun main() = withContext(Dispatchers.Default) {
    // Collects only the first three values from the upstream flow
    val flow = (0..1000).asFlow().take(3)

    println(flow.toList())
    // [0, 1, 2]
}
```

Concurrent processing operators

By default, a flow pipeline processes values sequentially. The upstream flow emits a value, and collectors process it before the next value is emitted.

To run the upstream flow concurrently with downstream collection, use concurrent processing operators to introduce a buffer. A buffer stores values that the upstream flow has emitted, but the collector hasn't processed yet.

One operator that introduces this buffer is the `.buffer()` operator. It lets you configure the buffer capacity and what happens when the buffer is full, for example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds

suspend fun main() = withContext(Dispatchers.Default) {
    flow {
        repeat(10) {
            emit(it)
            println("Emitted $it!")
        }
    }
        // Lets the upstream flow emit up to four values ahead of the collector
        .buffer(4)
        .collect {
            println("Processed $it!")
            delay(20.milliseconds)
        }
}
```

When the collector is slower than the upstream flow, the pipeline needs a way to handle values that the collector hasn't processed yet.

By default, the collector applies *backpressure* to the upstream flow. With this strategy, the upstream flow suspends when the buffer is full and resumes when the collector frees up space.

To drop values instead of suspending the upstream flow, set the `onBufferOverflow` parameter:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.BufferOverflow
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds

suspend fun main() = withContext(Dispatchers.Default) {
    flow {
        repeat(10) {
            emit(it)
            println("Emitted $it!")
        }
    }
        // Stores up to four values before applying the overflow behavior
        // Drops the oldest buffered value when the buffer is full
        .buffer(4, onBufferOverflow = BufferOverflow.DROP_OLDEST)
        .collect { value ->
            println("Processed $value!")
            delay(20.milliseconds)
        }
}
```

You can also use the `.conflate()` operator, which is a shorthand for `buffer(1, onBufferOverflow = BufferOverflow.DROP_OLDEST)`. Use it when you only want to process the latest values, skipping the values emitted while the previous value is being collected:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds

suspend fun main() = withContext(Dispatchers.Default) {
    flow {
        repeat(10) {
            emit(it)
            println("Emitted $it!")
        }
    }.conflate().collect {
        println("Processed $it!")
        delay(20.milliseconds)
    }
}
```

The `.conflate()` operator only affects which buffered values the collector processes. It doesn't cancel processing that has already started. To do that, use `collectLatest()` instead.

In the previous examples, the `.buffer()` and `.conflate()` operators run the upstream flow concurrently in a separate coroutine without changing its coroutine context.

To run the upstream flow in a different coroutine context, use the `.flowOn()` operator. If this changes the dispatcher, `.flowOn()` collects the upstream flow in a separate coroutine and uses a buffer between upstream emission and downstream collection.

Here's a simplified example that uses `.flowOn()` to run the upstream flow in `Dispatchers.IO`:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds

suspend fun main() = withContext(Dispatchers.Default) {
    flow {
        repeat(10) {
            emit(it)
        }
        println("Finished emitting!")
    }.flowOn(Dispatchers.IO).collect {
        println("Received $it!")
        delay(10.milliseconds)
    }
}
```

In this example, the `.flowOn()` operator can introduce concurrent upstream processing, but the buffer behavior isn't configured explicitly.

To configure both the coroutine context of the upstream flow and the buffer behavior, combine `.flowOn()` with `.buffer()` or `.conflate()`. When you use these operators together, the operators perform *operator fusion* and share a single buffer.

Here's an example that uses `.flowOn(Dispatchers.IO)` to run the upstream flow in `Dispatchers.IO` and `.conflate()` to keep the newest buffered value:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.random.Random
import kotlin.time.Duration.Companion.milliseconds
import kotlin.math.round

fun awaitSensorSignal(): SensorSignal {
    Thread.sleep(10)
    val reading =
        round(Random.nextDouble(25.0, 100.0) * 100.0)/100.0
    println("Measured $reading as the temperature")
    return SensorSignal(temperatureCelsius = reading)
}

data class SensorSignal(
    val temperatureCelsius: Double
)

suspend fun sendLatestTemperature(temperatureCelsius: Double) {
    println("Starting to send $temperatureCelsius...")
    delay(50.milliseconds)
    println("Sent $temperatureCelsius.")
}

suspend fun main() = withContext(Dispatchers.Default) {
    val smartHomeTemperatureFlow = flow {
        while (true) {
            val signal = awaitSensorSignal()
            emit(signal.temperatureCelsius)
            println("Emitted $signal")
        }
    }
        // Runs the upstream flow in Dispatchers.IO
        .flowOn(Dispatchers.IO)
        // Keeps the newest buffered value and drops older ones
        .conflate()
        // Collects the first two values from the upstream flow
        .take(2)
        .collect { temperature ->
            println("Received $temperature!")
            sendLatestTemperature(temperature)
        }
}
```

Combining operators

Combining operators consume values from multiple upstream flows and return a single downstream flow. Use them when collectors need values from more than one flow.

To pair values from two upstream flows, use the `.zip()` operator. It combines the first value from each flow, then the second value from each flow, and so on. The resulting flow completes as soon as one of the upstream flows completes.

Here's an example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.random.Random
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.TimeSource

suspend fun main() = withContext(Dispatchers.Default) {
    // Emits a ticker value every 100 milliseconds
    val tickerFlow = flow {
        while (true) {
            emit(Unit)
            delay(100.milliseconds)
        }
    }

    val start = TimeSource.Monotonic.markNow()
    tickerFlow
        // Combines each ticker emission with the next number
        .zip(flowOf(1, 2, 3)) { _, value ->
            value
        }.collect {
            println("${start.elapsedNow()}: received $it")
        }
}
```

To combine the latest value from multiple flows, use the `.combine()` operator. It emits a new value when any upstream flow emits a value, using the latest value from each upstream flow:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.TimeSource

enum class Theme {
    Dark,
    Light,
}

data class UiState(
    val messages: List<String>,
    val theme: Theme,
)

val messagesFlow = MutableStateFlow(
    listOf(
        "Hello!",
        "Is anyone here?",
    )
)

val themeFlow = MutableStateFlow(
    Theme.Light
)

// Combines the latest values from both upstream flows
val uiStateFlow = combine(messagesFlow, themeFlow) { messages, theme ->
    UiState(messages, theme)
}

suspend fun main() {
    withContext(Dispatchers.Default) {
        // Uses UNDISPATCHED to subscribe before the first update happens
        val uiUpdateJob = launch(start = CoroutineStart.UNDISPATCHED) {
            uiStateFlow.collect {
                // Draws the UI
                println(it)
            }
        }
        messagesFlow.update { messages -> messages + "I'll be back!" }
        delay(100.milliseconds)

        themeFlow.value = Theme.Dark
        delay(100.milliseconds)

        uiUpdateJob.cancel()
    }
}
```

In this example, `combine()` creates `uiStateFlow` from the latest values of `messagesFlow` and `themeFlow`. Updating either upstream flow emits a new `UiState` with the latest messages and theme.

If you want to collect values from multiple flows concurrently and emit their values into one downstream flow, use the `.merge()` operator:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.TimeSource

interface UiEvent

class ClickEvent: UiEvent

class RightClickEvent: UiEvent

suspend fun main() {
    withContext(Dispatchers.Default) {

        val clickFlow = MutableSharedFlow<ClickEvent>()
        val rightClickFlow = MutableSharedFlow<RightClickEvent>()

        coroutineScope {
            // Uses UNDISPATCHED to subscribe before the first update happens
            val collectJob = launch(start = CoroutineStart.UNDISPATCHED) {

                // Collects both upstream flows concurrently and emits their values downstream
                merge(clickFlow, rightClickFlow).collect {
                    println("Observed an event: $it")
                }
            }
            clickFlow.emit(ClickEvent())
            delay(100.milliseconds)

            clickFlow.emit(ClickEvent())
            delay(100.milliseconds)

            rightClickFlow.emit(RightClickEvent())
            delay(100.milliseconds)

            collectJob.cancel()
        }
    }
}
```

Lifecycle operators

Lifecycle operators accept a suspending lambda that runs at a specific point during flow collection. You can use them to place logic before the flow is collected, before each value is emitted, after collection completes, or when the flow completes without emitting values.

The `.onStart()` operator runs its lambda before the upstream flow is collected. For code that needs to run before each emitted value, use `.onEach()`.

> Similarly to `.onStart()`, you can use `.onSubscription()` for hot flows to run code after a subscriber starts collecting the flow, but before it collects any emitted values.

Here's an example that uses these operators to print a message before collection starts and before each value is emitted downstream:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.TimeSource

// A simplified custom version of the default .onStart() operator
fun <T> Flow<T>.myOnStart(
    action: suspend FlowCollector<T>.() -> Unit
): Flow<T> = flow {
    this@flow.action()
    this@myOnStart.collect(this@flow)
}

suspend fun main() {
    withContext(Dispatchers.Default) {
        flowOf("Page 1", "Page 2", "Page 3").onStart {
            println("Processing pages!")
        }.onEach {
            println("Emitted $it")
        }.collect {
            println("Collected $it")
        }
    }
}
```

To run code after collection completes, use the `.onCompletion()` operator. Its lambda can emit values downstream when the upstream flow completes successfully, for example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.TimeSource

// A simplified custom version of the default .onCompletion() operator
fun <T> Flow<T>.myOnCompletion(
    action: suspend FlowCollector<T>.(cause: Throwable?) -> Unit
): Flow<T> = flow {
    var exception: Throwable? = null
    try {
        this@myOnCompletion.collect(this@flow)
    } catch (e: Throwable) {
        // Run `action`, but if `action` calls `emit`, throw `e` from it
        FlowCollector<T> { throw e }.action(e)
        throw e
    }
    this@flow.action(null)
}

suspend fun main() {
    withContext(Dispatchers.Default) {
        flowOf("Page 1", "Page 2", "Page 3").onCompletion {
            println("Almost done...")
            // Emits an additional value after the upstream flow completes
            emit("Last Page!")
        }.collect {
            println("Collected $it")
        }
    }
}
```

To run code when the upstream flow completes without emitting any values, use the `.onEmpty()` operator:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.TimeSource

// A simplified custom version of the default .onEmpty() operator
fun <T> Flow<T>.myOnEmpty(
    action: suspend FlowCollector<T>.() -> Unit
): Flow<T> = flow {
    var emittedSomething = false
    this@myOnEmpty.collect { value ->
        emittedSomething = true
        this@flow.emit(value)
    }
    if (!emittedSomething) {
        action()
    }
}

suspend fun main() {
    withContext(Dispatchers.Default) {
        flowOf("Page 1", "Page 2", "Page 3").onEmpty {
            // Doesn't print anything, because the upstream flow emits values
            println("No pages to load!")
        }.collect()
        flowOf<Int>().onEmpty {
            println("No pages to load!")
            // No pages to load!
        }.collect()
    }
}
```

Terminal operators

Terminal operators collect a flow. You can use them to consume emitted values, return a result based on the collected values, or collect a flow in a specific `CoroutineScope`.

To collect a flow, use the `collect()` operator. If you pass a lambda to `collect()`, it receives each emitted value:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

suspend fun main() {
    withContext(Dispatchers.Default) {
        flowOf(1, 2, 3).collect {
            println("Collected $it!")
        }
    }
}
```

You can also call `collect()` without a lambda:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

suspend fun main() {
    withContext(Dispatchers.Default) {
        flowOf(1, 2, 3).onEach {
            println("Collected $it!")
        }.collect()
    }
}
```

If you want to collect a flow but cancel unfinished work when a new value is emitted, use the `collectLatest()` operator:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.TimeSource

suspend fun main() {
    withContext(Dispatchers.Default) {
        flow {
            println("Emitting Page 1")
            emit("Page 1")
            delay(50.milliseconds)
            println("Emitting Page 2 in quick succession")
            emit("Page 2")
            delay(200.milliseconds)
            println("Emitting Page 3")
            emit("Page 3")
        }.flowOn(Dispatchers.IO).collectLatest {
            println("Starting to process $it!")
            try {
                delay(100.milliseconds)
            } catch (e: CancellationException) {
                println("Canceled processing $it.")
                throw e
            }
            println("Done processing!")
        }
    }
}
```

Some terminal operators collect a flow and return a result based on the collected values. For example, you can use the `.first()` operator to return the first emitted value and then cancel collection:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.TimeSource


suspend fun main() {
    withContext(Dispatchers.Default) {
        val firstValue = flowOf(1, 2, 3).first()

        println(firstValue)
        // 1
    }
}
```

You can collect emitted values into a collection with the `.toList()` or `.toSet()` operators:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

// A simplified custom implementation of the default .toList() operator
suspend fun <T> Flow<T>.myToList(): List<T> = buildList {
    this@myToList.collect { value ->
        // Adds each emitted value to the resulting list
        add(value)
    }
}

suspend fun main() {
    withContext(Dispatchers.Default) {
        val list = flowOf(1, 2, 3).toList()
        println(list)
        // [1, 2, 3]

        val set = flowOf(1, 2, 2, 3).toSet()
        println(set)
        // [1, 2, 3]
    }
}
```

To combine emitted values into a single result, use the `.reduce()` or `.fold()` operators. The `.fold()` operator uses the value you provide as its starting value, and the `.reduce()` operator uses the first emitted value instead.

Here's an example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

suspend fun main() {
    withContext(Dispatchers.Default) {
        // Uses the first emitted value as the starting value
        val reduced = flowOf(1, 2, 3).reduce { accumulator, value ->
            accumulator + value
        }

        // Starts with the provided starting value
        val folded = flowOf(1, 2, 3).fold(2) { accumulator, value ->
            accumulator + value
        }

        println(reduced)
        // 6

        println(folded)
        // 8
    }
}
```

Collect a flow in a specific `CoroutineScope`

When a screen or another long-lived object needs values from a flow, start the collector in that object's `CoroutineScope`. This ensures that canceling the object's `CoroutineScope` when the object is destroyed also cancels collection.

To collect a flow in a specific `CoroutineScope`, use the `.launchIn()` terminal operator. This operator returns the `Job` of the collecting coroutine.

Here's an example where a screen collects values from a `StateFlow` and stops the collecting coroutine when the screen closes:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.TimeSource

// A simplified custom version of the default .launchIn() operator
fun <T> Flow<T>.myLaunchIn(scope: CoroutineScope): Job = scope.launch {
    this@myLaunchIn.collect()
}

data class Coordinate(val x: Int, val y: Int)

class MyScreen(val scope: CoroutineScope) {
    private val _mousePosition =
        MutableStateFlow<Coordinate>(Coordinate(0, 0))
    val mousePosition get() = _mousePosition.asStateFlow()

    init {
        // Starts collecting the StateFlow in the screen's CoroutineScope
        mousePosition.onEach {
            updateStatusBar()
        }.launchIn(scope)
    }

    fun moveMouse(newCoordinate: Coordinate) {
        _mousePosition.value = newCoordinate
    }

    private fun updateStatusBar() {
        println("Mouse is at ${_mousePosition.value}")
    }
}

suspend fun main() {
    withContext(Dispatchers.Default) {
        val childScope = CoroutineScope(
            currentCoroutineContext() + Job(currentCoroutineContext()[Job])
        )
        val screen = MyScreen(childScope)
        delay(100.milliseconds)

        screen.moveMouse(Coordinate(10, 15))
        delay(100.milliseconds)

        screen.moveMouse(Coordinate(1, 3))
        delay(100.milliseconds)

        childScope.cancel()
    }
}
```

Coroutines and channels - tutorial

An upcoming update will revise this tutorial. In the meantime, for an up-to-date guide to getting started with coroutines, see Coroutines basics.

In this tutorial, you'll learn how to use coroutines in IntelliJ IDEA to perform network requests without blocking the underlying thread or callbacks.

No prior knowledge of coroutines is required, but you're expected to be familiar with basic Kotlin syntax.

You'll learn:

- Why and how to use suspending functions to perform network requests.
- How to send requests concurrently using coroutines.
- How to share information between different coroutines using channels.

For network requests, you'll need the Retrofit library, but the approach shown in this tutorial works similarly for any other libraries that support coroutines.

You can find solutions for all of the tasks on the `solutions` branch of the project's repository.

Before you start

1. Download and install the latest version of IntelliJ IDEA.
2. Clone the project template by choosing **Get from VCS** on the Welcome screen or selecting **File | New | Project from Version Control**.

   You can also clone it from the command line:

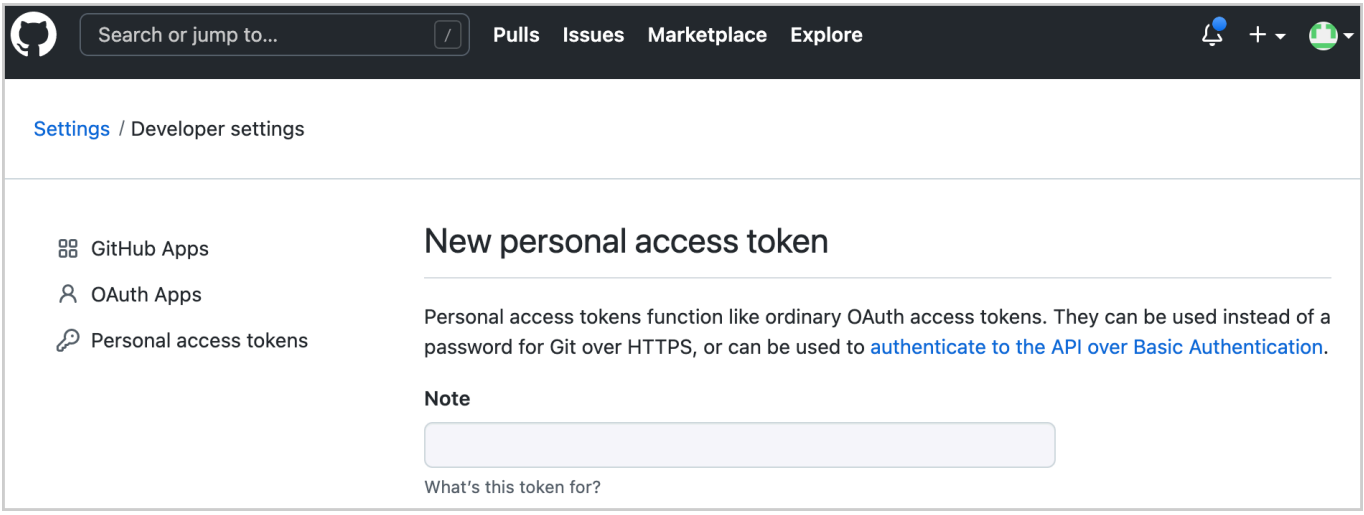
   ```
   git clone https://github.com/kotlin-hands-on/intro-coroutines
   ```

Generate a GitHub developer token

You'll be using the GitHub API in your project. To get access, provide your GitHub account name and either a password or a token. If you have two-factor authentication enabled, a token will be enough.

Generate a new GitHub token to use the GitHub API with your account:

1. Specify the name of your token, for example, `coroutines-tutorial` :



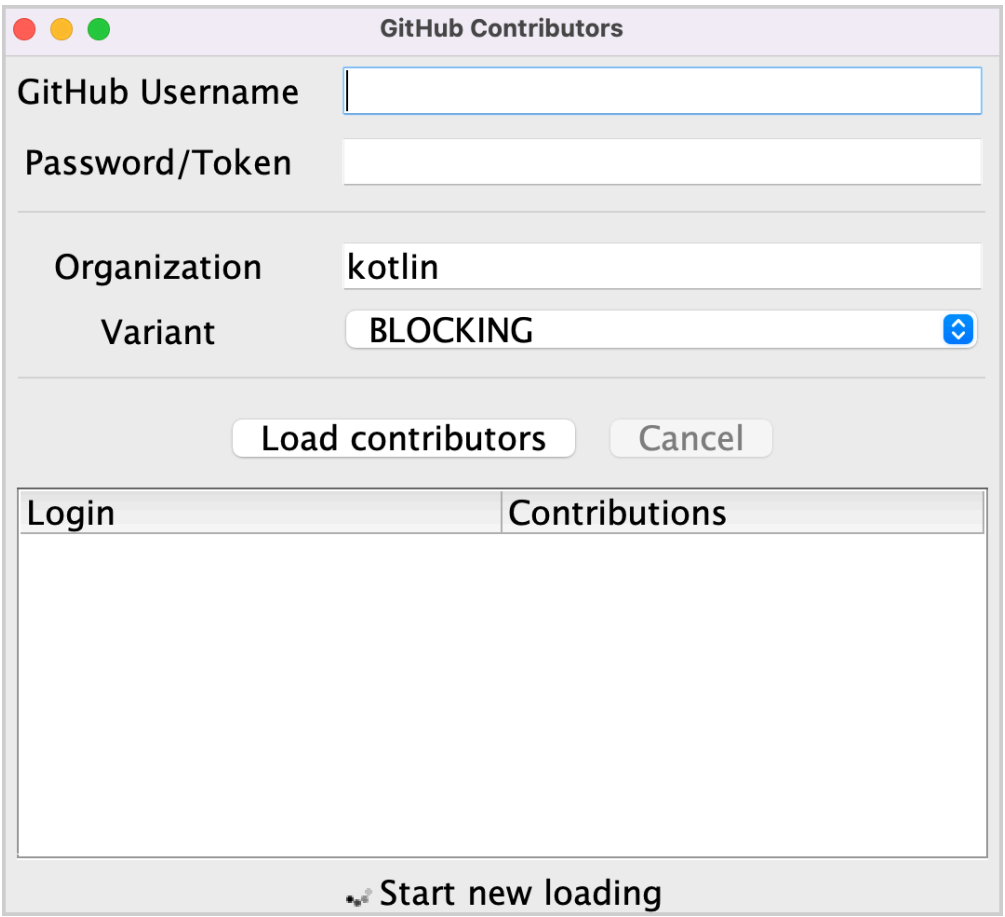
Generate a new GitHub token

2. Do not select any scopes. Click **Generate token** at the bottom of the page.
3. Copy the generated token.

Run the code

The program loads the contributors for all of the repositories under the given organization (named "kotlin" by default). Later you'll add logic to sort the users by the number of their contributions.

1. Open the `src/contributors/main.kt` file and run the `main()` function. You'll see the following window:



First window

If the font is too small, adjust it by changing the value of `setDefaultFontSize(18f)` in the `main()` function.

2. Provide your GitHub username and token (or password) in the corresponding fields.
3. Make sure that the *BLOCKING* option is selected in the *Variant* dropdown menu.
4. Click *Load contributors*. The UI should freeze for some time and then show the list of contributors.
5. Open the program output to ensure the data has been loaded. The list of contributors is logged after each successful request.

There are different ways of implementing this logic: by using blocking requests or callbacks. You'll compare these solutions with one that uses coroutines and see how channels can be used to share information between different coroutines.

Blocking requests

You will use the Retrofit library to perform HTTP requests to GitHub. It allows requesting the list of repositories under the given organization and the list of contributors for each repository:

```
interface GitHubService {
    @GET("orgs/{org}/repos?per_page=100")
    fun getOrgReposCall(
        @Path("org") org: String
    ): Call<List<Repo>>

    @GET("repos/{owner}/{repo}/contributors?per_page=100")
    fun getRepoContributorsCall(
        @Path("owner") owner: String,
        @Path("repo") repo: String
    ): Call<List<User>>
}
```

This API is used by the `loadContributorsBlocking()` function to fetch the list of contributors for the given organization.

1. Open `src/tasks/Request1Blocking.kt` to see its implementation:

```
fun loadContributorsBlocking(
    service: GitHubService,
    req: RequestData
): List<User> {
    val repos = service
        .getOrgReposCall(req.org)   // #1
        .execute()                  // #2
        .also { logRepos(req, it) } // #3
        .body() ?: emptyList()      // #4

    return repos.flatMap { repo ->
        service
            .getRepoContributorsCall(req.org, repo.name) // #1
            .execute()                                   // #2
            .also { logUsers(repo, it) }                 // #3
            .bodyList()                                  // #4
    }.aggregate()
}
```

-

At first, you get a list of the repositories under the given organization and store it in the `repos` list. Then for each repository, the list of contributors is requested, and all of the lists are merged into one final list of contributors.

- `getOrgReposCall()` and `getRepoContributorsCall()` both return an instance of the `*Call` class (`#1`). At this point, no request is sent.
- `*Call.execute()` is then invoked to perform the request (`#2`). `execute()` is a synchronous call that blocks the underlying thread.
- When you get the response, the result is logged by calling the specific `logRepos()` and `logUsers()` functions (`#3`). If the HTTP response contains an error, this error will be logged here.
- Finally, get the response's body, which contains the data you need. For this tutorial, you'll use an empty list as a result in case there is an error, and you'll log the corresponding error (`#4`).

2. To avoid repeating `.body() ?: emptyList()` , an extension function `bodyList()` is declared:

```
fun <T> Response<List<T>>.bodyList(): List<T> {
    return body() ?: emptyList()
}
```

3. Run the program again and take a look at the system output in IntelliJ IDEA. It should have something like this:

```
1770 [AWT-EventQueue-0] INFO  Contributors - kotlin: loaded 40 repos
2025 [AWT-EventQueue-0] INFO  Contributors - kotlin-examples: loaded 23 contributors
2229 [AWT-EventQueue-0] INFO  Contributors - kotlin-koans: loaded 45 contributors
...
```

- The first item on each line is the number of milliseconds that have passed since the program started, then the thread name in square brackets. You can see from which thread the loading request is called.
- The final item on each line is the actual message: how many repositories or contributors were loaded.

This log output demonstrates that all of the results were logged from the main thread. When you run the code with a *BLOCKING* option, the window freezes and doesn't react to input until the loading is finished. All of the requests are executed from the same thread as the one called `loadContributorsBlocking()` is from, which is the main UI thread (in Swing, it's an AWT event dispatching thread). This main thread becomes blocked, and that's why the UI is frozen:



The blocked main thread

After the list of contributors has loaded, the result is updated.

4. In `src/contributors/Contributors.kt` , find the `loadContributors()` function responsible for choosing how the contributors are loaded and look at how `loadContributorsBlocking()` is called:

```
when (getSelectedVariant()) {
    BLOCKING -> { // Blocking UI thread
        val users = loadContributorsBlocking(service, req)
        updateResults(users, startTime)
    }
}
```

- The `updateResults()` call goes right after the `loadContributorsBlocking()` call.
- `updateResults()` updates the UI, so it must always be called from the UI thread.
- Since `loadContributorsBlocking()` is also called from the UI thread, the UI thread becomes blocked and the UI is frozen.

Task 1

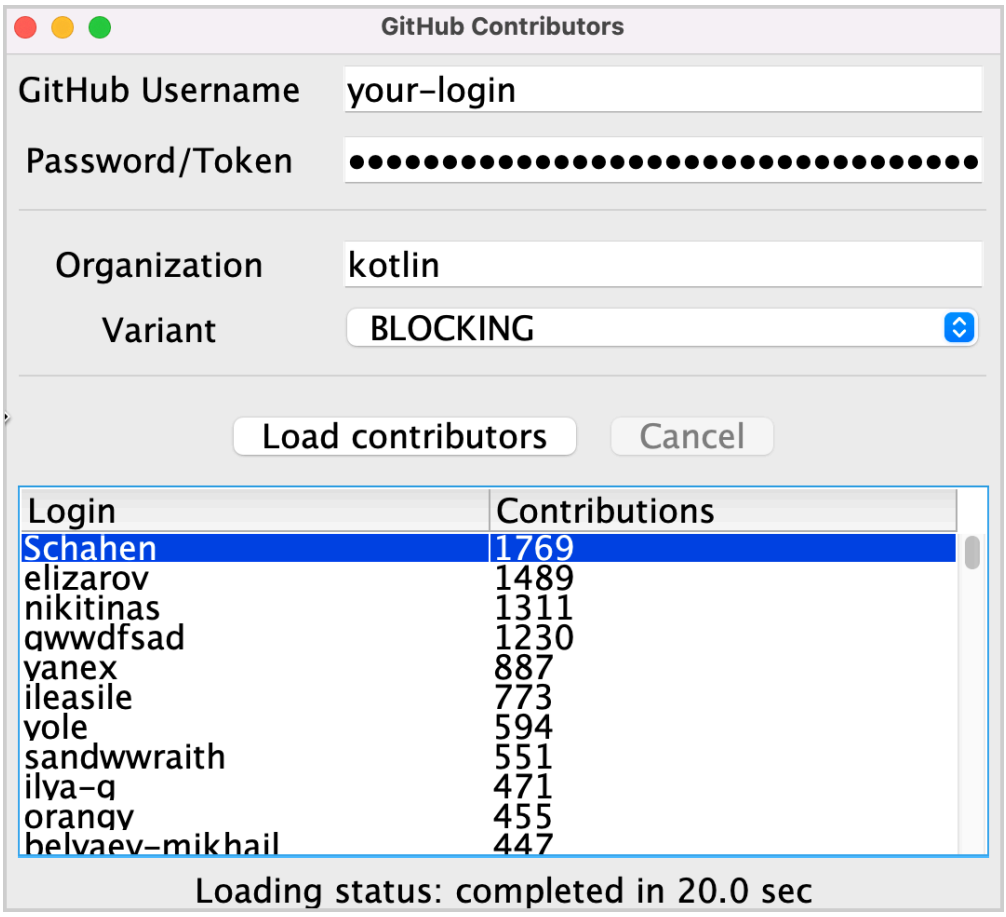
The first task helps you familiarize yourself with the task domain. Currently, each contributor's name is repeated several times, once for every project they have taken part in. Implement the `aggregate()` function combining the users so that each contributor is added only once. The `User.contributions` property should contain the total number of contributions of the given user to *all* the projects. The resulting list should be sorted in descending order according to the number of contributions.

Open `src/tasks/Aggregation.kt` and implement the `List<User>.aggregate()` function. Users should be sorted by the total number of their contributions.

The corresponding test file `test/tasks/AggregationKtTest.kt` shows an example of the expected result.

> You can jump between the source code and the test class automatically by using the IntelliJ IDEA shortcut `Ctrl+Shift+T` / `⇧ ⌘ T`.

After implementing this task, the resulting list for the "kotlin" organization should be similar to the following:



The list for the "kotlin" organization

Solution for task 1

1. To group users by login, use `groupBy()`, which returns a map from a login to all occurrences of the user with this login in different repositories.
2. For each map entry, count the total number of contributions for each user and create a new instance of the `User` class by the given name and total of contributions.
3. Sort the resulting list in descending order:

```
fun List<User>.aggregate(): List<User> =
    groupBy { it.login }
        .map { (login, group) -> User(login, group.sumOf { it.contributions }) }
        .sortedByDescending { it.contributions }
```

An alternative solution is to use the `groupingBy()` function instead of `groupBy()`.

Callbacks

The previous solution works, but it blocks the thread and therefore freezes the UI. A traditional approach that avoids this is to use *callbacks*.

Instead of calling the code that should be invoked right after the operation is completed, you can extract it into a separate callback, often a lambda, and pass that lambda to the caller in order for it to be called later.

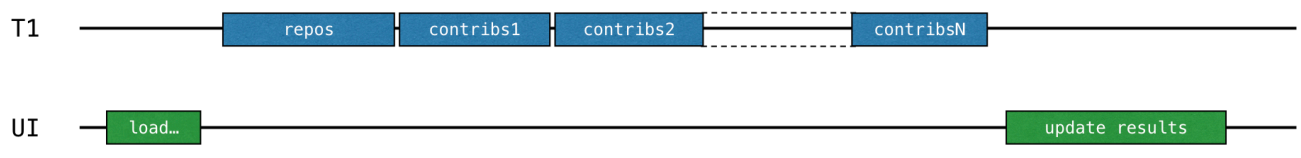
To make the UI responsive, you can either move the whole computation to a separate thread or switch to the Retrofit API which uses callbacks instead of blocking calls.

Use a background thread

1. Open `src/tasks/Request2Background.kt` and see its implementation. First, the whole computation is moved to a different thread. The `thread()` function starts a new thread:

```
thread {
    loadContributorsBlocking(service, req)
}
```

Now that all of the loading has been moved to a separate thread, the main thread is free and can be occupied by other tasks:



The freed main thread

2. The signature of the `loadContributorsBackground()` function changes. It takes an `updateResults()` callback as the last argument to call it after all the loading completes:

```
fun loadContributorsBackground(
    service: GitHubService, req: RequestData,
    updateResults: (List<User>) -> Unit
)
```

3. Now when the `loadContributorsBackground()` is called, the `updateResults()` call goes in the callback, not immediately afterward as it did before:

```
loadContributorsBackground(service, req) { users →
    SwingUtilities.invokeLater {
        updateResults(users, startTime)
    }
}
```

   By calling `SwingUtilities.invokeLater`, you ensure that the `updateResults()` call, which updates the results, happens on the main UI thread (AWT event dispatching thread).

However, if you try to load the contributors via the `BACKGROUND` option, you can see that the list is updated but nothing changes in UI.

Task 2

Fix the `loadContributorsBackground()` function in `src/tasks/Request2Background.kt` so that the resulting list is shown in the UI.

Solution for task 2

If you try to load the contributors, you can see in the log that the contributors are loaded but the result isn't displayed. To fix this, call `updateResults()` on the resulting list of users:

```
thread {
    updateResults(loadContributorsBlocking(service, req))
}
```

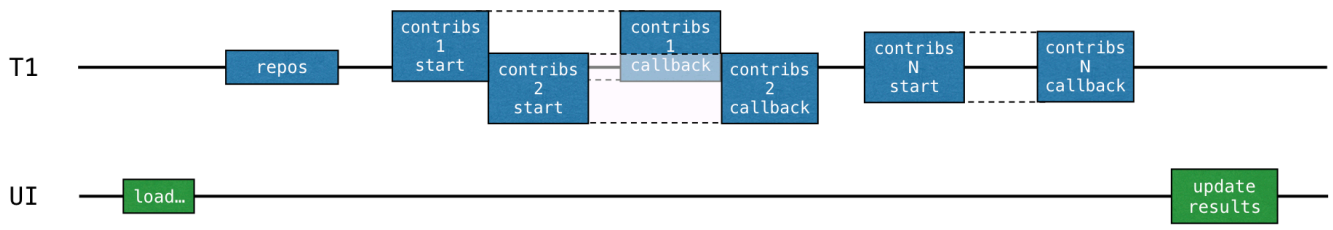
Make sure to call the logic passed in the callback explicitly. Otherwise, nothing will happen.

Use the Retrofit callback API

In the previous solution, the whole loading logic is moved to the background thread, but that still isn't the best use of resources. All of the loading requests go sequentially and the thread is blocked while waiting for the loading result, while it could have been occupied by other tasks. Specifically, the thread could start loading another request to receive the entire result earlier.

Handling the data for each repository should then be divided into two parts: loading and processing the resulting response. The second *processing* part should be extracted into a callback.

The loading for each repository can then be started before the result for the previous repository is received (and the corresponding callback is called):



Using callback API

The Retrofit callback API can help achieve this. The `Call.enqueue()` function starts an HTTP request and takes a callback as an argument. In this callback, you need to specify what needs to be done after each request.

Open `src/tasks/Request3Callbacks.kt` and see the implementation of `loadContributorsCallbacks()` that uses this API:

```
fun loadContributorsCallbacks(
    service: GitHubService, req: RequestData,
    updateResults: (List<User>) -> Unit
) {
    service.getOrgReposCall(req.org).onResponse { responseRepos ->  // #1
        logRepos(req, responseRepos)
        val repos = responseRepos.bodyList()

        val allUsers = mutableListOf<User>()
        for (repo in repos) {
            service.getRepoContributorsCall(req.org, repo.name)
                .onResponse { responseUsers ->  // #2
                    logUsers(repo, responseUsers)
                    val users = responseUsers.bodyList()
                    allUsers += users
                }
        }
        // TODO: Why doesn't this code work? How to fix that?
        updateResults(allUsers.aggregate())
    }
```

- For convenience, this code fragment uses the `onResponse()` extension function declared in the same file. It takes a lambda as an argument rather than an object expression.
- The logic for handling the responses is extracted into callbacks: the corresponding lambdas start at lines `#1` and `#2`.

However, the provided solution doesn't work. If you run the program and load contributors by choosing the *CALLBACKS* option, you'll see that nothing is shown. However, the test from `Request3CallbacksKtTest` immediately returns the result that it successfully passed.

Think about why the given code doesn't work as expected and try to fix it, or see the solutions below.

Task 3 (optional)

Rewrite the code in the `src/tasks/Request3Callbacks.kt` file so that the loaded list of contributors is shown.

The first attempted solution for task 3

In the current solution, many requests are started concurrently, which decreases the total loading time. However, the result isn't loaded. This is because the `updateResults()` callback is called right after all of the loading requests are started, before the `allUsers` list has been filled with the data.

You could try to fix this with a change like the following:

```
val allUsers = mutableListOf<User>()
for ((index, repo) in repos.withIndex()) {   // #1
    service.getRepoContributorsCall(req.org, repo.name)
        .onResponse { responseUsers ->
            logUsers(repo, responseUsers)
            val users = responseUsers.bodyList()
            allUsers += users
            if (index == repos.lastIndex) {    // #2
                updateResults(allUsers.aggregate())
            }
        }
}
```

- First, you iterate over the list of repos with an index (`#1`).
- Then, from each callback, you check whether it's the last iteration (`#2`).
- And if that's the case, the result is updated.

However, this code also fails to achieve our objective. Try to find the answer yourself, or see the solution below.

The second attempted solution for task 3

Since the loading requests are started concurrently, there's no guarantee that the result for the last one comes last. The results can come in any order.

Thus, if you compare the current index with the `lastIndex` as a condition for completion, you risk losing the results for some repos.

If the request that processes the last repo returns faster than some prior requests (which is likely to happen), all of the results for requests that take more time will be lost.

One way to fix this is to introduce an index and check whether all of the repositories have already been processed:

```
val allUsers = Collections.synchronizedList(mutableListOf<User>())
val numberOfProcessed = AtomicInteger()
for (repo in repos) {
    service.getRepoContributorsCall(req.org, repo.name)
        .onResponse { responseUsers ->
            logUsers(repo, responseUsers)
            val users = responseUsers.bodyList()
            allUsers += users
            if (numberOfProcessed.incrementAndGet() == repos.size) {
                updateResults(allUsers.aggregate())
            }
        }
}
```

This code uses a synchronized version of the list and `AtomicInteger()` because, in general, there's no guarantee that different callbacks that process `getRepoContributors()` requests will always be called from the same thread.

The third attempted solution for task 3

An even better solution is to use the `CountDownLatch` class. It stores a counter initialized with the number of repositories. This counter is decremented after processing each repository. It then waits until the latch is counted down to zero before updating the results:

```
val countDownLatch = CountDownLatch(repos.size)
for (repo in repos) {
    service.getRepoContributorsCall(req.org, repo.name)
        .onResponse { responseUsers ->
            // processing repository
            countDownLatch.countDown()
        }
}
countDownLatch.await()
updateResults(allUsers.aggregate())
```

The result is then updated from the main thread. This is more direct than delegating the logic to the child threads.

After reviewing these three attempts at a solution, you can see that writing correct code with callbacks is non-trivial and error-prone, especially when several underlying threads and synchronization occur.

> As an additional exercise, you can implement the same logic using a reactive approach with the RxJava library. All of the necessary dependencies and solutions for using RxJava can be found in a separate `rx` branch. It is also possible to complete this tutorial and implement or check the proposed Rx versions for a proper comparison.

Suspending functions

You can implement the same logic using suspending functions. Instead of returning `Call<List<Repo>>`, define the API call as a suspending function as follows:

```
interface GitHubService {
    @GET("orgs/{org}/repos?per_page=100")
    suspend fun getOrgRepos(
        @Path("org") org: String
    ): List<Repo>
}
```

- `getOrgRepos()` is defined as a `suspend` function. When you use a suspending function to perform a request, the underlying thread isn't blocked. More details about how this works will come in later sections.
- `getOrgRepos()` returns the result directly instead of returning a `Call`. If the result is unsuccessful, an exception is thrown.

Alternatively, Retrofit allows returning the result wrapped in `Response`. In this case, the result body is provided, and it is possible to check for errors manually. This tutorial uses the versions that return `Response`.

In `src/contributors/GitHubService.kt`, add the following declarations to the `GitHubService` interface:

```
interface GitHubService {
    // getOrgReposCall & getRepoContributorsCall declarations

    @GET("orgs/{org}/repos?per_page=100")
    suspend fun getOrgRepos(
        @Path("org") org: String
    ): Response<List<Repo>>

    @GET("repos/{owner}/{repo}/contributors?per_page=100")
    suspend fun getRepoContributors(
        @Path("owner") owner: String,
        @Path("repo") repo: String
    ): Response<List<User>>
}
```

Task 4

Your task is to change the code of the function that loads contributors to make use of two new suspending functions, `getOrgRepos()` and `getRepoContributors()`. The new `loadContributorsSuspend()` function is marked as `suspend` to use the new API.

> Suspending functions can't be called everywhere. Calling a suspending function from `loadContributorsBlocking()` will result in an error with the message "Suspend function 'getOrgRepos' should be called only from a coroutine or another suspend function".

1. Copy the implementation of `loadContributorsBlocking()` that is defined in `src/tasks/Request1Blocking.kt` into the `loadContributorsSuspend()` that is defined in `src/tasks/Request4Suspend.kt`.
2. Modify the code so that the new suspending functions are used instead of the ones that return `Call`s.
3. Run the program by choosing the *SUSPEND* option and ensure that the UI is still responsive while the GitHub requests are performed.

Solution for task 4

Replace `.getOrgReposCall(req.org).execute()` with `.getOrgRepos(req.org)` and repeat the same replacement for the second "contributors" request:

```
suspend fun loadContributorsSuspend(service: GitHubService, req: RequestData): List<User> {
    val repos = service
        .getOrgRepos(req.org)
        .also { logRepos(req, it) }
        .bodyList()

    return repos.flatMap { repo ->
        service.getRepoContributors(req.org, repo.name)
            .also { logUsers(repo, it) }
            .bodyList()
    }.aggregate()
}
```

- `loadContributorsSuspend()` should be defined as a `suspend` function.
-

You no longer need to call `execute`, which returned the `Response` before, because now the API functions return the `Response` directly. Note that this detail is specific to the Retrofit library. With other libraries, the API will be different, but the concept is the same.

Coroutines

The code with suspending functions looks similar to the "blocking" version. The major difference from the blocking version is that instead of blocking the thread, the coroutine is suspended:

```
block → suspend
thread → coroutine
```

Coroutines are often called lightweight threads because you can run code on coroutines, similar to how you run code on threads. The operations that were blocking before (and had to be avoided) can now suspend the coroutine instead.

Starting a new coroutine

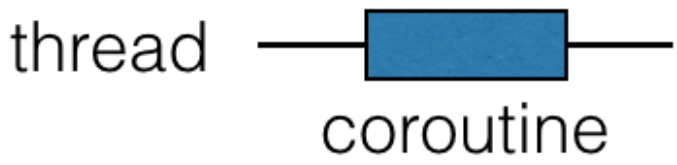
If you look at how `loadContributorsSuspend()` is used in `src/contributors/Contributors.kt`, you can see that it's called inside `launch`. `launch` is a library function that takes a lambda as an argument:

```
launch {
    val users = loadContributorsSuspend(req)
    updateResults(users, startTime)
}
```

Here `launch` starts a new computation that is responsible for loading the data and showing the results. The computation is suspendable – when performing network requests, it is suspended and releases the underlying thread. When the network request returns the result, the computation is resumed.

Such a suspendable computation is called a *coroutine*. So, in this case, `launch` *starts a new coroutine* responsible for loading data and showing the results.

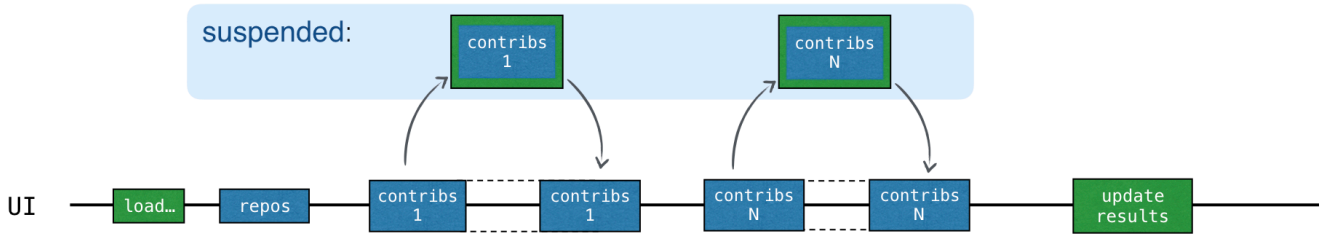
Coroutines run on top of threads and can be suspended. When a coroutine is suspended, the corresponding computation is paused, removed from the thread, and stored in memory. Meanwhile, the thread is free to be occupied by other tasks:



When the computation is ready to be continued, it is returned to a thread (not necessarily the same one).

In the `loadContributorsSuspend()` example, each "contributors" request now waits for the result using the suspension mechanism. First, the new request is sent. Then, while waiting for the response, the whole "load contributors" coroutine that was started by the `launch` function is suspended.

The coroutine resumes only after the corresponding response is received:



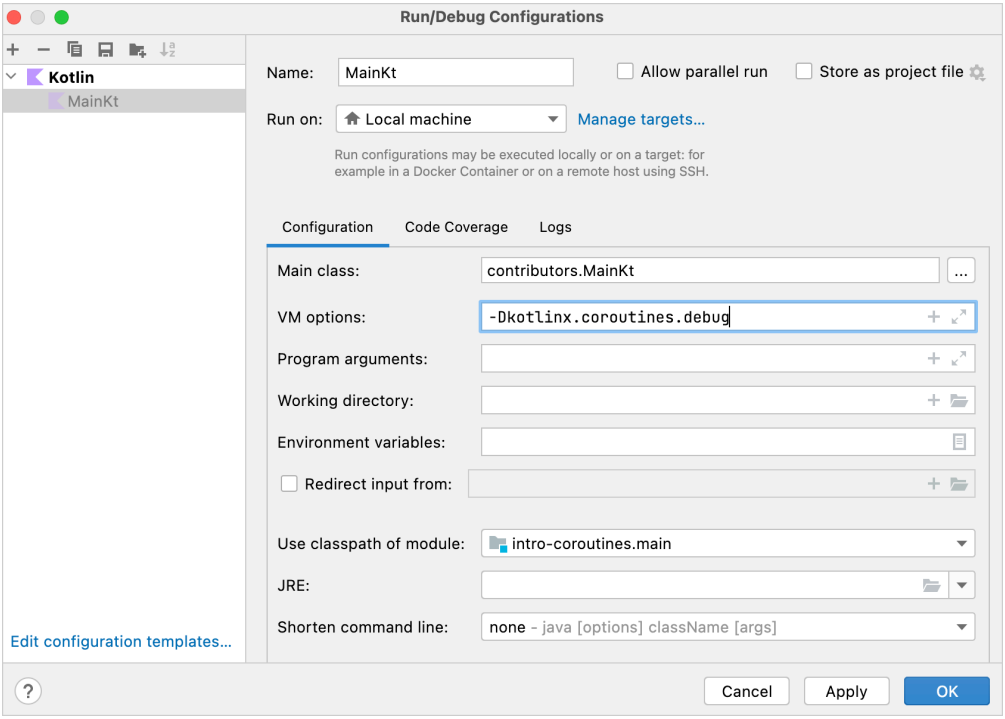
Suspending request

While the response is waiting to be received, the thread is free to be occupied by other tasks. The UI stays responsive, despite all the requests taking place on the main UI thread:

1. Run the program using the *SUSPEND* option. The log confirms that all of the requests are sent from the main UI thread:

```
2538 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin: loaded 30 repos
2729 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - ts2kt: loaded 11 contributors
3029 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin-koans: loaded 45 contributors
...
11252 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin-coroutines-workshop: loaded 1
contributors
```

2. The log can show you which coroutine the corresponding code is running on. To enable it, open **Run | Edit configurations** and add the `-Dkotlinx.coroutines.debug` VM option:



Edit run configuration

The coroutine name will be attached to the thread name while `main()` is run with this option. You can also modify the template for running all of the Kotlin files and enable this option by default.

Now all of the code runs on one coroutine, the "load contributors" coroutine mentioned above, denoted as `@coroutine#1`. While waiting for the result, you shouldn't reuse the thread for sending other requests because the code is written sequentially. The new request is sent only when the previous result is received.

Suspending functions treat the thread fairly and don't block it for "waiting". However, this doesn't yet bring any concurrency into the picture.

Concurrency

Kotlin coroutines are much less resource-intensive than threads. Each time you want to start a new computation asynchronously, you can create a new coroutine instead.

To start a new coroutine, use one of the main *coroutine builders*: `launch`, `async`, or `runBlocking`. Different libraries can define additional coroutine builders.

`async` starts a new coroutine and returns a `Deferred` object. `Deferred` represents a concept known by other names such as `Future` or `Promise`. It stores a computation, but it *defers* the moment you get the final result; it *promises* the result sometime in the *future*.

The main difference between `async` and `launch` is that `launch` is used to start a computation that isn't expected to return a specific result. `launch` returns a `Job` that represents the coroutine. It is possible to wait until it completes by calling `Job.join()`.

`Deferred` is a generic type that extends `Job`. An `async` call can return a `Deferred<Int>` or a `Deferred<CustomType>`, depending on what the lambda returns (the last expression inside the lambda is the result).

To get the result of a coroutine, you can call `await()` on the `Deferred` instance. While waiting for the result, the coroutine that this `await()` is called from is suspended:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val deferred: Deferred<Int> = async {
        loadData()
    }
    println("waiting...")
    println(deferred.await())
}

suspend fun loadData(): Int {
    println("loading...")
    delay(1000L)
    println("loaded!")
    return 42
}
```

`runBlocking` is used as a bridge between regular and suspending functions, or between the blocking and non-blocking worlds. It works as an adaptor for starting the top-level main coroutine. It is intended primarily to be used in `main()` functions and tests.

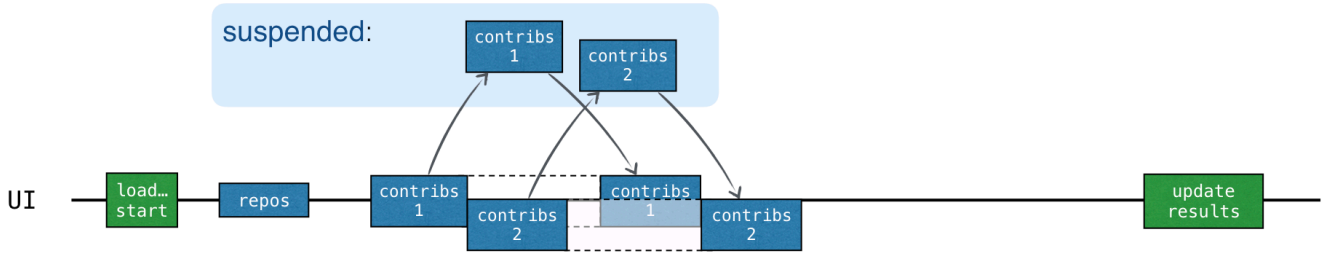
Watch this video for a better understanding of coroutines.

If there is a list of deferred objects, you can call `awaitAll()` to await the results of all of them:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val deferreds: List<Deferred<Int>> = (1..3).map {
        async {
            delay(1000L * it)
            println("Loading $it")
            it
        }
    }
    val sum = deferreds.awaitAll().sum()
    println("$sum")
}
```

When each "contributors" request is started in a new coroutine, all of the requests are started asynchronously. A new request can be sent before the result for the previous one is received:



Concurrent coroutines

The total loading time is approximately the same as in the *CALLBACKS* version, but it doesn't need any callbacks. What's more, `async` explicitly emphasizes which parts run concurrently in the code.

Task 5

In the `Request5Concurrent.kt` file, implement a `loadContributorsConcurrent()` function by using the previous `loadContributorsSuspend()` function.

Tip for task 5

You can only start a new coroutine inside a coroutine scope. Copy the content from `loadContributorsSuspend()` to the `coroutineScope` call so that you can call `async` functions there:

```
suspend fun loadContributorsConcurrent(
    service: GitHubService,
    req: RequestData
): List<User> = coroutineScope {
    // ...
}
```

Base your solution on the following scheme:

```
val deferreds: List<Deferred<List<User>>> = repos.map { repo ->
    async {
        // load contributors for each repo
    }
}
deferreds.awaitAll() // List<List<User>>
```

Solution for task 5

Wrap each "contributors" request with `async` to create as many coroutines as there are repositories. `async` returns `Deferred<List<User>>`. This is not an issue because creating new coroutines is not very resource-intensive, so you can create as many as you need.

1. You can no longer use `flatMap` because the `map` result is now a list of `Deferred` objects, not a list of lists. `awaitAll()` returns `List<List<User>>`, so call `flatten().aggregate()` to get the result:

   ```
   suspend fun loadContributorsConcurrent(
       service: GitHubService,
       req: RequestData
   ): List<User> = coroutineScope {
       val repos = service
           .getOrgRepos(req.org)
           .also { logRepos(req, it) }
           .bodyList()

       val deferreds: List<Deferred<List<User>>> = repos.map { repo ->
           async {
               service.getRepoContributors(req.org, repo.name)
                   .also { logUsers(repo, it) }
                   .bodyList()
           }
       }
       deferreds.awaitAll().flatten().aggregate()
   }
   ```

2. Run the code and check the log. All of the coroutines still run on the main UI thread because multithreading hasn't been employed yet, but you can already see the benefits of running coroutines concurrently.

3. To change this code to run "contributors" coroutines on different threads from the common thread pool, specify `Dispatchers.Default` as the context argument for the `async` function:

```
async(Dispatchers.Default) { }
```

- `CoroutineDispatcher` determines what thread or threads the corresponding coroutine should be run on. If you don't specify one as an argument, `async` will use the dispatcher from the outer scope.
- `Dispatchers.Default` represents a shared pool of threads on the JVM. This pool provides a means for parallel execution. It consists of as many threads as there are CPU cores available, but it will still have two threads if there's only one core.

4. Modify the code in the `loadContributorsConcurrent()` function to start new coroutines on different threads from the common thread pool. Also, add additional logging before sending the request:

```
async(Dispatchers.Default) {
    log("starting loading for ${repo.name}")
    service.getRepoContributors(req.org, repo.name)
        .also { logUsers(repo, it) }
        .bodyList()
}
```

5. Run the program once again. In the log, you can see that each coroutine can be started on one thread from the thread pool and resumed on another:

```
1946 [DefaultDispatcher-worker-2 @coroutine#4] INFO  Contributors - starting loading for kotlin-
koans
1946 [DefaultDispatcher-worker-3 @coroutine#5] INFO  Contributors - starting loading for dokka
1946 [DefaultDispatcher-worker-1 @coroutine#3] INFO  Contributors - starting loading for ts2kt
...
2178 [DefaultDispatcher-worker-1 @coroutine#4] INFO  Contributors - kotlin-koans: loaded 45
contributors
2569 [DefaultDispatcher-worker-1 @coroutine#5] INFO  Contributors - dokka: loaded 36 contributors
2821 [DefaultDispatcher-worker-2 @coroutine#3] INFO  Contributors - ts2kt: loaded 11 contributors
```

For instance, in this log excerpt, `coroutine#4` is started on the `worker-2` thread and continued on the `worker-1` thread.

In `src/contributors/Contributors.kt`, check the implementation of the *CONCURRENT* option:

1. To run the coroutine only on the main UI thread, specify `Dispatchers.Main` as an argument:

```
launch(Dispatchers.Main) {
    updateResults()
}
```

- If the main thread is busy when you start a new coroutine on it, the coroutine becomes suspended and scheduled for execution on this thread. The coroutine will only resume when the thread becomes free.
- It's considered good practice to use the dispatcher from the outer scope rather than explicitly specifying it on each end-point. If you define `loadContributorsConcurrent()` without passing `Dispatchers.Default` as an argument, you can call this function in any context: with a `Default` dispatcher, with the main UI thread, or with a custom dispatcher.
- As you'll see later, when calling `loadContributorsConcurrent()` from tests, you can call it in the context with `TestDispatcher`, which simplifies testing. That makes this solution much more flexible.

2. To specify the dispatcher on the caller side, apply the following change to the project while letting `loadContributorsConcurrent` start coroutines in the inherited context:

```
launch(Dispatchers.Default) {
    val users = loadContributorsConcurrent(service, req)
    withContext(Dispatchers.Main) {
        updateResults(users, startTime)
    }
}
```

   - `updateResults()` should be called on the main UI thread, so you call it with the context of `Dispatchers.Main`.
   - `withContext()` calls the given code with the specified coroutine context, is suspended until it completes, and returns the result. An alternative but more verbose way to express this would be to start a new coroutine and explicitly wait (by suspending) until it completes: `launch(context) { ... }.join()`.

3. Run the code and ensure that the coroutines are executed on the threads from the thread pool.

Structured concurrency

- The *coroutine scope* is responsible for the structure and parent-child relationships between different coroutines. New coroutines usually need to be started inside a scope.
- The *coroutine context* stores additional technical information used to run a given coroutine, like the coroutine custom name, or the dispatcher specifying the threads the coroutine should be scheduled on.

When `launch`, `async`, or `runBlocking` are used to start a new coroutine, they automatically create the corresponding scope. All of these functions take a lambda with a receiver as an argument, and `CoroutineScope` is the implicit receiver type:

```
launch { /* this: CoroutineScope */ }
```

- New coroutines can only be started inside a scope.
- `launch` and `async` are declared as extensions to `CoroutineScope`, so an implicit or explicit receiver must always be passed when you call them.
- The coroutine started by `runBlocking` is the only exception because `runBlocking` is defined as a top-level function. But because it blocks the current thread, it's intended primarily to be used in `main()` functions and tests as a bridge function.

A new coroutine inside `runBlocking`, `launch`, or `async` is started automatically inside the scope:

```
import kotlinx.coroutines.*

fun main() = runBlocking { /* this: CoroutineScope */
    launch { /* ... */ }
    // the same as:
    this.launch { /* ... */ }
}
```

When you call `launch` inside `runBlocking`, it's called as an extension to the implicit receiver of the `CoroutineScope` type. Alternatively, you could explicitly write `this.launch`.

The nested coroutine (started by `launch` in this example) can be considered as a child of the outer coroutine (started by `runBlocking`). This "parent-child" relationship works through scopes; the child coroutine is started from the scope corresponding to the parent coroutine.

It's possible to create a new scope without starting a new coroutine, by using the `coroutineScope` function. To start new coroutines in a structured way inside a `suspend` function without access to the outer scope, you can create a new coroutine scope that automatically becomes a child of the outer scope that this `suspend` function is called from. `loadContributorsConcurrent()` is a good example.

You can also start a new coroutine from the global scope using `GlobalScope.async` or `GlobalScope.launch`. This will create a top-level "independent" coroutine.

The mechanism behind the structure of the coroutines is called *structured concurrency*. It provides the following benefits over global scopes:

- The scope is generally responsible for child coroutines, whose lifetime is attached to the lifetime of the scope.
- The scope can automatically cancel child coroutines if something goes wrong or a user changes their mind and decides to revoke the operation.
- The scope automatically waits for the completion of all child coroutines. Therefore, if the scope corresponds to a coroutine, the parent coroutine does not complete until all the coroutines launched in its scope have completed.

When using `GlobalScope.async`, there is no structure that binds several coroutines to a smaller scope. Coroutines started from the global scope are all independent – their lifetime is limited only by the lifetime of the whole application. It's possible to store a reference to the coroutine started from the global scope and wait for its completion or cancel it explicitly, but that won't happen automatically as it would with structured concurrency.

Canceling the loading of contributors

Create two versions of the function that loads the list of contributors. Compare how both versions behave when you try to cancel the parent coroutine. The first version will use `coroutineScope` to start all of the child coroutines, whereas the second will use `GlobalScope`.

1. In `Request5Concurrent.kt`, add a 3-second delay to the `loadContributorsConcurrent()` function:

   ```
   suspend fun loadContributorsConcurrent(
       service: GitHubService,
       req: RequestData
   ): List<User> = coroutineScope {
       // ...
       async {
           log("starting loading for ${repo.name}")
           delay(3000)
           // load repo contributors
       }
       // ...
   }
   ```

   The delay affects all of the coroutines that send requests, so that there's enough time to cancel the loading after the coroutines are started but before the requests are sent.

2. Create the second version of the loading function: copy the implementation of `loadContributorsConcurrent()` to `loadContributorsNotCancellable()` in `Request5NotCancellable.kt` and then remove the creation of a new `coroutineScope`.

3. The `async` calls now fail to resolve, so start them by using `GlobalScope.async`:

```
suspend fun loadContributorsNotCancellable(
    service: GitHubService,
    req: RequestData
): List<User> {   // #1
    // ...
    GlobalScope.async {   // #2
        log("starting loading for ${repo.name}")
        // load repo contributors
    }
    // ...
    return deferreds.awaitAll().flatten().aggregate()  // #3
}
```

- The function now returns the result directly, not as the last expression inside the lambda (lines `#1` and `#3`).
- All of the "contributors" coroutines are started inside the `GlobalScope`, not as children of the coroutine scope (line `#2`).

4. Run the program and choose the *CONCURRENT* option to load the contributors.
5. Wait until all of the "contributors" coroutines are started, and then click *Cancel*. The log shows no new results, which means that all of the requests were indeed canceled:

```
2896 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin: loaded 40 repos
2901 [DefaultDispatcher-worker-2 @coroutine#4] INFO  Contributors - starting loading for kotlin-
koans
...
2909 [DefaultDispatcher-worker-5 @coroutine#36] INFO  Contributors - starting loading for mpp-
example
/* click on 'cancel' */
/* no requests are sent */
```

6. Repeat step 5, but this time choose the `NOT_CANCELLABLE` option:

```
2570 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin: loaded 30 repos
2579 [DefaultDispatcher-worker-1 @coroutine#4] INFO  Contributors - starting loading for kotlin-
koans
...
2586 [DefaultDispatcher-worker-6 @coroutine#36] INFO  Contributors - starting loading for mpp-
example
/* click on 'cancel' */
/* but all the requests are still sent: */
6402 [DefaultDispatcher-worker-5 @coroutine#4] INFO  Contributors - kotlin-koans: loaded 45
contributors
...
9555 [DefaultDispatcher-worker-8 @coroutine#36] INFO  Contributors - mpp-example: loaded 8
contributors
```

   In this case, no coroutines are canceled, and all the requests are still sent.

7. Check how the cancellation is triggered in the "contributors" program. When the *Cancel* button is clicked, the main "loading" coroutine is explicitly canceled and the child coroutines are canceled automatically:

```
interface Contributors {

    fun loadContributors() {
        // ...
        when (getSelectedVariant()) {
            CONCURRENT -> {
                launch {
                    val users = loadContributorsConcurrent(service, req)
                    updateResults(users, startTime)
                }.setUpCancellation()      // #1
            }
        }
    }

    private fun Job.setUpCancellation() {
        val loadingJob = this              // #2

        // cancel the loading job if the 'cancel' button was clicked:
        val listener = ActionListener {
            loadingJob.cancel()            // #3
            updateLoadingStatus(CANCELED)
        }
        // add a listener to the 'cancel' button:
        addCancelListener(listener)

        // update the status and remove the listener
        // after the loading job is completed
    }
}
```

The `launch` function returns an instance of `Job`. `Job` stores a reference to the "loading coroutine", which loads all of the data and updates the results. You can call the `setUpCancellation()` extension function on it (line `#1`), passing an instance of `Job` as a receiver.

Another way you could express this would be to explicitly write:

```
val job = launch { }
job.setUpCancellation()
```

- For readability, you could refer to the `setUpCancellation()` function receiver inside the function with the new `loadingJob` variable (line `#2`).
- Then you could add a listener to the *Cancel* button so that when it's clicked, the `loadingJob` is canceled (line `#3`).

With structured concurrency, you only need to cancel the parent coroutine and this automatically propagates cancellation to all of the child coroutines.

Using the outer scope's context

When you start new coroutines inside the given scope, it's much easier to ensure that all of them run with the same context. It is also much easier to replace the context if needed.

Now it's time to learn how using the dispatcher from the outer scope works. The new scope created by the `coroutineScope` or by the coroutine builders always inherits the context from the outer scope. In this case, the outer scope is the scope the `suspend loadContributorsConcurrent()` function was called from:

```
launch(Dispatchers.Default) {  // outer scope
    val users = loadContributorsConcurrent(service, req)
    // ...
}
```

All of the nested coroutines are automatically started with the inherited context. The dispatcher is a part of this context. That's why all of the coroutines started by `async` are started with the context of the default dispatcher:

```
suspend fun loadContributorsConcurrent(
    service: GitHubService, req: RequestData
): List<User> = coroutineScope {
    // this scope inherits the context from the outer scope
    // ...
    async {   // nested coroutine started with the inherited context
        // ...
    }
    // ...
}
```

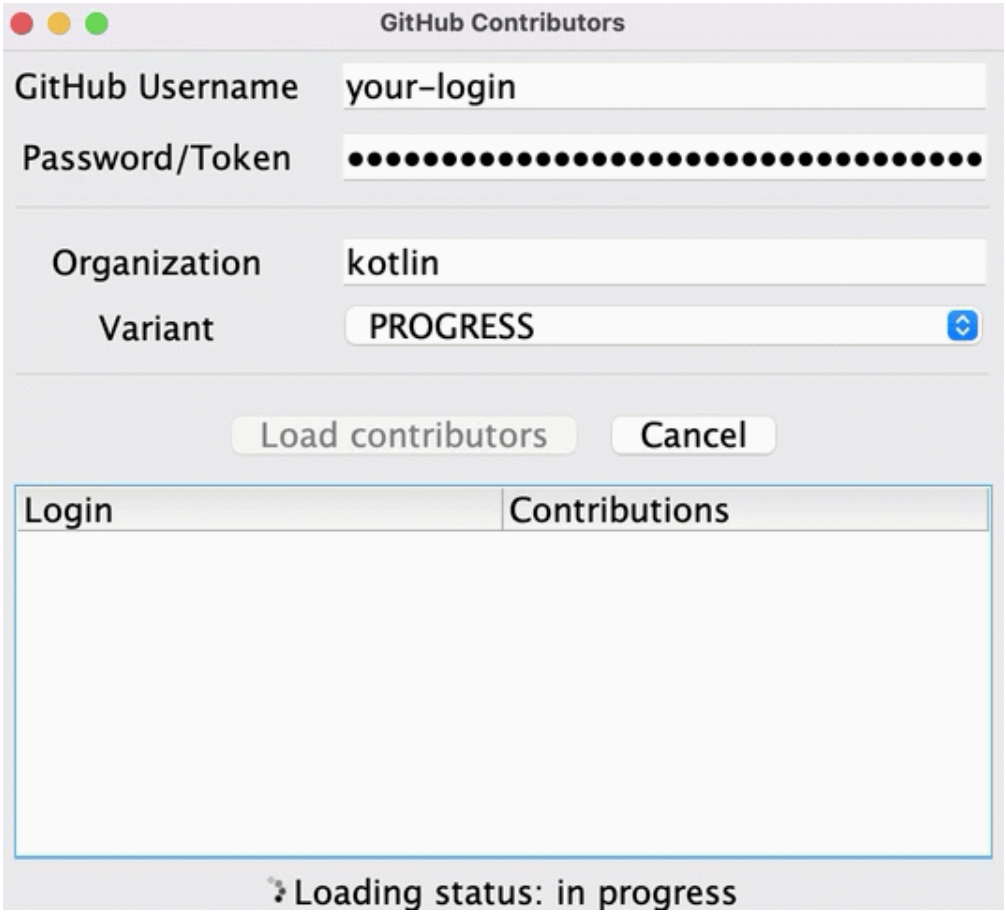
With structured concurrency, you can specify the major context elements (like dispatcher) once, when creating the top-level coroutine. All the nested coroutines then inherit the context and modify it only if needed.

When you write code with coroutines for UI applications, for example Android ones, it's a common practice to use `CoroutineDispatchers.Main` by default for the top coroutine and then to explicitly put a different dispatcher when you need to run the code on a different thread.

Showing progress

Despite the information for some repositories being loaded rather quickly, the user only sees the resulting list after all of the data has been loaded. Until then, the loader icon runs showing the progress, but there's no information about the current state or what contributors are already loaded.

You can show the intermediate results earlier and display all of the contributors after loading the data for each of the repositories:



To implement this functionality, in the `src/tasks/Request6Progress.kt`, you'll need to pass the logic updating the UI as a callback, so that it's called on each intermediate state:

```
suspend fun loadContributorsProgress(
    service: GitHubService,
    req: RequestData,
    updateResults: suspend (List<User>, completed: Boolean) -> Unit
) {
    // loading the data
    // calling `updateResults()` on intermediate states
}
```

On the call site in `Contributors.kt`, the callback is passed to update the results from the `Main` thread for the *PROGRESS* option:

```
launch(Dispatchers.Default) {
    loadContributorsProgress(service, req) { users, completed ->
        withContext(Dispatchers.Main) {
            updateResults(users, startTime, completed)
        }
    }
}
```

- The `updateResults()` parameter is declared as `suspend` in `loadContributorsProgress()`. It's necessary to call `withContext`, which is a `suspend` function inside the corresponding lambda argument.
- `updateResults()` callback takes an additional Boolean parameter as an argument specifying whether the loading has completed and the results are final.

Task 6

In the `Request6Progress.kt` file, implement the `loadContributorsProgress()` function that shows the intermediate progress. Base it on the `loadContributorsSuspend()` function from `Request4Suspend.kt`.

- Use a simple version without concurrency; you'll add it later in the next section.
- The intermediate list of contributors should be shown in an "aggregated" state, not just the list of users loaded for each repository.
- The total number of contributions for each user should be increased when the data for each new repository is loaded.

Solution for task 6

To store the intermediate list of loaded contributors in the "aggregated" state, define an `allUsers` variable which stores the list of users, and then update it after contributors for each new repository are loaded:

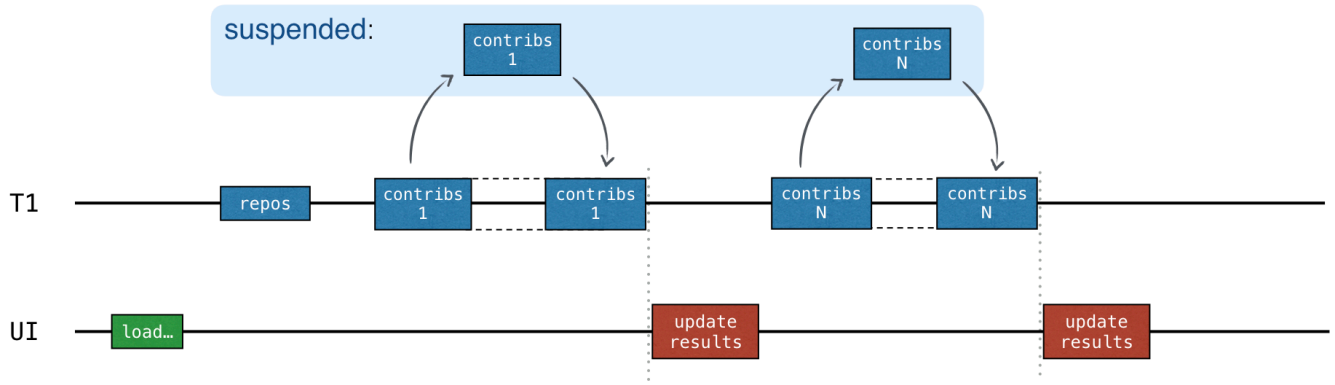
```
suspend fun loadContributorsProgress(
    service: GitHubService,
    req: RequestData,
    updateResults: suspend (List<User>, completed: Boolean) -> Unit
) {
    val repos = service
        .getOrgRepos(req.org)
        .also { logRepos(req, it) }
        .bodyList()

    var allUsers = emptyList<User>()
    for ((index, repo) in repos.withIndex()) {
        val users = service.getRepoContributors(req.org, repo.name)
            .also { logUsers(repo, it) }
            .bodyList()

        allUsers = (allUsers + users).aggregate()
        updateResults(allUsers, index == repos.lastIndex)
    }
}
```

Consecutive vs concurrent

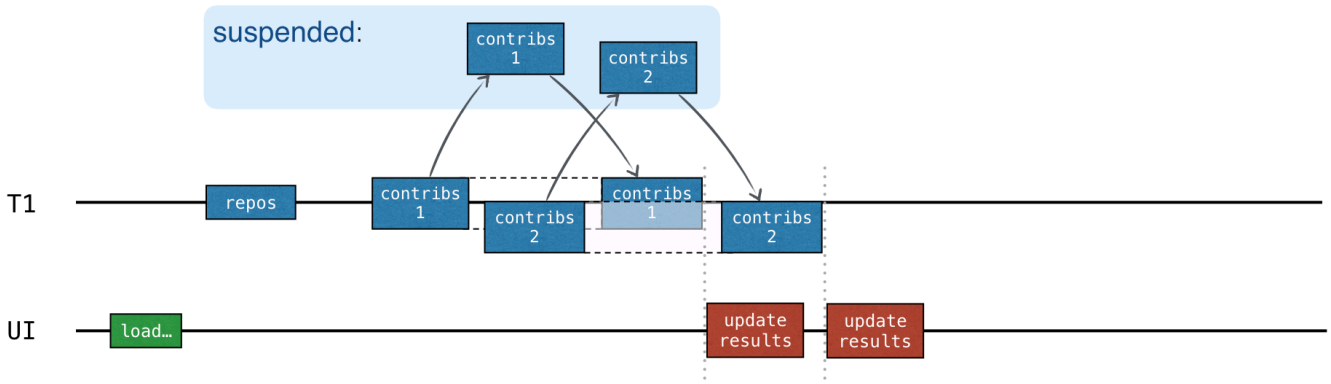
An `updateResults()` callback is called after each request is completed:



Progress on requests

This code doesn't include concurrency. It's sequential, so you don't need synchronization.

The best option would be to send requests concurrently and update the intermediate results after getting the response for each repository:



Concurrent requests

To add concurrency, use *channels*.

Channels

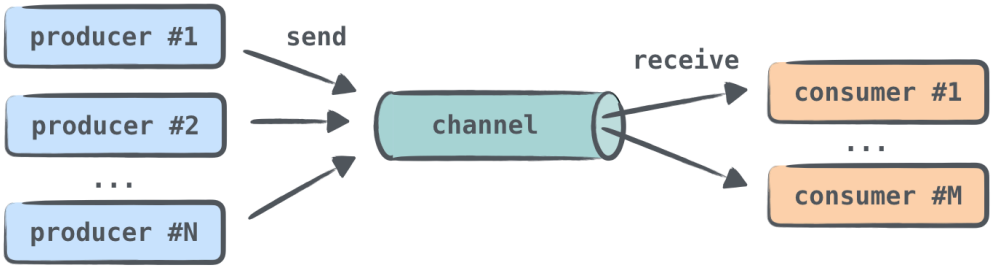
Writing code with a shared mutable state is quite difficult and error-prone (like in the solution using callbacks). A simpler way is to share information by communication rather than by using a common mutable state. Coroutines can communicate with each other through *channels*.

Channels are communication primitives that allow data to be passed between coroutines. One coroutine can *send* some information to a channel, while another can *receive* that information from it:



Using channels

A coroutine that sends (produces) information is often called a producer, and a coroutine that receives (consumes) information is called a consumer. One or multiple coroutines can send information to the same channel, and one or multiple coroutines can receive data from it:



Using channels with many coroutines

When many coroutines receive information from the same channel, each element is handled only once by one of the consumers. Once an element is handled, it is immediately removed from the channel.

You can think of a channel as similar to a collection of elements, or more precisely, a queue, in which elements are added to one end and received from the other. However, there's an important difference: unlike collections, even in their synchronized versions, a channel can *suspend* `send()` and `receive()` operations. This happens when the channel is empty or full. The channel can be full if the channel size has an upper bound.

`Channel` is represented by three different interfaces: `SendChannel`, `ReceiveChannel`, and `Channel`, with the latter extending the first two. You usually create a channel and give it to producers as a `SendChannel` instance so that only they can send information to the channel. You give a channel to consumers as a `ReceiveChannel` instance so that only they can receive from it. Both `send` and `receive` methods are declared as `suspend`:

```
interface SendChannel<in E> {
    suspend fun send(element: E)
    fun close(): Boolean
}

interface ReceiveChannel<out E> {
    suspend fun receive(): E
}

interface Channel<E> : SendChannel<E>, ReceiveChannel<E>
```

The producer can close a channel to indicate that no more elements are coming.

Several types of channels are defined in the library. They differ in how many elements they can internally store and whether the `send()` call can be suspended or not. For all of the channel types, the `receive()` call behaves similarly: it receives an element if the channel is not empty; otherwise, it is suspended.

Unlimited channel

An unlimited channel is the closest analog to a queue: producers can send elements to this channel and it will keep growing indefinitely. The `send()` call will never be suspended. If the program runs out of memory, you'll get an `OutOfMemoryException`. The difference between an unlimited channel and a queue is that when a consumer tries to receive from an empty channel, it becomes suspended until some new elements are sent.



Unlimited channel

Buffered channel

The size of a buffered channel is constrained by the specified number. Producers can send elements to this channel until the size limit is reached. All of the elements are internally stored. When the channel is full, the next `send()` call on it is suspended until more free space becomes available.



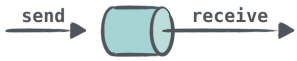
Buffered channel

Rendezvous channel

The "Rendezvous" channel is a channel without a buffer, the same as a buffered channel with zero size. One of the functions (`send()` or `receive()`) is always suspended until the other is called.

If the `send()` function is called and there's no suspended `receive()` call ready to process the element, then `send()` is suspended. Similarly, if the `receive()` function is called and the channel is empty or, in other words, there's no suspended `send()` call ready to send the element, the `receive()` call is suspended.

The "rendezvous" name ("a meeting at an agreed time and place") refers to the fact that `send()` and `receive()` should "meet on time".



Rendezvous channel

Conflated channel

A new element sent to the conflated channel will overwrite the previously sent element, so the receiver will always get only the latest element. The `send()` call is never suspended.



When you create a channel, specify its type or the buffer size (if you need a buffered one):

```
val rendezvousChannel = Channel<String>()
val bufferedChannel = Channel<String>(10)
val conflatedChannel = Channel<String>(CONFLATED)
val unlimitedChannel = Channel<String>(UNLIMITED)
```

By default, a "Rendezvous" channel is created.

In the following task, you'll create a "Rendezvous" channel, two producer coroutines, and a consumer coroutine:

```
import kotlinx.coroutines.channels.Channel
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    val channel = Channel<String>()
    launch {
        channel.send("A1")
        channel.send("A2")
        log("A done")
    }
    launch {
        channel.send("B1")
        log("B done")
    }
    launch {
        repeat(3) {
            val x = channel.receive()
            log(x)
        }
    }
}

fun log(message: Any?) {
    println("[${Thread.currentThread().name}] $message")
}
```

Watch this video for a better understanding of channels.

Task 7

In `src/tasks/Request7Channels.kt`, implement the function `loadContributorsChannels()` that requests all of the GitHub contributors concurrently and shows intermediate progress at the same time.

Use the previous functions, `loadContributorsConcurrent()` from `Request5Concurrent.kt` and `loadContributorsProgress()` from `Request6Progress.kt`.

Tip for task 7

Different coroutines that concurrently receive contributor lists for different repositories can send all of the received results to the same channel:

```
val channel = Channel<List<User>>()
for (repo in repos) {
    launch {
        val users = TODO()
        // ...
        channel.send(users)
    }
}
```

Then the elements from this channel can be received one by one and processed:

```
repeat(repos.size) {
    val users = channel.receive()
    // ...
}
```

Since the `receive()` calls are sequential, no additional synchronization is needed.

Solution for task 7

As with the `loadContributorsProgress()` function, you can create an `allUsers` variable to store the intermediate states of the "all contributors" list. Each new list received from the channel is added to the list of all users. You aggregate the result and update the state using the `updateResults` callback:

```
suspend fun loadContributorsChannels(
    service: GitHubService,
    req: RequestData,
    updateResults: suspend (List<User>, completed: Boolean) -> Unit
) = coroutineScope {

    val repos = service
        .getOrgRepos(req.org)
        .also { logRepos(req, it) }
        .bodyList()

    val channel = Channel<List<User>>()
    for (repo in repos) {
        launch {
            val users = service.getRepoContributors(req.org, repo.name)
                .also { logUsers(repo, it) }
                .bodyList()
            channel.send(users)
        }
    }
    var allUsers = emptyList<User>()
    repeat(repos.size) {
        val users = channel.receive()
        allUsers = (allUsers + users).aggregate()
        updateResults(allUsers, it == repos.lastIndex)
    }
}
```

- Results for different repositories are added to the channel as soon as they are ready. At first, when all of the requests are sent, and no data is received, the `receive()` call is suspended. In this case, the whole "load contributors" coroutine is suspended.
- Then, when the list of users is sent to the channel, the "load contributors" coroutine resumes, the `receive()` call returns this list, and the results are immediately updated.

You can now run the program and choose the *CHANNELS* option to load the contributors and see the result.

Although neither coroutines nor channels completely remove the complexity that comes with concurrency, they make life easier when you need to understand what's going on.

Testing coroutines

Let's now test all solutions to check that the solution with concurrent coroutines is faster than the solution with the `suspend` functions, and check that the solution with channels is faster than the simple "progress" one.

In the following task, you'll compare the total running time of the solutions. You'll mock a GitHub service and make this service return results after the given timeouts:

```
repos request - returns an answer within 1000 ms delay
repo-1 - 1000 ms delay
repo-2 - 1200 ms delay
repo-3 - 800 ms delay
```

The sequential solution with the `suspend` functions should take around 4000 ms (4000 = 1000 + (1000 + 1200 + 800)). The concurrent solution should take around 2200 ms (2200 = 1000 + max(1000, 1200, 800)).

For the solutions that show progress, you can also check the intermediate results with timestamps.

The corresponding test data is defined in `test/contributors/testData.kt`, and the files `Request4SuspendKtTest`, `Request7ChannelsKtTest`, and so on contain the straightforward tests that use mock service calls.

However, there are two problems here:

- These tests take too long to run. Each test takes around 2 to 4 seconds, and you need to wait for the results each time. It's not very efficient.
- You can't rely on the exact time the solution runs because it still takes additional time to prepare and run the code. You could add a constant, but then the time would differ from machine to machine. The mock service delays should be higher than this constant so you can see a difference. If the constant is 0.5 sec, making the delays 0.1 sec won't be enough.

A better way would be to use special frameworks to test the timing while running the same code several times (which increases the total time even more), but that is complicated to learn and set up.

To solve these problems and make sure that solutions with provided test delays behave as expected, one faster than the other, use *virtual* time with a special test dispatcher. This dispatcher keeps track of the virtual time passed from the start and runs everything immediately in real time. When you run coroutines on this dispatcher, the `delay` will return immediately and advance the virtual time.

Tests that use this mechanism run fast, but you can still check what happens at different moments in virtual time. The total running time drastically decreases:



Comparison for total running time

To use virtual time, replace the `runBlocking` invocation with a `runTest`. `runTest` takes an extension lambda to `TestScope` as an argument. When you call `delay` in a `suspend` function inside this special scope, `delay` will increase the virtual time instead of delaying in real time:

```
@Test
fun testDelayInSuspend() = runTest {
    val realStartTime = System.currentTimeMillis()
    val virtualStartTime = currentTime

    foo()
    println("${System.currentTimeMillis() - realStartTime} ms") // ~ 6 ms
    println("${currentTime - virtualStartTime} ms")             // 1000 ms
}

suspend fun foo() {
    delay(1000)    // auto-advances without delay
    println("foo") // executes eagerly when foo() is called
}
```

You can check the current virtual time using the `currentTime` property of `TestScope`.

The actual running time in this example is several milliseconds, whereas virtual time equals the delay argument, which is 1000 milliseconds.

To get the full effect of "virtual" `delay` in child coroutines, start all of the child coroutines with `TestDispatcher`. Otherwise, it won't work. This dispatcher is automatically inherited from the other `TestScope`, unless you provide a different dispatcher:

```
@Test
fun testDelayInLaunch() = runTest {
    val realStartTime = System.currentTimeMillis()
    val virtualStartTime = currentTime

    bar()

    println("${System.currentTimeMillis() - realStartTime} ms") // ~ 11 ms
    println("${currentTime - virtualStartTime} ms")             // 1000 ms
}

suspend fun bar() = coroutineScope {
    launch {
        delay(1000)    // auto-advances without delay
        println("bar") // executes eagerly when bar() is called
    }
}
```

If `launch` is called with the context of `Dispatchers.Default` in the example above, the test will fail. You'll get an exception saying that the job has not been completed yet.

You can test the `loadContributorsConcurrent()` function this way only if it starts the child coroutines with the inherited context, without modifying it using the `Dispatchers.Default` dispatcher.

You can specify the context elements like the dispatcher when *calling* a function rather than when *defining* it, which allows for more flexibility and easier testing.

The testing API that supports virtual time is Experimental and may change in the future.

By default, the compiler shows warnings if you use the experimental testing API. To suppress these warnings, annotate the test function or the whole class containing the tests with `@OptIn(ExperimentalCoroutinesApi::class)`. Add the compiler argument instructing the compiler that you're using the experimental API:

```
compileTestKotlin {
    kotlinOptions {
        freeCompilerArgs += "-Xuse-experimental=kotlin.Experimental"
    }
}
```

In the project corresponding to this tutorial, the compiler argument has already been added to the Gradle script.

Task 8

Refactor the following tests in `tests/tasks/` to use virtual time instead of real time:

- `Request4SuspendKtTest.kt`
- `Request5ConcurrentKtTest.kt`
- `Request6ProgressKtTest.kt`
- `Request7ChannelsKtTest.kt`

Compare the total running times before and after applying your refactoring.

Tip for task 8

1. Replace the `runBlocking` invocation with `runTest`, and replace `System.currentTimeMillis()` with `currentTime`:

```
@Test
fun test() = runTest {
    val startTime = currentTime
    // action
    val totalTime = currentTime - startTime
    // testing result
}
```

2. Uncomment the assertions that check the exact virtual time.

3. Don't forget to add `@UseExperimental(ExperimentalCoroutinesApi::class)`.

Solution for task 8

Here are the solutions for the concurrent and channels cases:

```
fun testConcurrent() = runTest {
    val startTime = currentTime
    val result = loadContributorsConcurrent(MockGithubService, testRequestData)
    Assert.assertEquals("Wrong result for 'loadContributorsConcurrent'",
expectedConcurrentResults.users, result)
    val totalTime = currentTime - startTime

    Assert.assertEquals(
        "The calls run concurrently, so the total virtual time should be 2200 ms: " +
                "1000 for repos request plus max(1000, 1200, 800) = 1200 for concurrent contributors
requests)",
        expectedConcurrentResults.timeFromStart, totalTime
    )
}
```

First, check that the results are available exactly at the expected virtual time, and then check the results themselves:

```
fun testChannels() = runTest {
    val startTime = currentTime
    var index = 0
    loadContributorsChannels(MockGithubService, testRequestData) { users, _ →
        val expected = concurrentProgressResults[index++]
        val time = currentTime - startTime
        Assert.assertEquals(
            "Expected intermediate results after ${expected.timeFromStart} ms:",
            expected.timeFromStart, time
        )
        Assert.assertEquals("Wrong intermediate results after $time:", expected.users, users)
    }
}
```

The first intermediate result for the last version with channels becomes available sooner than the progress version, and you can see the difference in tests that use virtual time.

The tests for the remaining "suspend" and "progress" tasks are very similar – you can find them in the project's `solutions` branch.

What's next

- Check out the Asynchronous Programming with Kotlin workshop at KotlinConf.
- Find out more about using virtual time and the experimental testing package.

Composing suspending functions

This section covers various approaches to composition of suspending functions.

Sequential by default

Assume that we have two suspending functions defined elsewhere that do something useful like some kind of remote service call or computation. We just pretend they are useful, but actually each one just delays for a second for the purpose of this example:

```
suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

What do we do if we need them to be invoked *sequentially* — first `doSomethingUsefulOne` *and then* `doSomethingUsefulTwo`, and compute the sum of their results? In practice, we do this if we use the result of the first function to make a decision on whether we need to invoke the second one or to decide on how to invoke it.

We use a normal sequential invocation, because the code in the coroutine, just like in the regular code, is *sequential* by default. The following example demonstrates it by measuring the total time it takes to execute both suspending functions:

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
    val time = measureTimeMillis {
        val one = doSomethingUsefulOne()
        val two = doSomethingUsefulTwo()
        println("The answer is ${one + two}")
    }
    println("Completed in $time ms")
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

You can get the full code here.

It produces something like this:

```
The answer is 42
Completed in 2017 ms
```

Concurrent using async

What if there are no dependencies between invocations of `doSomethingUsefulOne` and `doSomethingUsefulTwo` and we want to get the answer faster, by doing both *concurrently*? This is where async comes to help.

Conceptually, async is just like launch. It starts a separate coroutine which is a light-weight thread that works concurrently with all the other coroutines. The difference is that `launch` returns a Job and does not carry any resulting value, while `async` returns a Deferred — a light-weight non-blocking future that represents a promise to provide a result later. You can use `.await()` on a deferred value to get its eventual result, but `Deferred` is also a `Job`, so you can cancel it if needed.

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
    val time = measureTimeMillis {
        val one = async { doSomethingUsefulOne() }
        val two = async { doSomethingUsefulTwo() }
        println("The answer is ${one.await() + two.await()}")
    }
    println("Completed in $time ms")
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

You can get the full code here.

It produces something like this:

```
The answer is 42
Completed in 1017 ms
```

This is twice as fast, because the two coroutines execute concurrently. Note that concurrency with coroutines is always explicit.

Lazily started async

Optionally, async can be made lazy by setting its `start` parameter to CoroutineStart.LAZY. In this mode it only starts the coroutine when its result is required by await, or if its `Job` 's start function is invoked. Run the following example:

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
    val time = measureTimeMillis {
        val one = async(start = CoroutineStart.LAZY) { doSomethingUsefulOne() }
        val two = async(start = CoroutineStart.LAZY) { doSomethingUsefulTwo() }
        // some computation
        one.start() // start the first one
        two.start() // start the second one
        println("The answer is ${one.await() + two.await()}")
    }
    println("Completed in $time ms")
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

You can get the full code here.

It produces something like this:

```
The answer is 42
Completed in 1017 ms
```

So, here the two coroutines are defined but not executed as in the previous example, but the control is given to the programmer on when exactly to start the execution by calling start. We first start `one`, then start `two`, and then await for the individual coroutines to finish.

Note that if we just call await in `println` without first calling start on individual coroutines, this will lead to sequential behavior, since await starts the coroutine execution and waits for its finish, which is not the intended use-case for laziness. The use-case for `async(start = CoroutineStart.LAZY)` is a replacement for the standard lazy function in cases when computation of the value involves suspending functions.

Async-style functions

This programming style with async functions is provided here only for illustration, because it is a popular style in other programming languages. Using this style with Kotlin coroutines is **strongly discouraged** for the reasons explained below.

We can define async-style functions that invoke `doSomethingUsefulOne` and `doSomethingUsefulTwo` *asynchronously* using the async coroutine builder using a GlobalScope reference to opt-out of the structured concurrency. We name such functions with the "...Async" suffix to highlight the fact that they only start asynchronous computation and one needs to use the resulting deferred value to get the result.

GlobalScope is a delicate API that can backfire in non-trivial ways, one of which will be explained below, so you must explicitly opt-in into using `GlobalScope` with `@OptIn(DelicateCoroutinesApi::class)`.

```
// The result type of somethingUsefulOneAsync is Deferred<Int>
@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
}

// The result type of somethingUsefulTwoAsync is Deferred<Int>
@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulTwoAsync() = GlobalScope.async {
    doSomethingUsefulTwo()
}
```

Note that these `xxxAsync` functions are **not** *suspending* functions. They can be used from anywhere. However, their use always implies asynchronous (here meaning *concurrent*) execution of their action with the invoking code.

The following example shows their use outside of coroutine:

```
import kotlinx.coroutines.*
import kotlin.system.*

// note that we don't have `runBlocking` to the right of `main` in this example
fun main() {
    val time = measureTimeMillis {
        // we can initiate async actions outside of a coroutine
        val one = somethingUsefulOneAsync()
        val two = somethingUsefulTwoAsync()
        // but waiting for a result must involve either suspending or blocking.
        // here we use `runBlocking { ... }` to block the main thread while waiting for the result
        runBlocking {
            println("The answer is ${one.await() + two.await()}")
        }
    }
    println("Completed in $time ms")
}

@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
}

@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulTwoAsync() = GlobalScope.async {
    doSomethingUsefulTwo()
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

You can get the full code here.

Consider what happens if between the `val one = somethingUsefulOneAsync()` line and `one.await()` expression there is some logic error in the code, and the program throws an exception, and the operation that was being performed by the program aborts. Normally, a global error-handler could catch this exception, log and report the error for developers, but the program could otherwise continue doing other operations. However, here we have `somethingUsefulOneAsync` still running in the background, even though the operation that initiated it was aborted. This problem does not happen with structured concurrency, as shown in the section below.

Structured concurrency with async

Let's refactor the Concurrent using async example into a function that runs `doSomethingUsefulOne` and `doSomethingUsefulTwo` concurrently and returns their combined results. Since async is a CoroutineScope extension, we'll use the coroutineScope function to provide the necessary scope:

```
suspend fun concurrentSum(): Int = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    one.await() + two.await()
}
```

This way, if something goes wrong inside the code of the `concurrentSum` function, and it throws an exception, all the coroutines that were launched in its scope will be cancelled.

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
    val time = measureTimeMillis {
        println("The answer is ${concurrentSum()}")
    }
    println("Completed in $time ms")
}

suspend fun concurrentSum(): Int = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    one.await() + two.await()
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

You can get the full code here.

We still have concurrent execution of both operations, as evident from the output of the above `main` function:

```
The answer is 42
Completed in 1017 ms
```

Cancellation is always propagated through coroutines hierarchy:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    try {
        failedConcurrentSum()
    } catch(e: ArithmeticException) {
        println("Computation failed with ArithmeticException")
    }
}

suspend fun failedConcurrentSum(): Int = coroutineScope {
    val one = async<Int> {
        try {
            delay(Long.MAX_VALUE) // Emulates very long computation
            42
        } finally {
            println("First child was cancelled")
        }
    }
    val two = async<Int> {
        println("Second child throws an exception")
        throw ArithmeticException()
    }
    one.await() + two.await()
}
```

You can get the full code here.

Note how both the first `async` and the awaiting parent are cancelled on failure of one of the children (namely, `two`):

```
Second child throws an exception
First child was cancelled
Computation failed with ArithmeticException
```

Coroutine context and dispatchers

Coroutines always execute in some context represented by a value of the CoroutineContext type, defined in the Kotlin standard library.

The coroutine context is a set of various elements. The main elements are the Job of the coroutine, which we've seen before, and its dispatcher, which is covered in this section.

Dispatchers and threads

The coroutine context includes a *coroutine dispatcher* (see CoroutineDispatcher) that determines what thread or threads the corresponding coroutine uses for its execution. The coroutine dispatcher can confine coroutine execution to a specific thread, dispatch it to a thread pool, or let it run unconfined.

All coroutine builders like launch and async accept an optional CoroutineContext parameter that can be used to explicitly specify the dispatcher for the new coroutine and other context elements.

Try the following example:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    launch { // context of the parent, main runBlocking coroutine
        println("main runBlocking      : I'm working in thread ${Thread.currentThread().name}")
    }
    launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
        println("Unconfined            : I'm working in thread ${Thread.currentThread().name}")
    }
    launch(Dispatchers.Default) { // will get dispatched to DefaultDispatcher
        println("Default               : I'm working in thread ${Thread.currentThread().name}")
    }
    launch(newSingleThreadContext("MyOwnThread")) { // will get its own new thread
        println("newSingleThreadContext: I'm working in thread ${Thread.currentThread().name}")
    }
}
```

You can get the full code here.

It produces the following output (maybe in different order):

```
Unconfined            : I'm working in thread main
Default               : I'm working in thread DefaultDispatcher-worker-1
newSingleThreadContext: I'm working in thread MyOwnThread
main runBlocking      : I'm working in thread main
```

When `launch { ... }` is used without parameters, it inherits the context (and thus dispatcher) from the CoroutineScope it is being launched from. In this case, it inherits the context of the main `runBlocking` coroutine which runs in the `main` thread.

Dispatchers.Unconfined is a special dispatcher that also appears to run in the `main` thread, but it is, in fact, a different mechanism that is explained later.

The default dispatcher is used when no other dispatcher is explicitly specified in the scope. It is represented by Dispatchers.Default and uses a shared background pool of threads.

newSingleThreadContext creates a thread for the coroutine to run. A dedicated thread is a very expensive resource. In a real application it must be either released, when no longer needed, using the close function, or stored in a top-level variable and reused throughout the application.

Unconfined vs confined dispatcher

The Dispatchers.Unconfined coroutine dispatcher starts a coroutine in the caller thread, but only until the first suspension point. After suspension it resumes the coroutine in the thread that is fully determined by the suspending function that was invoked. The unconfined dispatcher is appropriate for coroutines which neither consume CPU time nor update any shared data (like UI) confined to a specific thread.

On the other side, the dispatcher is inherited from the outer CoroutineScope by default. The default dispatcher for the runBlocking coroutine, in particular, is confined to the invoker thread, so inheriting it has the effect of confining execution to this thread with predictable FIFO scheduling.

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
        println("Unconfined      : I'm working in thread ${Thread.currentThread().name}")
        delay(500)
        println("Unconfined      : After delay in thread ${Thread.currentThread().name}")
    }
    launch { // context of the parent, main runBlocking coroutine
        println("main runBlocking: I'm working in thread ${Thread.currentThread().name}")
        delay(1000)
        println("main runBlocking: After delay in thread ${Thread.currentThread().name}")
    }
}
```

You can get the full code here.

Produces the output:

```
Unconfined      : I'm working in thread main
main runBlocking: I'm working in thread main
Unconfined      : After delay in thread kotlinx.coroutines.DefaultExecutor
main runBlocking: After delay in thread main
```

So, the coroutine with the context inherited from `runBlocking {...}` continues to execute in the `main` thread, while the unconfined one resumes in the default executor thread that the delay function is using.

The unconfined dispatcher is an advanced mechanism that can be helpful in certain corner cases where dispatching of a coroutine for its execution later is not needed or produces undesirable side-effects, because some operation in a coroutine must be performed right away. The unconfined dispatcher should not be used in general code.

Debugging coroutines and threads

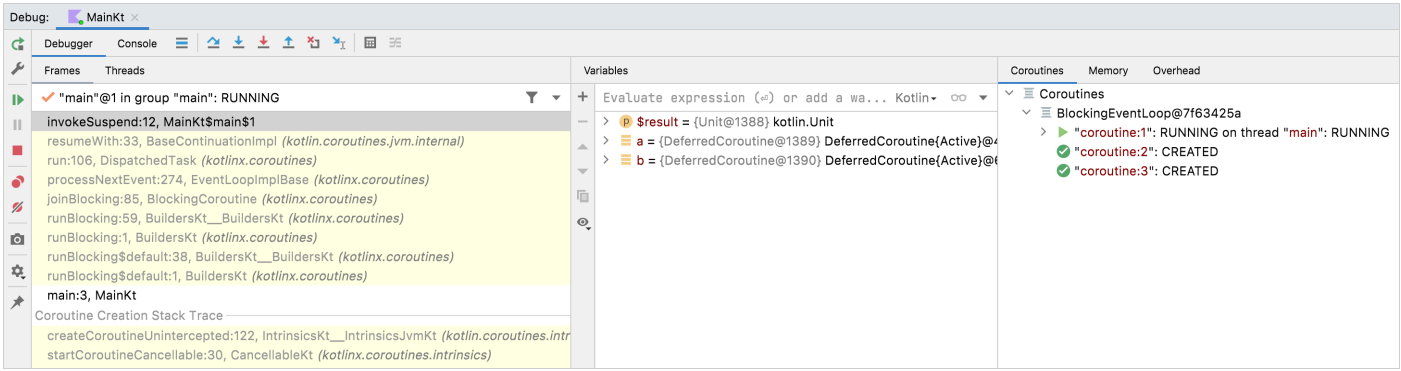
Coroutines can suspend on one thread and resume on another thread. Even with a single-threaded dispatcher it might be hard to figure out what the coroutine was doing, where, and when if you don't have special tooling.

Debugging with IDEA

The Coroutine Debugger of the Kotlin plugin simplifies debugging coroutines in IntelliJ IDEA.

Debugging works for versions 1.3.8 or later of `kotlinx-coroutines-core`.

The **Debug** tool window contains the **Coroutines** tab. In this tab, you can find information about both currently running and suspended coroutines. The coroutines are grouped by the dispatcher they are running on.



Debugging coroutines

With the coroutine debugger, you can:

- Check the state of each coroutine.
- See the values of local and captured variables for both running and suspended coroutines.
- See a full coroutine creation stack, as well as a call stack inside the coroutine. The stack includes all frames with variable values, even those that would be lost during standard debugging.
- Get a full report that contains the state of each coroutine and its stack. To obtain it, right-click inside the **Coroutines** tab, and then click **Get Coroutines Dump**.

To start coroutine debugging, you just need to set breakpoints and run the application in debug mode.

Learn more about coroutines debugging in the tutorial.

Debugging using logging

Another approach to debugging applications with threads without Coroutine Debugger is to print the thread name in the log file on each log statement. This feature is universally supported by logging frameworks. When using coroutines, the thread name alone does not give much of a context, so `kotlinx.coroutines` includes debugging facilities to make it easier.

Run the following code with `-Dkotlinx.coroutines.debug` JVM option:

```
import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() = runBlocking<Unit> {
    val a = async {
        log("I'm computing a piece of the answer")
        6
    }
    val b = async {
        log("I'm computing another piece of the answer")
        7
    }
    log("The answer is ${a.await() * b.await()}")
}
```

You can get the full code here.

There are three coroutines. The main coroutine (#1) inside `runBlocking` and two coroutines computing the deferred values `a` (#2) and `b` (#3). They are all executing in the context of `runBlocking` and are confined to the main thread. The output of this code is:

```
[main @coroutine#2] I'm computing a piece of the answer
[main @coroutine#3] I'm computing another piece of the answer
[main @coroutine#1] The answer is 42
```

The `log` function prints the name of the thread in square brackets, and you can see that it is the `main` thread with the identifier of the currently executing coroutine appended to it. This identifier is consecutively assigned to all created coroutines when the debugging mode is on.

> Debugging mode is also turned on when JVM is run with `-ea` option. You can read more about debugging facilities in the documentation of the DEBUG_PROPERTY_NAME property.

Jumping between threads

Run the following code with the `-Dkotlinx.coroutines.debug` JVM option (see debug):

```
import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() {
    newSingleThreadContext("Ctx1").use { ctx1 →
        newSingleThreadContext("Ctx2").use { ctx2 →
            runBlocking(ctx1) {
                log("Started in ctx1")
                withContext(ctx2) {
                    log("Working in ctx2")
                }
                log("Back to ctx1")
            }
        }
    }
}
```

> You can get the full code here.

The example above demonstrates new techniques in coroutine usage.

The first technique shows how to use runBlocking with a specified context.
The second technique involves calling withContext, which may suspend the current coroutine and switch to a new context—provided the new context differs from the existing one. Specifically, if you specify a different CoroutineDispatcher, extra dispatches are required: the block is scheduled on the new dispatcher, and once it finishes, execution returns to the original dispatcher.

As a result, the output of the above code is:

```
[Ctx1 @coroutine#1] Started in ctx1
[Ctx2 @coroutine#1] Working in ctx2
[Ctx1 @coroutine#1] Back to ctx1
```

The example above uses the `use` function from the Kotlin standard library to properly release thread resources created by newSingleThreadContext when they're no longer needed.

Job in the context

The coroutine's Job is part of its context, and can be retrieved from it using the `coroutineContext[Job]` expression:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    println("My job is ${coroutineContext[Job]}")
}
```

You can get the full code here.

In debug mode, it outputs something like this:

```
My job is "coroutine#1":BlockingCoroutine{Active}@6d311334
```

Note that isActive in CoroutineScope is just a convenient shortcut for `coroutineContext[Job]?.isActive == true`.

Children of a coroutine

When a coroutine is launched in the CoroutineScope of another coroutine, it inherits its context via CoroutineScope.coroutineContext and the Job of the new coroutine becomes a *child* of the parent coroutine's job. When the parent coroutine is cancelled, all its children are recursively cancelled, too.

However, this parent-child relation can be explicitly overridden in one of two ways:

1. When a different scope is explicitly specified when launching a coroutine (for example, `GlobalScope.launch`), it does not inherit a `Job` from the parent scope.
2. When a different `Job` object is passed as the context for the new coroutine (as shown in the example below), it overrides the `Job` of the parent scope.

In both cases, the launched coroutine is not tied to the scope it was launched from and operates independently.

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    // launch a coroutine to process some kind of incoming request
    val request = launch {
        // it spawns two other jobs
        launch(Job()) {
            println("job1: I run in my own Job and execute independently!")
            delay(1000)
            println("job1: I am not affected by cancellation of the request")
        }
        // and the other inherits the parent context
        launch {
            delay(100)
            println("job2: I am a child of the request coroutine")
            delay(1000)
            println("job2: I will not execute this line if my parent request is cancelled")
        }
    }
    delay(500)
    request.cancel() // cancel processing of the request
    println("main: Who has survived request cancellation?")
    delay(1000) // delay the main thread for a second to see what happens
}
```

You can get the full code here.

The output of this code is:

```
job1: I run in my own Job and execute independently!
job2: I am a child of the request coroutine
main: Who has survived request cancellation?
job1: I am not affected by cancellation of the request
```

Parental responsibilities

A parent coroutine always waits for the completion of all its children. A parent does not have to explicitly track all the children it launches, and it does not have to use Job.join to wait for them at the end:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    // launch a coroutine to process some kind of incoming request
    val request = launch {
        repeat(3) { i -> // launch a few children jobs
            launch  {
                delay((i + 1) * 200L) // variable delay 200ms, 400ms, 600ms
                println("Coroutine $i is done")
            }
        }
        println("request: I'm done and I don't explicitly join my children that are still active")
    }
    request.join() // wait for completion of the request, including all its children
    println("Now processing of the request is complete")
}
```

You can get the full code here.

The result is going to be:

```
request: I'm done and I don't explicitly join my children that are still active
Coroutine 0 is done
Coroutine 1 is done
Coroutine 2 is done
Now processing of the request is complete
```

Naming coroutines for debugging

Automatically assigned ids are good when coroutines log often and you just need to correlate log records coming from the same coroutine. However, when a coroutine is tied to the processing of a specific request or doing some specific background task, it is better to name it explicitly for debugging purposes. The CoroutineName context element serves the same purpose as the thread name. It is included in the thread name that is executing this coroutine when the debugging mode is turned on.

The following example demonstrates this concept:

```
import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() = runBlocking(CoroutineName("main")) {
    log("Started main coroutine")
    // run two background value computations
    val v1 = async(CoroutineName("v1coroutine")) {
        delay(500)
        log("Computing v1")
        6
    }
    val v2 = async(CoroutineName("v2coroutine")) {
        delay(1000)
        log("Computing v2")
        7
    }
    log("The answer for v1 * v2 = ${v1.await() * v2.await()}")
}
```

You can get the full code here.

The output it produces with `-Dkotlinx.coroutines.debug` JVM option is similar to:

```
[main @main#1] Started main coroutine
[main @v1coroutine#2] Computing v1
[main @v2coroutine#3] Computing v2
[main @main#1] The answer for v1 * v2 = 42
```

Combining context elements

Sometimes we need to define multiple elements for a coroutine context. We can use the `+` operator for that. For example, we can launch a coroutine with an explicitly specified dispatcher and an explicitly specified name at the same time:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    launch(Dispatchers.Default + CoroutineName("test")) {
        println("I'm working in thread ${Thread.currentThread().name}")
    }
}
```

You can get the full code here.

The output of this code with the `-Dkotlinx.coroutines.debug` JVM option is:

```
I'm working in thread DefaultDispatcher-worker-1 @test#2
```

Coroutine scope

Let us put our knowledge about contexts, children, and jobs together. Assume that our application has an object with a lifecycle, but that object is not a coroutine. For example, we are writing an Android application, and launching various coroutines in the context of an Android activity to perform asynchronous operations to fetch and update data, do animations, etc. These coroutines must be cancelled when the activity is destroyed to avoid memory leaks. We, of course, can manipulate contexts and jobs manually to tie the lifecycles of the activity and its coroutines, but `kotlinx.coroutines` provides an abstraction encapsulating that: CoroutineScope. You should be already familiar with the coroutine scope as all coroutine builders are declared as extensions on it.

We manage the lifecycles of our coroutines by creating an instance of CoroutineScope tied to the lifecycle of our activity. A `CoroutineScope` instance can be created by the CoroutineScope() or MainScope() factory functions. The former creates a general-purpose scope, while the latter creates a scope for UI applications and uses Dispatchers.Main as the default dispatcher:

```
class Activity {
    private val mainScope = MainScope()

    fun destroy() {
        mainScope.cancel()
    }
    // to be continued ...
```

Now, we can launch coroutines in the scope of this `Activity` using the defined `mainScope`. For the demo, we launch ten coroutines that delay for a different time:

```
// class Activity continues
    fun doSomething() {
        // launch ten coroutines for a demo, each working for a different time
        repeat(10) { i ->
            mainScope.launch {
                delay((i + 1) * 200L) // variable delay 200ms, 400ms, ... etc
                println("Coroutine $i is done")
            }
        }
    }
} // class Activity ends
```

In our main function we create the activity, call our test `doSomething` function, and destroy the activity after 500ms. This cancels all the coroutines that were launched from `doSomething`. We can see that because after the destruction of the activity, no more messages are printed, even if we wait a little longer.

```
import kotlinx.coroutines.*

class Activity {
    private val mainScope = CoroutineScope(Dispatchers.Default) // use Default for test purposes

    fun destroy() {
        mainScope.cancel()
    }

    fun doSomething() {
        // launch ten coroutines for a demo, each working for a different time
        repeat(10) { i ->
            mainScope.launch {
                delay((i + 1) * 200L) // variable delay 200ms, 400ms, ... etc
                println("Coroutine $i is done")
            }
        }
    }
} // class Activity ends

fun main() = runBlocking<Unit> {
    val activity = Activity()
    activity.doSomething() // run test function
    println("Launched coroutines")
    delay(500L) // delay for half a second
    println("Destroying activity!")
    activity.destroy() // cancels all coroutines
    delay(1000) // visually confirm that they don't work
}
```


You can get the full code here.

The output of this example is:

```
Launched coroutines
Coroutine 0 is done
Coroutine 1 is done
Destroying activity!
```

As you can see, only the first two coroutines print a message and the others are cancelled by a single invocation of `mainScope.cancel()` in `Activity.destroy()`.

Note that Android has first-party support for coroutine scope in all entities with the lifecycle. See the corresponding documentation.

Thread-local data

Sometimes it is convenient to be able to pass some thread-local data to or between coroutines. However, since they are not bound to any particular thread, this will likely lead to boilerplate if done manually.

For `ThreadLocal`, the asContextElement extension function is here for the rescue. It creates an additional context element which keeps the value of the given `ThreadLocal` and restores it every time the coroutine switches its context.

It is easy to demonstrate it in action:

```
import kotlinx.coroutines.*

val threadLocal = ThreadLocal<String?>() // declare thread-local variable

fun main() = runBlocking<Unit> {
    threadLocal.set("main")
    println("Pre-main, current thread: ${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
    val job = launch(Dispatchers.Default + threadLocal.asContextElement(value = "launch")) {
        println("Launch start, current thread: ${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
        yield()
        println("After yield, current thread: ${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
    }
    job.join()
    println("Post-main, current thread: ${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
}
```

You can get the full code here.

In this example, we launch a new coroutine in a background thread pool using Dispatchers.Default, so it works on different threads from the thread pool, but it still has the value of the thread local variable that we specified using `threadLocal.asContextElement(value = "launch")`, no matter which thread the coroutine is executed on. Thus, the output (with debug) is:

```
Pre-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'main'
Launch start, current thread: Thread[DefaultDispatcher-worker-1 @coroutine#2,5,main], thread local
value: 'launch'
After yield, current thread: Thread[DefaultDispatcher-worker-2 @coroutine#2,5,main], thread local
value: 'launch'
Post-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'main'
```

It's easy to forget to set the corresponding context element. The thread-local variable accessed from the coroutine may then have an unexpected value if the thread running the coroutine is different. To avoid such situations, it is recommended to use the ensurePresent method and fail-fast on improper usages.

`ThreadLocal` has first-class support and can be used with any primitive `kotlinx.coroutines` provides. It has one key limitation, though: when a thread-local is mutated, a new value is not propagated to the coroutine caller (because a context element cannot track all `ThreadLocal` object accesses), and the updated value is lost on the

next suspension. Use withContext to update the value of the thread-local in a coroutine, see asContextElement for more details.

Alternatively, a value can be stored in a mutable box like `class Counter(var i: Int)`, which is, in turn, stored in a thread-local variable. However, in this case, you are fully responsible to synchronize potentially concurrent modifications to the variable in this mutable box.

For advanced usage, for example, for integration with logging MDC, transactional contexts or any other libraries that internally use thread-locals for passing data, see the documentation of the ThreadContextElement interface that should be implemented.

Channels

Deferred values provide a convenient way to transfer a single value between coroutines. Channels provide a way to transfer a stream of values.

Channel basics

A Channel is conceptually very similar to `BlockingQueue`. One key difference is that instead of a blocking `put` operation it has a suspending send, and instead of a blocking `take` operation it has a suspending receive.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
    val channel = Channel<Int>()
    launch {
        // this might be heavy CPU-consuming computation or async logic,
        // we'll just send five squares
        for (x in 1..5) channel.send(x * x)
    }
    // here we print five received integers:
    repeat(5) { println(channel.receive()) }
    println("Done!")
}
```

You can get the full code here.

The output of this code is:

```
1
4
9
16
25
Done!
```

Closing and iteration over channels

Unlike a queue, a channel can be closed to indicate that no more elements are coming. On the receiver side it is convenient to use a regular `for` loop to receive elements from the channel.

Conceptually, a close is like sending a special close token to the channel. The iteration stops as soon as this close token is received, so there is a guarantee that all previously sent elements before the close are received:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
    val channel = Channel<Int>()
    launch {
        for (x in 1..5) channel.send(x * x)
        channel.close() // we're done sending
    }
    // here we print received values using `for` loop (until the channel is closed)
    for (y in channel) println(y)
    println("Done!")
}
```

You can get the full code here.

Building channel producers

The pattern where a coroutine is producing a sequence of elements is quite common. This is a part of *producer-consumer* pattern that is often found in concurrent code. You could abstract such a producer into a function that takes channel as its parameter, but this goes contrary to common sense that results must be returned from functions.

There is a convenient coroutine builder named produce that makes it easy to do it right on producer side, and an extension function consumeEach, that replaces a `for` loop on the consumer side:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun CoroutineScope.produceSquares(): ReceiveChannel<Int> = produce {
    for (x in 1..5) send(x * x)
}

fun main() = runBlocking {
    val squares = produceSquares()
    squares.consumeEach { println(it) }
    println("Done!")
}
```

You can get the full code here.

Pipelines

A pipeline is a pattern where one coroutine is producing, possibly infinite, stream of values:

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // infinite stream of integers starting from 1
}
```

And another coroutine or coroutines are consuming that stream, doing some processing, and producing some other results. In the example below, the numbers are just squared:

```
fun CoroutineScope.square(numbers: ReceiveChannel<Int>): ReceiveChannel<Int> = produce {
    for (x in numbers) send(x * x)
}
```

The main code starts and connects the whole pipeline:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
    val numbers = produceNumbers() // produces integers from 1 and on
    val squares = square(numbers) // squares integers
    repeat(5) {
        println(squares.receive()) // print first five
    }
    println("Done!") // we are done
    coroutineContext.cancelChildren() // cancel children coroutines
}

fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // infinite stream of integers starting from 1
}

fun CoroutineScope.square(numbers: ReceiveChannel<Int>): ReceiveChannel<Int> = produce {
    for (x in numbers) send(x * x)
}
```

You can get the full code here.

All functions that create coroutines are defined as extensions on CoroutineScope, so that we can rely on structured concurrency to make sure that we don't have lingering global coroutines in our application.

Prime numbers with pipeline

Let's take pipelines to the extreme with an example that generates prime numbers using a pipeline of coroutines. We start with an infinite sequence of numbers.

```
fun CoroutineScope.numbersFrom(start: Int) = produce<Int> {
    var x = start
    while (true) send(x++) // infinite stream of integers from start
}
```

The following pipeline stage filters an incoming stream of numbers, removing all the numbers that are divisible by the given prime number:

```
fun CoroutineScope.filter(numbers: ReceiveChannel<Int>, prime: Int) = produce<Int> {
    for (x in numbers) if (x % prime != 0) send(x)
}
```

Now we build our pipeline by starting a stream of numbers from 2, taking a prime number from the current channel, and launching new pipeline stage for each prime number found:

```
numbersFrom(2) → filter(2) → filter(3) → filter(5) → filter(7) ...
```

The following example prints the first ten prime numbers, running the whole pipeline in the context of the main thread. Since all the coroutines are launched in the scope of the main runBlocking coroutine we don't have to keep an explicit list of all the coroutines we have started. We use cancelChildren extension function to cancel all the children coroutines after we have printed the first ten prime numbers.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
    var cur = numbersFrom(2)
    repeat(10) {
        val prime = cur.receive()
        println(prime)
        cur = filter(cur, prime)
    }
    coroutineContext.cancelChildren() // cancel all children to let main finish
}

fun CoroutineScope.numbersFrom(start: Int) = produce<Int> {
    var x = start
    while (true) send(x++) // infinite stream of integers from start
}

fun CoroutineScope.filter(numbers: ReceiveChannel<Int>, prime: Int) = produce<Int> {
    for (x in numbers) if (x % prime ≠ 0) send(x)
}
```

You can get the full code here.

The output of this code is:

```
2
3
5
7
11
13
17
19
23
29
```

Note that you can build the same pipeline using `iterator` coroutine builder from the standard library. Replace `produce` with `iterator`, `send` with `yield`, `receive` with `next`, `ReceiveChannel` with `Iterator`, and get rid of the coroutine scope. You will not need `runBlocking` either. However, the benefit of a pipeline that uses channels as shown above is that it can actually use multiple CPU cores if you run it in Dispatchers.Default context.

Anyway, this is an extremely impractical way to find prime numbers. In practice, pipelines do involve some other suspending invocations (like asynchronous calls to remote services) and these pipelines cannot be built using `sequence` / `iterator`, because they do not allow arbitrary suspension, unlike `produce`, which is fully asynchronous.

Fan-out

Multiple coroutines may receive from the same channel, distributing work between themselves. Let us start with a producer coroutine that is periodically producing integers (ten numbers per second):

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1 // start from 1
    while (true) {
        send(x++) // produce next
        delay(100) // wait 0.1s
    }
}
```

Then we can have several processor coroutines. In this example, they just print their id and received number:

```
fun CoroutineScope.launchProcessor(id: Int, channel: ReceiveChannel<Int>) = launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
}
```

Now let us launch five processors and let them work for almost a second. See what happens:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking<Unit> {
    val producer = produceNumbers()
    repeat(5) { launchProcessor(it, producer) }
    delay(950)
    producer.cancel() // cancel producer coroutine and thus kill them all
}

fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1 // start from 1
    while (true) {
        send(x++) // produce next
        delay(100) // wait 0.1s
    }
}

fun CoroutineScope.launchProcessor(id: Int, channel: ReceiveChannel<Int>) = launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
}
```



You can get the full code here.

The output will be similar to the following one, albeit the processor ids that receive each specific integer may be different:

```
Processor #2 received 1
Processor #4 received 2
Processor #0 received 3
Processor #1 received 4
Processor #3 received 5
Processor #2 received 6
Processor #4 received 7
Processor #0 received 8
Processor #1 received 9
Processor #3 received 10
```

Note that cancelling a producer coroutine closes its channel, thus eventually terminating iteration over the channel that processor coroutines are doing.

Also, pay attention to how we explicitly iterate over channel with `for` loop to perform fan-out in `launchProcessor` code. Unlike `consumeEach`, this `for` loop pattern is perfectly safe to use from multiple coroutines. If one of the processor coroutines fails, then others would still be processing the channel, while a processor that is written via `consumeEach` always consumes (cancels) the underlying channel on its normal or abnormal completion.

Fan-in

Multiple coroutines may send to the same channel. For example, let us have a channel of strings, and a suspending function that repeatedly sends a specified string to this channel with a specified delay:

```
suspend fun sendString(channel: SendChannel<String>, s: String, time: Long) {
    while (true) {
        delay(time)
        channel.send(s)
    }
}
```

Now, let us see what happens if we launch a couple of coroutines sending strings (in this example we launch them in the context of the main thread as main coroutine's children):

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
    val channel = Channel<String>()
    launch { sendString(channel, "foo", 200L) }
    launch { sendString(channel, "BAR!", 500L) }
    repeat(6) { // receive first six
        println(channel.receive())
    }
    coroutineContext.cancelChildren() // cancel all children to let main finish
}

suspend fun sendString(channel: SendChannel<String>, s: String, time: Long) {
    while (true) {
        delay(time)
        channel.send(s)
    }
}
```

You can get the full code here.

The output is:

```
foo
foo
BAR!
foo
foo
BAR!
```

Buffered channels

The channels shown so far had no buffer. Unbuffered channels transfer elements when sender and receiver meet each other (aka rendezvous). If send is invoked first, then it is suspended until receive is invoked, if receive is invoked first, it is suspended until send is invoked.

Both Channel() factory function and produce builder take an optional `capacity` parameter to specify *buffer size*. Buffer allows senders to send multiple elements before suspending, similar to the `BlockingQueue` with a specified capacity, which blocks when buffer is full.

Take a look at the behavior of the following code:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking<Unit> {
    val channel = Channel<Int>(4) // create buffered channel
    val sender = launch { // launch sender coroutine
        repeat(10) {
            println("Sending $it") // print before sending each element
            channel.send(it) // will suspend when buffer is full
        }
    }
    // don't receive anything... just wait....
    delay(1000)
    sender.cancel() // cancel sender coroutine
}
```

You can get the full code here.

It prints "sending" *five* times using a buffered channel with capacity of *four*:

```
Sending 0
Sending 1
Sending 2
Sending 3
Sending 4
```

The first four elements are added to the buffer and the sender suspends when trying to send the fifth one.

Channels are fair

Send and receive operations to channels are *fair* with respect to the order of their invocation from multiple coroutines. They are served in first-in first-out order, e.g. the first coroutine to invoke `receive` gets the element.

In the following example two coroutines "ping" and "pong" are receiving the "ball" object from the shared "table" channel.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

data class Ball(var hits: Int)

fun main() = runBlocking {
    val table = Channel<Ball>() // a shared table
    launch { player("ping", table) }
    launch { player("pong", table) }
    table.send(Ball(0)) // serve the ball
    delay(1000) // delay 1 second
    coroutineContext.cancelChildren() // game over, cancel them
}

suspend fun player(name: String, table: Channel<Ball>) {
    for (ball in table) { // receive the ball in a loop
        ball.hits++
        println("$name $ball")
        delay(300) // wait a bit
        table.send(ball) // send the ball back
    }
}
```

You can get the full code here.

The "ping" coroutine is started first, so it is the first one to receive the ball. Even though "ping" coroutine immediately starts receiving the ball again after sending it back to the table, the ball gets received by the "pong" coroutine, because it was already waiting for it:

```
ping Ball(hits=1)
pong Ball(hits=2)
ping Ball(hits=3)
pong Ball(hits=4)
```

Note that sometimes channels may produce executions that look unfair due to the nature of the executor that is being used. See this issue for details.

Ticker channels

Ticker channel is a special rendezvous channel that produces `Unit` every time given delay passes since last consumption from this channel. Though it may seem to be useless standalone, it is a useful building block to create complex time-based produce pipelines and operators that do windowing and other time-dependent processing. Ticker channel can be used in select to perform "on tick" action.

To create such channel use a factory method ticker. To indicate that no further elements are needed use ReceiveChannel.cancel method on it.

Now let's see how it works in practice:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking<Unit> {
    val tickerChannel = ticker(delayMillis = 200, initialDelayMillis = 0) // create a ticker channel
    var nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
    println("Initial element is available immediately: $nextElement") // no initial delay

    nextElement = withTimeoutOrNull(100) { tickerChannel.receive() } // all subsequent elements have
200ms delay
    println("Next element is not ready in 100 ms: $nextElement")

    nextElement = withTimeoutOrNull(120) { tickerChannel.receive() }
    println("Next element is ready in 200 ms: $nextElement")

    // Emulate large consumption delays
    println("Consumer pauses for 300ms")
    delay(300)
    // Next element is available immediately
    nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
    println("Next element is available immediately after large consumer delay: $nextElement")
    // Note that the pause between `receive` calls is taken into account and next element arrives
faster
    nextElement = withTimeoutOrNull(120) { tickerChannel.receive() }
    println("Next element is ready in 100ms after consumer pause in 300ms: $nextElement")

    tickerChannel.cancel() // indicate that no more elements are needed
}
```

You can get the full code here.

It prints following lines:

```
Initial element is available immediately: kotlin.Unit
Next element is not ready in 100 ms: null
Next element is ready in 200 ms: kotlin.Unit
Consumer pauses for 300ms
Next element is available immediately after large consumer delay: kotlin.Unit
Next element is ready in 100ms after consumer pause in 300ms: kotlin.Unit
```

Note that ticker is aware of possible consumer pauses and, by default, adjusts next produced element delay if a pause occurs, trying to maintain a fixed rate of produced elements.

Optionally, a `mode` parameter equal to TickerMode.FIXED_DELAY can be specified to maintain a fixed delay between elements.

Coroutine exceptions handling

This section covers exception handling and cancellation on exceptions. We already know that a cancelled coroutine throws CancellationException in suspension points and that it is ignored by the coroutines' machinery. Here we look at what happens if an exception is thrown during cancellation or multiple children of the same coroutine throw an exception.

Exception propagation

Coroutine builders come in two flavors: propagating exceptions automatically (launch) or exposing them to users (async and produce). When these builders are used to create a *root* coroutine, that is not a *child* of another coroutine, the former builders treat exceptions as **uncaught** exceptions, similar to Java's `Thread.uncaughtExceptionHandler`, while the latter are relying on the user to consume the final exception, for example via await or receive (produce and receive are covered in Channels section).

It can be demonstrated by a simple example that creates root coroutines using the GlobalScope:

> GlobalScope is a delicate API that can backfire in non-trivial ways. Creating a root coroutine for the whole application is one of the rare legitimate uses for `GlobalScope`, so you must explicitly opt-in into using `GlobalScope` with `@OptIn(DelicateCoroutinesApi::class)`.

```
import kotlinx.coroutines.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
    val job = GlobalScope.launch { // root coroutine with launch
        println("Throwing exception from launch")
        throw IndexOutOfBoundsException() // Will be printed to the console by
Thread.defaultUncaughtExceptionHandler
    }
    job.join()
    println("Joined failed job")
    val deferred = GlobalScope.async { // root coroutine with async
        println("Throwing exception from async")
        throw ArithmeticException() // Nothing is printed, relying on user to call await
    }
    try {
        deferred.await()
        println("Unreached")
    } catch (e: ArithmeticException) {
        println("Caught ArithmeticException")
    }
}
```

> You can get the full code here.

The output of this code is (with debug):

```
Throwing exception from launch
Exception in thread "DefaultDispatcher-worker-1 @coroutine#2" java.lang.IndexOutOfBoundsException
Joined failed job
Throwing exception from async
Caught ArithmeticException
```

CoroutineExceptionHandler

It is possible to customize the default behavior of printing **uncaught** exceptions to the console. CoroutineExceptionHandler context element on a *root* coroutine can be used as a generic `catch` block for this root coroutine and all its children where custom exception handling may take place. It is similar to `Thread.uncaughtExceptionHandler`. You cannot recover from the exception in the

`CoroutineExceptionHandler`. The coroutine had already completed with the corresponding exception when the handler is called. Normally, the handler is used to log the exception, show some kind of error message, terminate, and/or restart the application.

`CoroutineExceptionHandler` is invoked only on **uncaught** exceptions — exceptions that were not handled in any other way. In particular, all *children* coroutines (coroutines created in the context of another Job) delegate handling of their exceptions to their parent coroutine, which also delegates to the parent, and so on until the root, so the `CoroutineExceptionHandler` installed in their context is never used. In addition to that, async builder always catches all exceptions and represents them in the resulting Deferred object, so its `CoroutineExceptionHandler` has no effect either.

Coroutines running in supervision scope do not propagate exceptions to their parent and are excluded from this rule. A further Supervision section of this document gives more details.

```
import kotlinx.coroutines.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception →
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) { // root coroutine, running in GlobalScope
        throw AssertionError()
    }
    val deferred = GlobalScope.async(handler) { // also root, but async instead of launch
        throw ArithmeticException() // Nothing will be printed, relying on user to call
deferred.await()
    }
    joinAll(job, deferred)
}
```

You can get the full code here.

The output of this code is:

```
CoroutineExceptionHandler got java.lang.AssertionError
```

Cancellation and exceptions

Cancellation is closely related to exceptions. Coroutines internally use `CancellationException` for cancellation, these exceptions are ignored by all handlers, so they should be used only as the source of additional debug information, which can be obtained by `catch` block. When a coroutine is cancelled using Job.cancel, it terminates, but it does not cancel its parent.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job = launch {
        val child = launch {
            try {
                delay(Long.MAX_VALUE)
            } finally {
                println("Child is cancelled")
            }
        }
        yield()
        println("Cancelling child")
        child.cancel()
        child.join()
        yield()
        println("Parent is not cancelled")
    }
    job.join()
}
```



You can get the full code here.

The output of this code is:

```
Cancelling child
Child is cancelled
Parent is not cancelled
```

If a coroutine encounters an exception other than `CancellationException`, it cancels its parent with that exception. This behaviour cannot be overridden and is used to provide stable coroutines hierarchies for structured concurrency. CoroutineExceptionHandler implementation is not used for child coroutines.



In these examples, CoroutineExceptionHandler is always installed to a coroutine that is created in GlobalScope. It does not make sense to install an exception handler to a coroutine that is launched in the scope of the main runBlocking, since the main coroutine is going to be always cancelled when its child completes with exception despite the installed handler.

The original exception is handled by the parent only when all its children terminate, which is demonstrated by the following example.

```
import kotlinx.coroutines.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) {
        launch { // the first child
            try {
                delay(Long.MAX_VALUE)
            } finally {
                withContext(NonCancellable) {
                    println("Children are cancelled, but exception is not handled until all children
terminate")
                    delay(100)
                    println("The first child finished its non cancellable block")
                }
            }
        }
        launch { // the second child
            delay(10)
            println("Second child throws an exception")
            throw ArithmeticException()
        }
    }
    job.join()
}
```

You can get the full code here.

The output of this code is:

```
Second child throws an exception
Children are cancelled, but exception is not handled until all children terminate
The first child finished its non cancellable block
CoroutineExceptionHandler got java.lang.ArithmeticException
```

Exceptions aggregation

When multiple children of a coroutine fail with an exception, the general rule is "the first exception wins", so the first exception gets handled. All additional exceptions that happen after the first one are attached to the first exception as suppressed ones.

```
import kotlinx.coroutines.*
import java.io.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception with suppressed
${exception.suppressed.contentToString()}")
    }
    val job = GlobalScope.launch(handler) {
        launch {
            try {
                delay(Long.MAX_VALUE) // it gets cancelled when another sibling fails with IOException
            } finally {
                throw ArithmeticException() // the second exception
            }
        }
        launch {
            delay(100)
            throw IOException() // the first exception
        }
        delay(Long.MAX_VALUE)
    }
    job.join()
}
```

You can get the full code here.

The output of this code is:

```
CoroutineExceptionHandler got java.io.IOException with suppressed [java.lang.ArithmeticException]
```

Note that this mechanism currently only works on Java version 1.7+. The JS and Native restrictions are temporary and will be lifted in the future.

Cancellation exceptions are transparent and are unwrapped by default:

```
import kotlinx.coroutines.*
import java.io.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) {
        val innerJob = launch { // all this stack of coroutines will get cancelled
            launch {
                launch {
                    throw IOException() // the original exception
                }
            }
        }
        try {
            innerJob.join()
        } catch (e: CancellationException) {
            println("Rethrowing CancellationException with original cause")
            throw e // cancellation exception is rethrown, yet the original IOException gets to the
handler
        }
    }
    job.join()
}
```

You can get the full code here.

The output of this code is:

```
Rethrowing CancellationException with original cause
CoroutineExceptionHandler got java.io.IOException
```

Supervision

As we have studied before, a failure of a coroutine is a bidirectional relationship propagating through the whole hierarchy of coroutines. Let us take a look at the case when unidirectional failure propagation is required.

A good example of such a requirement is a UI component with the job defined in its scope. If any of the UI's child tasks have failed, it is not always necessary to cancel (effectively kill) the whole UI component, but if the UI component is destroyed (and its job is cancelled), then it is necessary to cancel all child jobs as their results are no longer needed.

Another example is a server process that spawns multiple child jobs and needs to *supervise* their execution, tracking their failures and only restarting the failed ones.

Supervision job

The SupervisorJob can be used for these purposes. It is similar to a regular Job with the only exception that a failure or cancellation of a child does not propagate to the supervisor job or its other children. This can easily be demonstrated using the following example:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val supervisor = SupervisorJob()
    with(CoroutineScope(coroutineContext + supervisor)) {
        // launch the first child -- its exception is ignored for this example (don't do this in
practice!)
        val firstChild = launch(CoroutineExceptionHandler { _, _ -> }) {
            println("The first child is failing")
            throw AssertionError("The first child is cancelled")
        }
        // launch the second child
        val secondChild = launch {
            firstChild.join()
            // Cancellation of the first child is not propagated to the second child
            println("The first child is cancelled: ${firstChild.isCancelled}, but the second one is
still active")
            try {
                delay(Long.MAX_VALUE)
            } finally {
                // But cancellation of the supervisor is propagated
                println("The second child is cancelled because the supervisor was cancelled")
            }
        }
        // wait until the first child fails & completes
        firstChild.join()
        println("Cancelling the supervisor")
        supervisor.cancel()
        secondChild.join()
    }
}
```


You can get the full code here.

The output of this code is:

```
The first child is failing
The first child is cancelled: true, but the second one is still active
Cancelling the supervisor
The second child is cancelled because the supervisor was cancelled
```

Supervision scope

Instead of coroutineScope, we can use supervisorScope for *scoped* concurrency. It propagates the cancellation in one direction only and cancels all its children only if it failed itself. It also waits for all children before completion just like coroutineScope does.

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
    try {
        supervisorScope {
            val child = launch {
                try {
                    println("The child is sleeping")
                    delay(Long.MAX_VALUE)
                } finally {
                    println("The child is cancelled")
                }
            }
            // Give our child a chance to execute and print using yield
            yield()
            println("Throwing an exception from the scope")
            throw AssertionError()
        }
    } catch(e: AssertionError) {
        println("Caught an assertion error")
    }
}
```

You can get the full code here.

The output of this code is:

```
The child is sleeping
Throwing an exception from the scope
The child is cancelled
Caught an assertion error
```

Exceptions in supervised coroutines

Another crucial difference between regular and supervisor jobs is exception handling. Every child should handle its exceptions by itself via the exception handling mechanism. This difference comes from the fact that child's failure does not propagate to the parent. It means that coroutines launched directly inside the supervisorScope *do* use the CoroutineExceptionHandler that is installed in their scope in the same way as root coroutines do (see the CoroutineExceptionHandler section for details).

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    supervisorScope {
        val child = launch(handler) {
            println("The child throws an exception")
            throw AssertionError()
        }
        println("The scope is completing")
    }
    println("The scope is completed")
}
```

You can get the full code here.

The output of this code is:

```
The scope is completing
The child throws an exception
CoroutineExceptionHandler got java.lang.AssertionError
The scope is completed
```

Shared mutable state and concurrency

Coroutines can be executed parallelly using a multi-threaded dispatcher like the Dispatchers.Default. It presents all the usual parallelism problems. The main problem being synchronization of access to **shared mutable state**. Some solutions to this problem in the land of coroutines are similar to the solutions in the multi-threaded world, but others are unique.

The problem

Let us launch a hundred coroutines all doing the same action a thousand times. We'll also measure their completion time for further comparisons:

```
suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}
```

We start with a very simple action that increments a shared mutable variable using multi-threaded Dispatchers.Default.

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
```

You can get the full code here.

What does it print at the end? It is highly unlikely to ever print "Counter = 100000", because a hundred coroutines increment the `counter` concurrently from multiple threads without any synchronization.

Volatiles are of no help

There is a common misconception that making a variable `volatile` solves concurrency problem. Let us try it:

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

@Volatile // in Kotlin `volatile` is an annotation
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
```

You can get the full code here.

This code works slower, but we still don't always get "Counter = 100000" at the end, because volatile variables guarantee linearizable (this is a technical term for "atomic") reads and writes to the corresponding variable, but do not provide atomicity of larger actions (increment in our case).

Thread-safe data structures

The general solution that works both for threads and for coroutines is to use a thread-safe (aka synchronized, linearizable, or atomic) data structure that provides all the necessary synchronization for the corresponding operations that needs to be performed on a shared state. In the case of a simple counter we can use `AtomicInteger` class which has atomic `incrementAndGet` operations:

```
import kotlinx.coroutines.*
import java.util.concurrent.atomic.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

val counter = AtomicInteger()

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter.incrementAndGet()
        }
    }
    println("Counter = $counter")
}
```

You can get the full code here.

This is the fastest solution for this particular problem. It works for plain counters, collections, queues and other standard data structures and basic operations on them. However, it does not easily scale to complex state or to complex operations that do not have ready-to-use thread-safe implementations.

Thread confinement fine-grained

Thread confinement is an approach to the problem of shared mutable state where all access to the particular shared state is confined to a single thread. It is typically used in UI applications, where all UI state is confined to the single event-dispatch/application thread. It is easy to apply with coroutines by using a single-threaded context.

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            // confine each increment to a single-threaded context
            withContext(counterContext) {
                counter++
            }
        }
    }
    println("Counter = $counter")
}
```

You can get the full code here.

This code works very slowly, because it does *fine-grained* thread-confinement. Each individual increment switches from multi-threaded Dispatchers.Default context to the single-threaded context using withContext(counterContext) block.

Thread confinement coarse-grained

In practice, thread confinement is performed in large chunks, e.g. big pieces of state-updating business logic are confined to the single thread. The following example does it like that, running each coroutine in the single-threaded context to start with.

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking {
    // confine everything to a single-threaded context
    withContext(counterContext) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
```

You can get the full code here.

This now works much faster and produces correct result.

Mutual exclusion

Mutual exclusion solution to the problem is to protect all modifications of the shared state with a *critical section* that is never executed concurrently. In a blocking world you'd typically use `synchronized` or `ReentrantLock` for that. Coroutine's alternative is called Mutex. It has lock and unlock functions to delimit a critical section. The key difference is that `Mutex.lock()` is a suspending function. It does not block a thread.

There is also withLock extension function that conveniently represents `mutex.lock(); try { ... } finally { mutex.unlock() }` pattern:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.sync.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

val mutex = Mutex()
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            // protect each increment with lock
            mutex.withLock {
                counter++
            }
        }
    }
    println("Counter = $counter")
}
```

You can get the full code here.

The locking in this example is fine-grained, so it pays the price. However, it is a good choice for some situations where you absolutely must modify some shared state periodically, but there is no natural thread that this state is confined to.

Select expression (experimental)

Select expression makes it possible to await multiple suspending functions simultaneously and *select* the first one that becomes available.

Select expressions are an experimental feature of `kotlinx.coroutines`. Their API is expected to evolve in the upcoming updates of the `kotlinx.coroutines` library with potentially breaking changes.

Selecting from channels

Let us have two producers of strings: `fizz` and `buzz`. The `fizz` produces "Fizz" string every 500 ms:

```
fun CoroutineScope.fizz() = produce<String> {
    while (true) { // sends "Fizz" every 500 ms
        delay(500)
        send("Fizz")
    }
}
```

And the `buzz` produces "Buzz!" string every 1000 ms:

```
fun CoroutineScope.buzz() = produce<String> {
    while (true) { // sends "Buzz!" every 1000 ms
        delay(1000)
        send("Buzz!")
    }
}
```

Using receive suspending function we can receive *either* from one channel or the other. But select expression allows us to receive from *both* simultaneously using its onReceive clauses:

```
suspend fun selectFizzBuzz(fizz: ReceiveChannel<String>, buzz: ReceiveChannel<String>) {
    select<Unit> { // <Unit> means that this select expression does not produce any result
        fizz.onReceive { value ->  // this is the first select clause
            println("fizz -> '$value'")
        }
        buzz.onReceive { value ->  // this is the second select clause
            println("buzz -> '$value'")
        }
    }
}
```

Let us run it all seven times:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.fizz() = produce<String> {
    while (true) { // sends "Fizz" every 500 ms
        delay(500)
        send("Fizz")
    }
}

fun CoroutineScope.buzz() = produce<String> {
    while (true) { // sends "Buzz!" every 1000 ms
        delay(1000)
        send("Buzz!")
    }
}

suspend fun selectFizzBuzz(fizz: ReceiveChannel<String>, buzz: ReceiveChannel<String>) {
    select<Unit> { // <Unit> means that this select expression does not produce any result
        fizz.onReceive { value ->  // this is the first select clause
            println("fizz -> '$value'")
        }
        buzz.onReceive { value ->  // this is the second select clause
            println("buzz -> '$value'")
        }
    }
}

fun main() = runBlocking<Unit> {
    val fizz = fizz()
    val buzz = buzz()
    repeat(7) {
        selectFizzBuzz(fizz, buzz)
    }
    coroutineContext.cancelChildren() // cancel fizz & buzz coroutines
}
```

You can get the full code here.

The result of this code is:

```
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
fizz -> 'Fizz'
```

Selecting on close

The onReceive clause in `select` fails when the channel is closed causing the corresponding `select` to throw an exception. We can use onReceiveCatching clause to perform a specific action when the channel is closed. The following example also shows that `select` is an expression that returns the result of its selected clause:

```
suspend fun selectAorB(a: ReceiveChannel<String>, b: ReceiveChannel<String>): String =
    select<String> {
        a.onReceiveCatching { it →
            val value = it.getOrNull()
            if (value ≠ null) {
                "a → '$value'"
            } else {
                "Channel 'a' is closed"
            }
        }
        b.onReceiveCatching { it →
            val value = it.getOrNull()
            if (value ≠ null) {
                "b → '$value'"
            } else {
                "Channel 'b' is closed"
            }
        }
    }
```

Let's use it with channel `a` that produces "Hello" string four times and channel `b` that produces "World" four times:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

suspend fun selectAorB(a: ReceiveChannel<String>, b: ReceiveChannel<String>): String =
    select<String> {
        a.onReceiveCatching { it →
            val value = it.getOrNull()
            if (value ≠ null) {
                "a → '$value'"
            } else {
                "Channel 'a' is closed"
            }
        }
        b.onReceiveCatching { it →
            val value = it.getOrNull()
            if (value ≠ null) {
                "b → '$value'"
            } else {
                "Channel 'b' is closed"
            }
        }
    }

fun main() = runBlocking<Unit> {
    val a = produce<String> {
        repeat(4) { send("Hello $it") }
    }
    val b = produce<String> {
        repeat(4) { send("World $it") }
    }
    repeat(8) { // print first eight results
        println(selectAorB(a, b))
    }
    coroutineContext.cancelChildren()
}
```

You can get the full code here.

The result of this code is quite interesting, so we'll analyze it in more detail:

```
a → 'Hello 0'
a → 'Hello 1'
b → 'World 0'
a → 'Hello 2'
a → 'Hello 3'
b → 'World 1'
Channel 'a' is closed
Channel 'a' is closed
```

There are a couple of observations to make out of it.

First of all, `select` is *biased* to the first clause. When several clauses are selectable at the same time, the first one among them gets selected. Here, both channels are constantly producing strings, so `a` channel, being the first clause in select, wins. However, because we are using unbuffered channel, the `a` gets suspended from time to time on its send invocation and gives a chance for `b` to send, too.

The second observation, is that onReceiveCatching gets immediately selected when the channel is already closed.

Selecting to send

Select expression has onSend clause that can be used for a great good in combination with a biased nature of selection.

Let us write an example of a producer of integers that sends its values to a `side` channel when the consumers on its primary channel cannot keep up with it:

```
fun CoroutineScope.produceNumbers(side: SendChannel<Int>) = produce<Int> {
    for (num in 1..10) { // produce 10 numbers from 1 to 10
        delay(100) // every 100 ms
        select<Unit> {
            onSend(num) {} // Send to the primary channel
            side.onSend(num) {} // or to the side channel
        }
    }
}
```

Consumer is going to be quite slow, taking 250 ms to process each number:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.produceNumbers(side: SendChannel<Int>) = produce<Int> {
    for (num in 1..10) { // produce 10 numbers from 1 to 10
        delay(100) // every 100 ms
        select<Unit> {
            onSend(num) {} // Send to the primary channel
            side.onSend(num) {} // or to the side channel
        }
    }
}

fun main() = runBlocking<Unit> {
    val side = Channel<Int>() // allocate side channel
    launch { // this is a very fast consumer for the side channel
        side.consumeEach { println("Side channel has $it") }
    }
    produceNumbers(side).consumeEach {
        println("Consuming $it")
        delay(250) // let us digest the consumed number properly, do not hurry
    }
    println("Done consuming")
    coroutineContext.cancelChildren()
}
```

You can get the full code here.

So let us see what happens:

```
Consuming 1
Side channel has 2
Side channel has 3
Consuming 4
Side channel has 5
Side channel has 6
Consuming 7
Side channel has 8
Side channel has 9
Consuming 10
Done consuming
```

Selecting deferred values

Deferred values can be selected using onAwait clause. Let us start with an async function that returns a deferred string value after a random delay:

```
fun CoroutineScope.asyncString(time: Int) = async {
    delay(time.toLong())
    "Waited for $time ms"
}
```

Let us start a dozen of them with a random delay.

```
fun CoroutineScope.asyncStringsList(): List<Deferred<String>> {
    val random = Random(3)
    return List(12) { asyncString(random.nextInt(1000)) }
}
```

Now the main function awaits for the first of them to complete and counts the number of deferred values that are still active. Note that we've used here the fact that `select` expression is a Kotlin DSL, so we can provide clauses for it using an arbitrary code. In this case we iterate over a list of deferred values to provide `onAwait` clause for each deferred value.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.selects.*
import java.util.*

fun CoroutineScope.asyncString(time: Int) = async {
    delay(time.toLong())
    "Waited for $time ms"
}

fun CoroutineScope.asyncStringsList(): List<Deferred<String>> {
    val random = Random(3)
    return List(12) { asyncString(random.nextInt(1000)) }
}

fun main() = runBlocking<Unit> {
    val list = asyncStringsList()
    val result = select<String> {
        list.withIndex().forEach { (index, deferred) ->
            deferred.onAwait { answer ->
                "Deferred $index produced answer '$answer'"
            }
        }
    }
    println(result)
    val countActive = list.count { it.isActive }
    println("$countActive coroutines are still active")
}
```


You can get the full code here.

The output is:

```
Deferred 4 produced answer 'Waited for 128 ms'
11 coroutines are still active
```

Switch over a channel of deferred values

Let us write a channel producer function that consumes a channel of deferred string values, waits for each received deferred value, but only until the next deferred value comes over or the channel is closed. This example puts together onReceiveCatching and onAwait clauses in the same `select` :

```
fun CoroutineScope.switchMapDeferreds(input: ReceiveChannel<Deferred<String>>) = produce<String> {
    var current = input.receive() // start with first received deferred value
    while (isActive) { // loop while not cancelled/closed
        val next = select<Deferred<String>?> { // return next deferred value from this select or null
            input.onReceiveCatching { update ->
                update.getOrNull()
            }
            current.onAwait { value ->
                send(value) // send value that current deferred has produced
                input.receiveCatching().getOrNull() // and use the next deferred from the input
channel
            }
        }
        if (next == null) {
            println("Channel was closed")
            break // out of loop
        } else {
            current = next
        }
    }
}
```

To test it, we'll use a simple async function that resolves to a specified string after a specified time:

```
fun CoroutineScope.asyncString(str: String, time: Long) = async {
    delay(time)
    str
}
```

The main function just launches a coroutine to print results of `switchMapDeferreds` and sends some test data to it:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.switchMapDeferreds(input: ReceiveChannel<Deferred<String>>) = produce<String> {
    var current = input.receive() // start with first received deferred value
    while (isActive) { // loop while not cancelled/closed
        val next = select<Deferred<String>?> { // return next deferred value from this select or null
            input.onReceiveCatching { update ->
                update.getOrNull()
            }
            current.onAwait { value ->
                send(value) // send value that current deferred has produced
                input.receiveCatching().getOrNull() // and use the next deferred from the input
channel
            }
        }
        if (next == null) {
            println("Channel was closed")
            break // out of loop
        } else {
            current = next
        }
    }
}

fun CoroutineScope.asyncString(str: String, time: Long) = async {
    delay(time)
    str
}

fun main() = runBlocking<Unit> {
    val chan = Channel<Deferred<String>>() // the channel for test
    launch { // launch printing coroutine
        for (s in switchMapDeferreds(chan))
            println(s) // print each received string
    }
    chan.send(asyncString("BEGIN", 100))
    delay(200) // enough time for "BEGIN" to be produced
    chan.send(asyncString("Slow", 500))
    delay(100) // not enough time to produce slow
    chan.send(asyncString("Replace", 100))
    delay(500) // give it time before the last one
    chan.send(asyncString("END", 500))
    delay(1000) // give it time to process
    chan.close() // close the channel ...
    delay(500) // and wait some time to let it finish
}
```



You can get the full code here.

The result of this code:

```
BEGIN
Replace
END
Channel was closed
```

Debug coroutines

Debugging applications that use coroutines can be challenging because multiple coroutines can run concurrently, suspend on one thread, and resume on another. Their execution order and the threads they use can also change between runs, making it difficult to follow the execution of a particular coroutine.

On the JVM, you can use the following features to make debugging coroutines easier:

- Debug mode adds a unique name to each coroutine so you can identify it in a debugger and diagnostic output.
- Stack trace recovery adds information about where a coroutine receives an exception instead of an expected result.
- The debug agent tracks active coroutines, reports their state, and more.

Debug mode and stack trace recovery are available in the `kotlinx-coroutines-core` module. The debug agent is available in the `kotlinx-coroutines-debug` module.

> The debug agent isn't supported on Android.

Enable debug mode

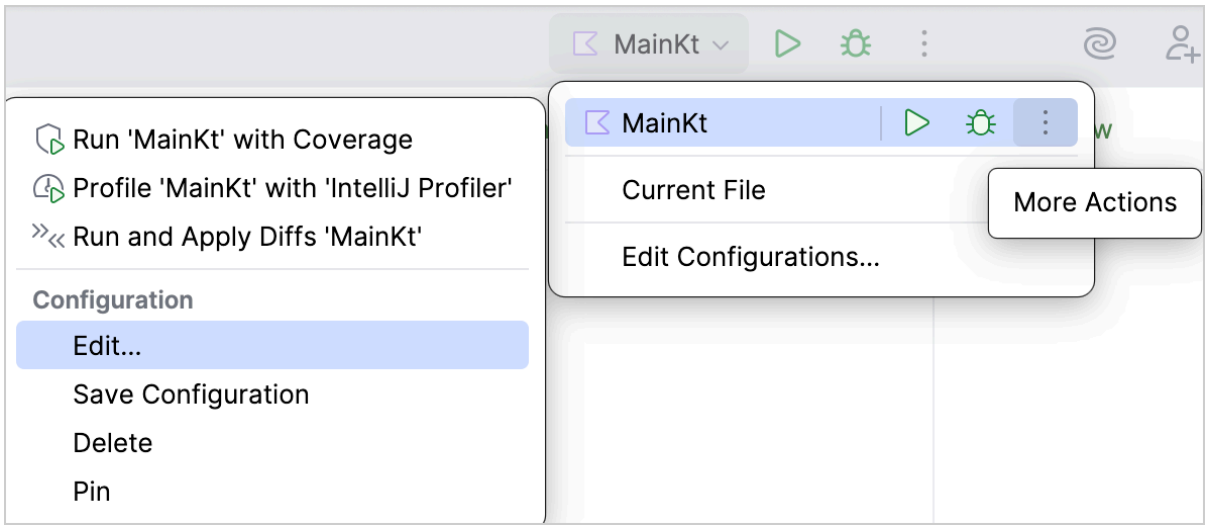
Debug mode assigns a unique name to every launched coroutine. You can see the coroutine names in a Java debugger, in the coroutine's string representation, and in a thread's name while it runs the coroutine. Debug mode has negligible runtime overhead, so you can keep it enabled to simplify logging and diagnostics.

When you run your code with Java assertions enabled, the `kotlinx.coroutines` library automatically enables debug mode. Unit tests run with assertions enabled by default, so you don't need to enable debug mode explicitly for them.

To enable debug mode explicitly, configure your build tool, such as Gradle or Maven, or IDE run configuration to pass the `-Dkotlinx.coroutines.debug` argument to the JVM that runs your application.

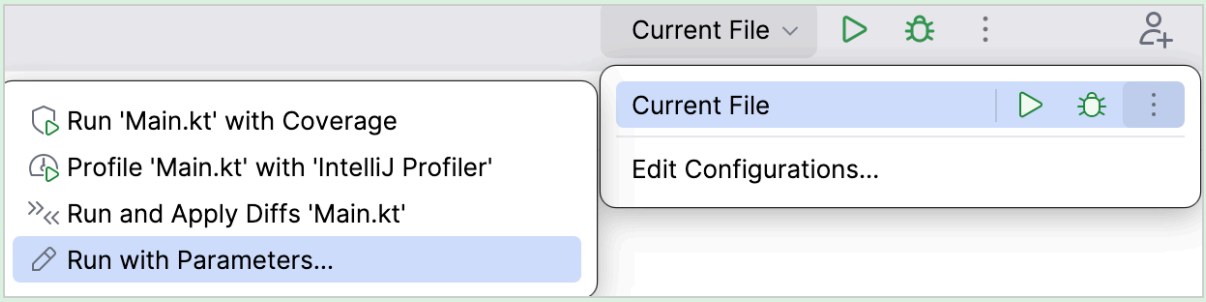
In IntelliJ IDEA, follow these steps to enable debug mode:

1. In the **Run widget**, select the run/debug configuration you want to update, then select **More Actions** | **Edit**:



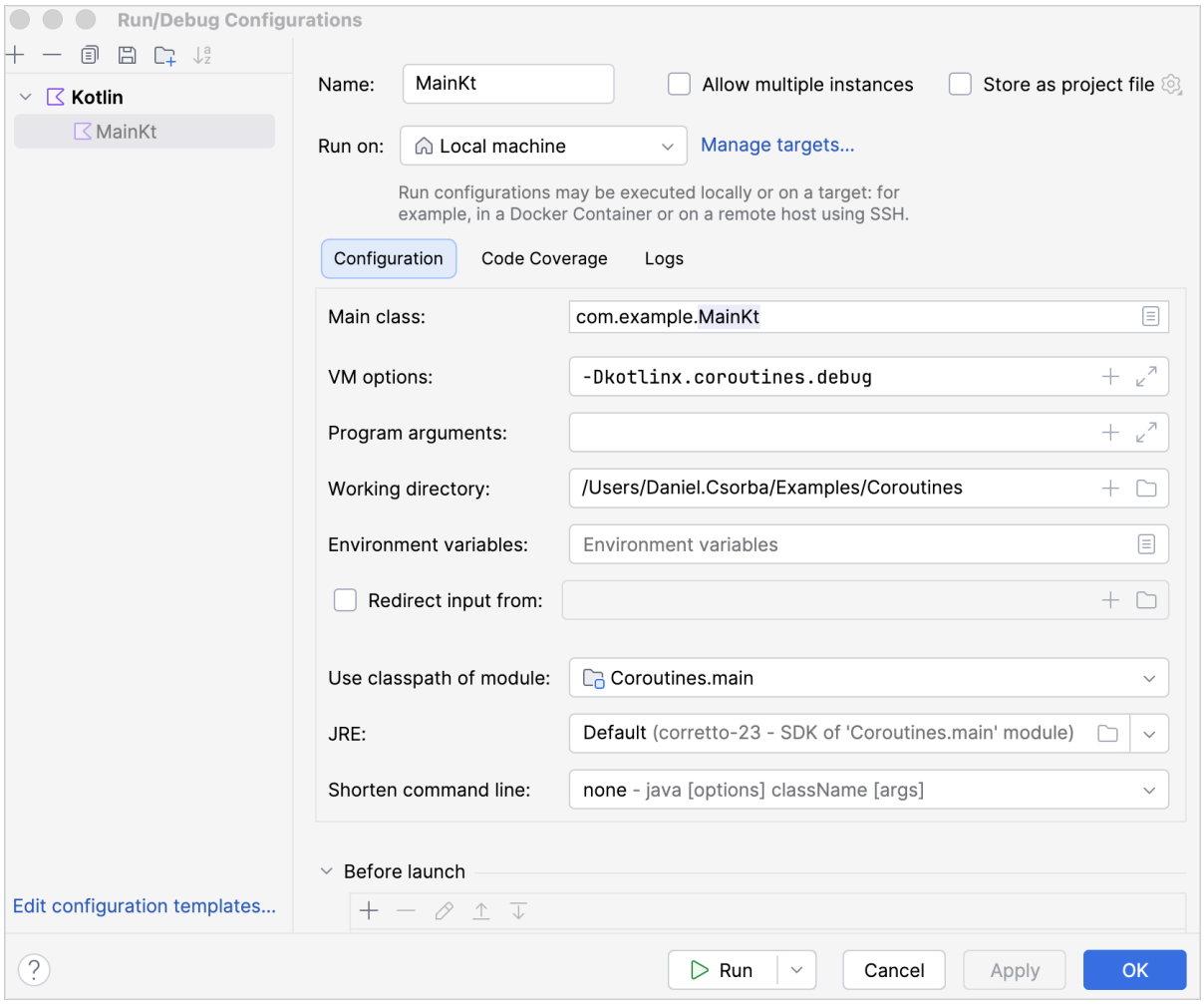
Selecting Edit from the More Actions menu for a run configuration in IntelliJ IDEA

If you don't have a run/debug configuration, select **Current File** in the **Run widget**, then select **More Actions** | **Run with Parameters** to open the run configuration settings.



Selecting Run with Parameters from the More Actions menu for a run configuration in IntelliJ IDEA

2. In the **Run/Debug Configurations** dialog, enter `-Dkotlinx.coroutines.debug` in the **VM options** field, and click **OK**:



Adding the -Dkotlinx.coroutines.debug option to a run/debug configuration in IntelliJ IDEA

Stack trace recovery

When a coroutine receives an exception from another coroutine through a suspending function such as `Deferred.await()`, the exception's stack trace doesn't contain the stack frames from the receiving coroutine.

Without these stack frames, the stack trace doesn't show where `Deferred.await()` is called or which functions lead to that call, which can make debugging difficult.

The `kotlinx.coroutines` library adds this information using *stack trace recovery*, which creates a copy of the exception with additional stack frames.

When the receiving coroutine resumes, it throws the copy instead of the original exception. The original exception becomes the cause of the copy. If the original exception has suppressed exceptions, they remain attached to it instead of being copied. Keeping them attached to the original exception can prevent cycles in the exception chain and crashes in some frameworks.

Debug mode enables stack trace recovery by default. To disable stack trace recovery in debug mode, pass the `-Dkotlinx.coroutines.stacktrace.recovery=false` VM option.

Here's an example to demonstrate the difference between stack traces with and without stack trace recovery:

```
import kotlinx.coroutines.*

object UserProfileService :
    CoroutineScope by CoroutineScope(CoroutineName("UserProfileService")) {

    private fun parseUserProfile(): String {
        error("Invalid user profile")
    }

    private fun loadUserProfile(): String {
        return parseUserProfile()
    }

    // Runs in the coroutine that calls this function
    suspend fun awaitUserProfile() {
        // Starts a new coroutine
        val userProfile = async(Dispatchers.Default) {
            // The new coroutine throws the exception
            loadUserProfile()
        }

        // The coroutine running awaitUserProfile()
        // receives the exception through the await() function
        userProfile.await()
    }
}

suspend fun main() {
    UserProfileService.awaitUserProfile()
}
```

In this example, the `parseUserProfile()` function throws the exception in the coroutine started by the `.async()` builder function. The coroutine that calls `awaitUserProfile()` receives the exception through the `Deferred.await()` function.

With stack trace recovery disabled, the stack trace shows where `parseUserProfile()` throws the exception in the coroutine created by the `.async()` function, but it doesn't include the `Deferred.await()` call in the `awaitUserProfile()` function:

```
Exception in thread "main" java.lang.IllegalStateException Create breakpoint : Invalid user profile
    at com.example.UserProfileService.parseUserProfile(RecoveryExample.kt:9)
    at com.example.UserProfileService.loadUserProfile(RecoveryExample.kt:13)
    at com.example.UserProfileService.access$loadUserProfile(RecoveryExample.kt:5)
    at com.example.UserProfileService$awaitUserProfile$userProfile$1.invokeSuspend(RecoveryExample.kt:18)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:34)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:100)
    at kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(CoroutineScheduler.kt:586)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTask(CoroutineScheduler.kt:807)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(CoroutineScheduler.kt:717)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(CoroutineScheduler.kt:704)
```

Exception stack trace without stack frames from the receiving coroutine

With stack trace recovery enabled, the stack trace also includes the `Deferred.await()` call in the `awaitUserProfile()` function:

```
Exception in thread "main" java.lang.IllegalStateException Create breakpoint : Invalid user profile
    at com.example.UserProfileService.parseUserProfile(RecoveryExample.kt:9)
    at com.example.UserProfileService.loadUserProfile(RecoveryExample.kt:13)
    at com.example.UserProfileService.access$loadUserProfile(RecoveryExample.kt:5)
    at com.example.UserProfileService$awaitUserProfile$userProfile$1.invokeSuspend(RecoveryExample.kt:18)
    at _COROUTINE._BOUNDARY._(CoroutineDebugging.kt:42)
    at com.example.UserProfileService.awaitUserProfile(RecoveryExample.kt:21)
Caused by: java.lang.IllegalStateException Create breakpoint : Invalid user profile
    at com.example.UserProfileService.parseUserProfile(RecoveryExample.kt:9)
    at com.example.UserProfileService.loadUserProfile(RecoveryExample.kt:13)
    at com.example.UserProfileService.access$loadUserProfile(RecoveryExample.kt:5)
    at com.example.UserProfileService$awaitUserProfile$userProfile$1.invokeSuspend(RecoveryExample.kt:18)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:34)
    at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:100)
    at kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(CoroutineScheduler.kt:586)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTask(CoroutineScheduler.kt:807)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(CoroutineScheduler.kt:717)
    at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(CoroutineScheduler.kt:704)
```

Recovered exception stack trace with stack frames from the receiving coroutine

Experimental

Stack trace recovery for custom exceptions

Stack trace recovery can copy an exception automatically when its class has a public constructor that accepts a message, a cause, both, or no arguments at all.

If you want the `kotlinx.coroutines` library to recover the stack trace of an exception that requires additional constructor arguments, such as a line number or an error code, implement the `StackTraceRecoverable` interface.

The `StackTraceRecoverable` interface is part of the Kotlin standard library, so you can implement it without adding a dependency on the `kotlinx.coroutines` library.

> The `StackTraceRecoverable` interface is available on all targets, but the `kotlinx.coroutines` library uses it for stack trace recovery only on the JVM.

To implement the interface, override the `copyForStackTraceRecovery()` function. In the override, return a new exception instance for stack trace recovery, or `null` if you don't want the `kotlinx.coroutines` library to copy the exception.

These APIs are Experimental and require opt-in with the `@OptIn(ExperimentalStdlibCoroutineSupportApi::class)` annotation.

Here's an example of a custom exception that preserves a `line` property when it creates a new instance for stack trace recovery:

```
import kotlinx.coroutines.*
import kotlin.coroutines.ExperimentalStdlibCoroutineSupportApi
import kotlin.coroutines.debug.StackTraceRecoverable

@OptIn(ExperimentalStdlibCoroutineSupportApi::class)
class FileEditException
// The implementation requires a private constructor
// to pass the cause to the IllegalStateException constructor
private constructor(
    val line: Int,
    private val detail: String,
    cause: Throwable?,
) : IllegalStateException("When editing line $line: $detail", cause),
    // Implements StackTraceRecoverable for stack trace recovery
    StackTraceRecoverable<FileEditException> {

    constructor(line: Int, detail: String) : this(line, detail, null)

    // Copies the line number and message details
    override fun copyForStackTraceRecovery(): FileEditException =
        FileEditException(line, detail, this)
}

private fun editFile() {
    throw FileEditException(15, "Unexpected token")
}

suspend fun main() {
    supervisorScope {
        // Starts a new coroutine
        val fileEdit = async(Dispatchers.Default) {
            // Throws the original exception
            editFile()
        }

        // Stack trace recovery creates a copy of the exception,
        // adds the calling coroutine's stack frames, and throws the copy
        fileEdit.await()
    }
}
```

With debug mode enabled, the output contains the recovered copy followed by the original exception as its cause.

```
Exception in thread "main" com.example.FileEditException: When editing line 15: Unexpected token
        at com.example.RecoveryExampleKt.editFile(RecoveryExample.kt:54)
        at com.example.RecoveryExampleKt.access$editFile(RecoveryExample.kt:1)
        at com.example.RecoveryExampleKt$main$2$fileEdit$1.invokeSuspend(RecoveryExample.kt:62)
        at _COROUTINE._BOUNDARY._(CoroutineDebugging.kt:42)
        at com.example.RecoveryExampleKt$main$2.invokeSuspend(RecoveryExample.kt:67)
Caused by: com.example.FileEditException: When editing line 15: Unexpected token
        at com.example.RecoveryExampleKt.editFile(RecoveryExample.kt:54)
        at com.example.RecoveryExampleKt.access$editFile(RecoveryExample.kt:1)
        at com.example.RecoveryExampleKt$main$2$fileEdit$1.invokeSuspend(RecoveryExample.kt:62)
        at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:34)
        at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:100)
        at kotlinx.coroutines.scheduling.CoroutineScheduler.runSafely(CoroutineScheduler.kt:586)
        at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.executeTask(CoroutineScheduler.kt:807)
        at
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.runWorker(CoroutineScheduler.kt:717)
        at kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.run(CoroutineScheduler.kt:704)
```

Experimental

The debug agent

The `kotlinx-coroutines-debug` module provides a debug agent for JVM applications. The agent tracks coroutines as they are created, suspended, and resumed.

The `DebugProbes` API is the main entry point for the debug agent. You can use it to print active coroutines and their current state. The output includes stack traces that show where each coroutine was created and where it is suspended. You can also use it to print a coroutine dump for the hierarchy of a specific `Job` or `CoroutineScope`.

If you enable `DebugProbes` in a production environment, it can significantly reduce your application's performance when it creates a stack trace for each new coroutine. To avoid this overhead, set `DebugProbes.enableCreationStackTraces` to `false`.

> The `kotlinx-coroutines-debug` module provides automatic BlockHound integration. You can use it to detect blocking operations in coroutine contexts where they aren't allowed.
>
> For setup instructions, see the BlockHound quick start guide.

Add the debug agent dependency

To use the debug agent in your project, add the `kotlinx-coroutines-debug` dependency:

Gradle

```
// build.gradle.kts
dependencies {
    testImplementation("org.jetbrains.kotlinx:kotlinx-coroutines-debug:1.11.0")
}
```

Maven

```
<!-- pom.xml -->
<dependency>
    <groupId>org.jetbrains.kotlinx</groupId>
    <artifactId>kotlinx-coroutines-debug</artifactId>
    <version>1.11.0</version>
    <scope>test</scope>
</dependency>
```

Track coroutines with the debug agent

To start tracking coroutines with the debug agent, you can either:

- Add `-javaagent:/path/to/kotlinx-coroutines-debug-1.11.0.jar` to your VM options to load the debug agent when the application starts.
- Call the `DebugProbes.install()` function before starting the coroutines you want to track.

Starting with JDK 21, dynamically loading the debug agent with the `DebugProbes.install()` function can produce a warning. To avoid this warning, load the agent with the `-javaagent` VM option.

With the debug agent active, you can use the following APIs:

- `DebugProbes.dumpCoroutines()` prints all active coroutines.
- `DebugProbes.dumpCoroutinesInfo()` returns information about active coroutines.
- `DebugProbes.printJob()` prints a coroutine dump for the hierarchy of a `Job`.
- `DebugProbes.printScope()` prints a coroutine dump for the hierarchy of a `CoroutineScope`.

Here's an example that uses the debug agent to print active coroutines and the coroutine hierarchy for a specific `Job`:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.debug.*
import kotlin.time.Duration.Companion.seconds

private suspend fun loadAccount() {
    delay(5.seconds)
}

private suspend fun loadPreferences() {
    delay(5.seconds)
}

private suspend fun loadUserProfile() = coroutineScope {
    launch { loadAccount() }
    launch { loadPreferences() }
}

@OptIn(ExperimentalCoroutinesApi::class)
fun main() {
    // Installs the debug agent
    // This is only required if you don't use the -javaagent VM option
    DebugProbes.install()

    runBlocking {
        // Starts a coroutine with two child coroutines
        val loadingJob = launch {
            loadUserProfile()
        }

        // Gives the child coroutines time to suspend
        delay(1.seconds)

        // Prints all active coroutines
        DebugProbes.dumpCoroutines()

        println("=============")

        // Prints the loading job and its child coroutines
        DebugProbes.printJob(loadingJob)
    }
}
```

With debug mode enabled, running the example produces the following output:

```
Coroutines dump 2026/08/18 14:00:08

Coroutine "coroutine#1":BlockingCoroutine{Active}@146ba0ac, state: RUNNING
        at java.base/java.lang.Thread.getStackTrace(Thread.java:2389)
        at
kotlinx.coroutines.debug.internal.DebugProbesImpl.enhanceStackTraceWithThreadDumpImpl(DebugProbesImpl.kt
        at
kotlinx.coroutines.debug.internal.DebugProbesImpl.dumpCoroutinesSynchronized(DebugProbesImpl.kt:294)
        at kotlinx.coroutines.debug.internal.DebugProbesImpl.dumpCoroutines(DebugProbesImpl.kt:266)
        at kotlinx.coroutines.debug.DebugProbes.dumpCoroutines(DebugProbes.kt:181)
        at kotlinx.coroutines.debug.DebugProbes.dumpCoroutines$default(DebugProbes.kt:181)
        at DebugAgentExampleKt$main$1.invokeSuspend(DebugAgentExample.kt:34)

Coroutine "coroutine#2":StandaloneCoroutine{Active}@4dfa3a9d, state: SUSPENDED
        at DebugAgentExampleKt$main$1$loadingJob$1.invokeSuspend(DebugAgentExample.kt:27)

Coroutine "coroutine#3":StandaloneCoroutine{Active}@6eebc39e, state: SUSPENDED
        at DebugAgentExampleKt$loadUserProfile$2$1.invokeSuspend(DebugAgentExample.kt:14)

Coroutine "coroutine#4":StandaloneCoroutine{Active}@464bee09, state: SUSPENDED
        at DebugAgentExampleKt$loadUserProfile$2$2.invokeSuspend(DebugAgentExample.kt:15)============
"coroutine#2":StandaloneCoroutine{Active}, continuation is SUSPENDED at line
DebugAgentExampleKt$main$1$loadingJob$1.invokeSuspend(DebugAgentExample.kt:27)
        "coroutine#3":StandaloneCoroutine{Active}, continuation is SUSPENDED at line
DebugAgentExampleKt$loadUserProfile$2$1.invokeSuspend(DebugAgentExample.kt:14)
        "coroutine#4":StandaloneCoroutine{Active}, continuation is SUSPENDED at line
DebugAgentExampleKt$loadUserProfile$2$2.invokeSuspend(DebugAgentExample.kt:15)
```

Print active coroutines when JUnit tests time out

You can set a timeout for JUnit tests with the corresponding `CoroutinesTimeout` API, depending on the JUnit version. The API installs debug probes automatically. If a test doesn't complete before the timeout, it prints all active coroutines and their stack traces and fails the test.

JUnit 4

To set a timeout for JUnit 4 tests and print all active coroutines and their stack traces if they exceed it, use the `CoroutinesTimeout` rule:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.debug.junit4.CoroutinesTimeout
import org.junit.Rule
import org.junit.Test
import kotlin.time.Duration

@OptIn(ExperimentalCoroutinesApi::class)
class UserProfileTest {
    @get:Rule
    val timeout = CoroutinesTimeout.seconds(1)

    private suspend fun loadUserProfile() {
        withContext(Dispatchers.IO) {
            // Simulates an operation that doesn't complete
            delay(Duration.INFINITE)
        }
    }

    @Test
    fun loadsUserProfile() = runBlocking {
        val loadingJob = launch {
            loadUserProfile()
        }

        // Waits for the coroutine, so the test doesn't complete
        loadingJob.join()
    }
}
```

After one second, the rule reports that the test timed out and prints all active coroutines and their stack traces. The test then fails with a `TestTimedOutException`.

```
Test loadsUserProfile timed out after 1 seconds

Coroutines dump 2026/08/18 13:48:21

Coroutine "coroutine#1":BlockingCoroutine{Active}@bf1ec20, state: SUSPENDED
        at UserProfileTest$loadsUserProfile$1.invokeSuspend(UserProfileTest.kt:27)
        at _COROUTINE._CREATION._(CoroutineDebugging.kt:30)
        at
kotlin.coroutines.intrinsics.IntrinsicsKt__IntrinsicsJvmKt.createCoroutineUnintercepted(IntrinsicsJvm.kt
        at kotlinx.coroutines.intrinsics.CancellableKt.startCoroutineCancellable(Cancellable.kt:26)
        at
kotlinx.coroutines.BuildersKt__Builders_concurrentKt.runBlockingK$default(Builders.concurrent.kt:157)
        at kotlinx.coroutines.BuildersKt.runBlockingK$default(Unknown Source)
        at UserProfileTest.loadsUserProfile(UserProfileTest.kt:21)
        at
java.base/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(DirectMethodHandleAccessor.java:103)
        at java.base/java.lang.reflect.Method.invoke(Method.java:580)
        at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:59)
        at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
        at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:56)
        at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
        at
kotlinx.coroutines.debug.junit4.CoroutinesTimeoutStatement$evaluate$$inlined$runWithTimeoutDumpingCorout
        at
kotlinx.coroutines.debug.junit4.CoroutinesTimeoutStatement$evaluate$$inlined$runWithTimeoutDumpingCorout
        at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:317)
        at java.base/java.lang.Thread.run(Thread.java:1575)

Coroutine "coroutine#2":StandaloneCoroutine{Active}@70efb718, state: SUSPENDED
        at UserProfileTest$loadUserProfile$2.invokeSuspend(UserProfileTest.kt:16)
        at UserProfileTest$loadsUserProfile$1$loadingJob$1.invokeSuspend(UserProfileTest.kt:23)
        at _COROUTINE._CREATION._(CoroutineDebugging.kt:30)
        at
kotlin.coroutines.intrinsics.IntrinsicsKt__IntrinsicsJvmKt.createCoroutineUnintercepted(IntrinsicsJvm.kt
        at kotlinx.coroutines.intrinsics.CancellableKt.startCoroutineCancellable(Cancellable.kt:26)
        at kotlinx.coroutines.BuildersKt__Builders_commonKt.launch$default(Builders.common.kt:200)
        at kotlinx.coroutines.BuildersKt.launch$default(Unknown Source)
        at UserProfileTest$loadsUserProfile$1.invokeSuspend(UserProfileTest.kt:22)
        at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:34)
        at kotlinx.coroutines.DispatchedTask.run(DispatchedTask.kt:100)
        at
kotlinx.coroutines.BuildersKt__Builders_concurrentKt.runBlockingK$default(Builders.concurrent.kt:157)
        at kotlinx.coroutines.BuildersKt.runBlockingK$default(Unknown Source)
        at UserProfileTest.loadsUserProfile(UserProfileTest.kt:21)
        at
java.base/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(DirectMethodHandleAccessor.java:103)
        at java.base/java.lang.reflect.Method.invoke(Method.java:580)
        at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:59)
        at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
        at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:56)
        at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
        at
kotlinx.coroutines.debug.junit4.CoroutinesTimeoutStatement$evaluate$$inlined$runWithTimeoutDumpingCorout
        at
kotlinx.coroutines.debug.junit4.CoroutinesTimeoutStatement$evaluate$$inlined$runWithTimeoutDumpingCorout
        at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:317)
        at java.base/java.lang.Thread.run(Thread.java:1575)
test timed out after 1000 milliseconds
org.junit.runners.model.TestTimedOutException: test timed out after 1000 milliseconds
        at java.base/jdk.internal.misc.Unsafe.park(Native Method)
        at java.base/java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:269)
        at kotlinx.coroutines.BlockingCoroutine.joinBlocking(Builders.kt:57)
        at kotlinx.coroutines.BuildersKt__BuildersKt.runBlockingImpl(Builders.kt:30)
```

```
        at kotlinx.coroutines.BuildersKt.runBlockingImpl(Unknown Source)
        at
kotlinx.coroutines.BuildersKt__Builders_concurrentKt.runBlockingK(Builders.concurrent.kt:172)
        at kotlinx.coroutines.BuildersKt.runBlockingK(Unknown Source)
        at
kotlinx.coroutines.BuildersKt__Builders_concurrentKt.runBlockingK$default(Builders.concurrent.kt:157)
        at kotlinx.coroutines.BuildersKt.runBlockingK$default(Unknown Source)
        at UserProfileTest.loadsUserProfile(UserProfileTest.kt:21)
        at
java.base/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(DirectMethodHandleAccessor.java:103)
        at java.base/java.lang.reflect.Method.invoke(Method.java:580)
        at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:59)
        at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
        at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:56)
        at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
        at
kotlinx.coroutines.debug.junit4.CoroutinesTimeoutStatement$evaluate$$inlined$runWithTimeoutDumpingCorout
        at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:317)
        at java.base/java.lang.Thread.run(Thread.java:1575)
```

JUnit 5

To apply a timeout to all test functions in a class, add the `@CoroutinesTimeout` annotation to the class:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.debug.junit5.CoroutinesTimeout
import org.junit.jupiter.api.Test
import kotlin.time.Duration

@OptIn(ExperimentalCoroutinesApi::class)
// Sets a one-second timeout for all test functions in the class
@CoroutinesTimeout(testTimeoutMs = 1_000)
class UserProfileTest {
    private suspend fun loadUserProfile() {
        withContext(Dispatchers.IO) {
            // Simulates an operation that doesn't complete
            delay(Duration.INFINITE)
        }
    }

    @Test
    fun loadsUserProfile() = runBlocking {
        val loadingJob = launch {
            loadUserProfile()
        }

        // Waits for the coroutine, so the test doesn't complete
        loadingJob.join()
    }
}
```

After one second, the `CoroutinesTimeout` API reports the timeout and prints all active coroutines and their stack traces. The test then fails with a `CoroutinesTimeoutException`.

```
Test loadsUserProfile timed out after 1 seconds

Coroutines dump 2026/08/18 13:46:15

Coroutine "coroutine#1":BlockingCoroutine{Active}@5c77053b, state: SUSPENDED
        at UserProfileTest$loadsUserProfile$1.invokeSuspend(UserProfileTest.kt:24)

Coroutine "coroutine#2":StandaloneCoroutine{Active}@26b894bd, state: SUSPENDED
        at UserProfileTest$loadUserProfile$2.invokeSuspend(UserProfileTest.kt:13)
        at UserProfileTest$loadsUserProfile$1$loadingJob$1.invokeSuspend(UserProfileTest.kt:20)
test timed out after 1000 ms
kotlinx.coroutines.debug.junit5.CoroutinesTimeoutException: test timed out after 1000 ms
        at java.base/jdk.internal.misc.Unsafe.park(Native Method)
        at java.base/java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:269)
        at kotlinx.coroutines.BlockingCoroutine.joinBlocking(Builders.kt:57)
        at kotlinx.coroutines.BuildersKt__BuildersKt.runBlockingImpl(Builders.kt:30)
        at kotlinx.coroutines.BuildersKt.runBlockingImpl(Unknown Source)
        at
kotlinx.coroutines.BuildersKt__Builders_concurrentKt.runBlockingK(Builders.concurrent.kt:172)
        at kotlinx.coroutines.BuildersKt.runBlockingK(Unknown Source)
        at
kotlinx.coroutines.BuildersKt__Builders_concurrentKt.runBlockingK$default(Builders.concurrent.kt:157)
        at kotlinx.coroutines.BuildersKt.runBlockingK$default(Unknown Source)
        at UserProfileTest.loadsUserProfile(UserProfileTest.kt:18)
        at java.base/java.lang.reflect.Method.invoke(Method.java:580)
        at
kotlinx.coroutines.debug.junit5.CoroutinesTimeoutExtension$interceptInvocation$$inlined$runWithTimeoutDu
        at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:317)
        at java.base/java.lang.Thread.run(Thread.java:1575)
```

Resolve `kotlinx-coroutines-debug` resource conflicts on Android

The debug agent isn't supported on Android.

The `kotlinx-coroutines-debug` module has transitive dependencies on JNA, JNA Platform, Byte Buddy, and Byte Buddy Agent. Some of these dependencies contain resources with the same paths. When Android merges dependency resources, the duplicate paths can cause a `DuplicateRelativeFileException` resulting in a build failure.

To resolve the build failure while keeping the `kotlinx-coroutines-debug` dependency, exclude the conflicting resources with the following `packaging` configuration in your `build.gradle.kts` file:

```
// build.gradle.kts
android {
    packaging {
        resources {
            // Excludes license files from JNA and JNA Platform
            excludes += setOf(
                "META-INF/AL2.0",
                "META-INF/LGPL2.1",
            )

            // Excludes the ASM license file from Byte Buddy
            excludes += "META-INF/licenses/ASM"

            // Retains one copy of each Byte Buddy Agent file
            pickFirsts += setOf(
                "win32-x86-64/attach_hotspot_windows.dll",
                "win32-x86/attach_hotspot_windows.dll",
            )
        }
    }
}
```

What's next

Learn how to debug coroutines in IntelliJ IDEA in Debug coroutines using IntelliJ IDEA and Debug Kotlin Flow using IntelliJ IDEA.

Debug coroutines using IntelliJ IDEA – tutorial

This tutorial demonstrates how to create Kotlin coroutines and debug them using IntelliJ IDEA.

The tutorial assumes you have prior knowledge of the coroutines concept.

Create coroutines

1. Open a Kotlin project in IntelliJ IDEA. If you don't have a project, create one.
2. To use the `kotlinx.coroutines` library in a Gradle project, add the following dependency to `build.gradle(.kts)`:

Kotlin

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0")
}
```

Groovy

```
dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0'
}
```

For other build systems, see instructions in the `kotlinx.coroutines` README.

3. Open the `Main.kt` file in `src/main/kotlin`.

   The `src` directory contains Kotlin source files and resources. The `Main.kt` file contains sample code that will print `Hello World!`.

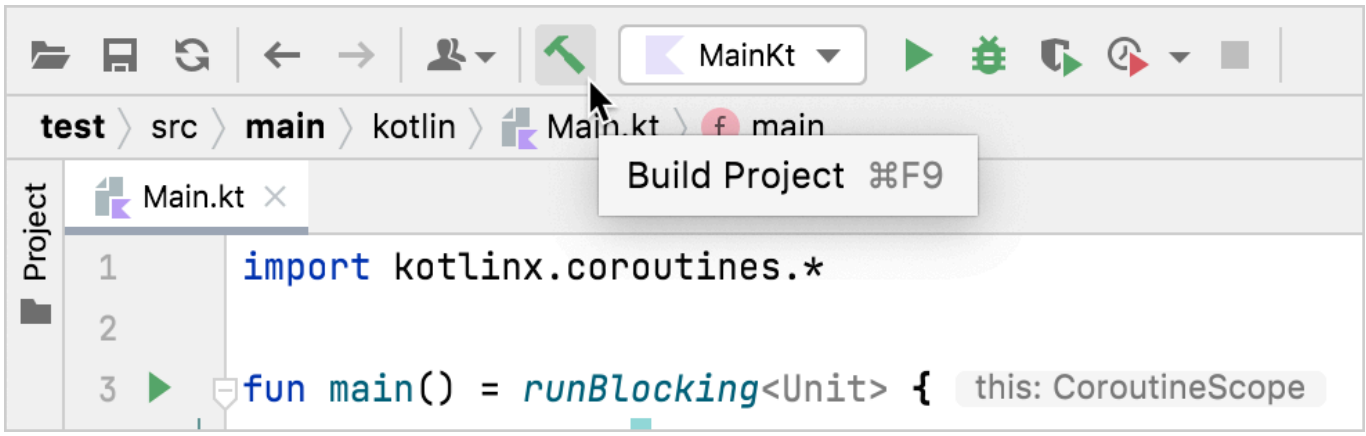
4. Change code in the `main()` function:

   - Use the `runBlocking()` block to wrap a coroutine.
   - Use the `async()` function to create coroutines that compute deferred values `a` and `b`.
   - Use the `await()` function to await the computation result.
   - Use the `println()` function to print computing status and the result of multiplication to the output.

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    val a = async {
        println("I'm computing part of the answer")
        6
    }
    val b = async {
        println("I'm computing another part of the answer")
        7
    }
    println("The answer is ${a.await() * b.await()}")
}
```

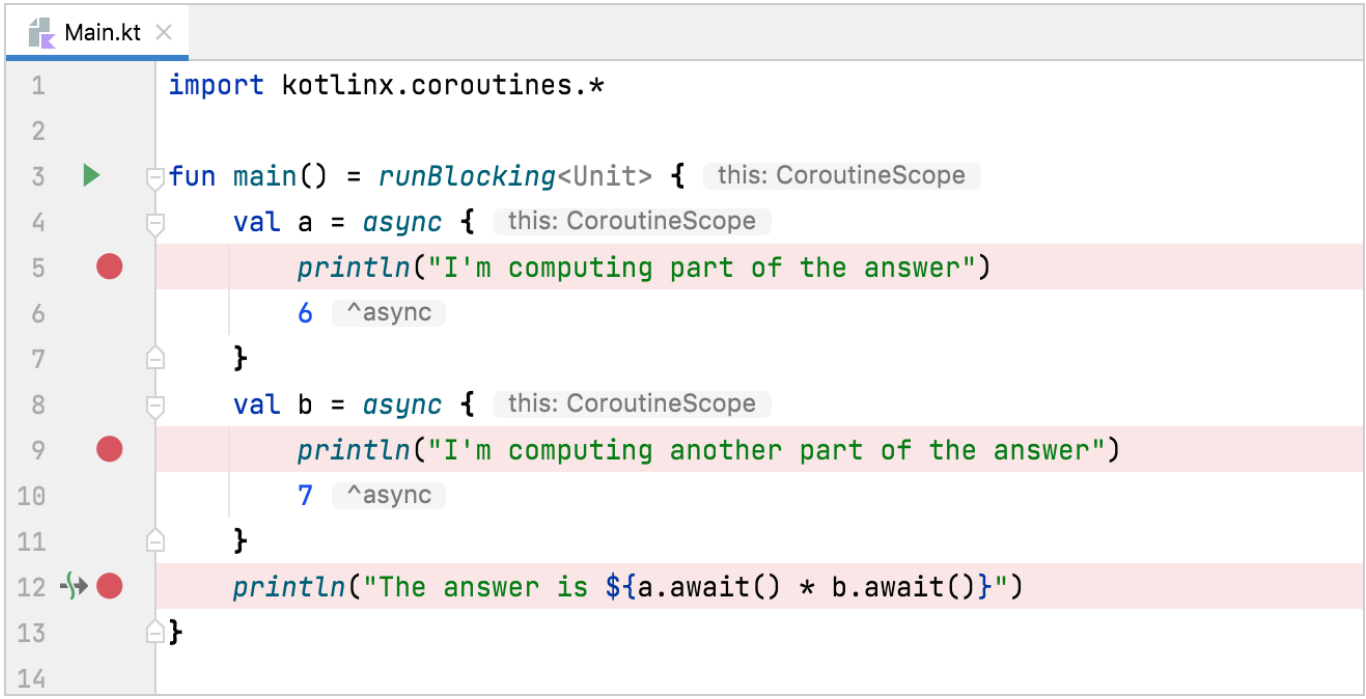
5. Build the code by clicking **Build Project**.



Build an application

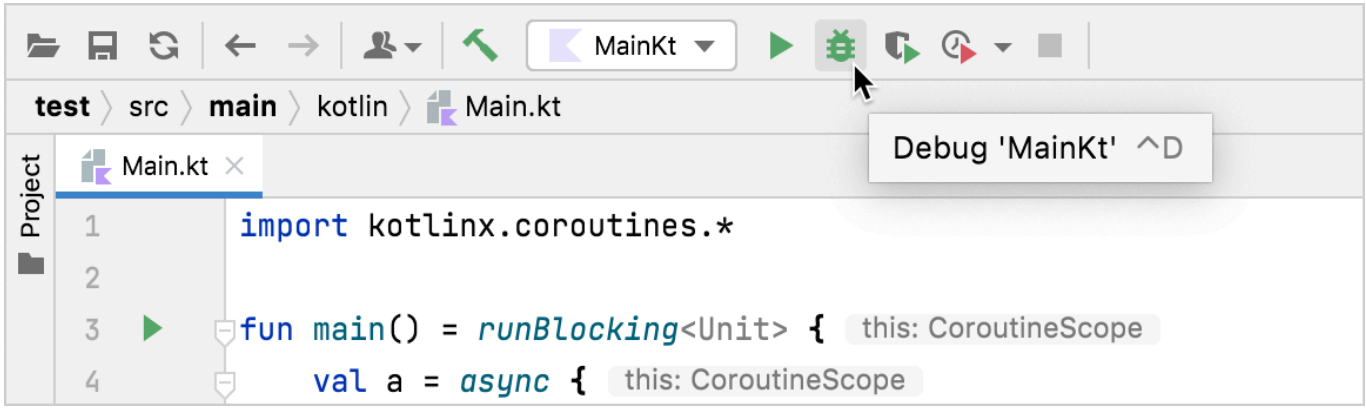
Debug coroutines

1. Set breakpoints at the lines with the `println()` function call:



Build a console application

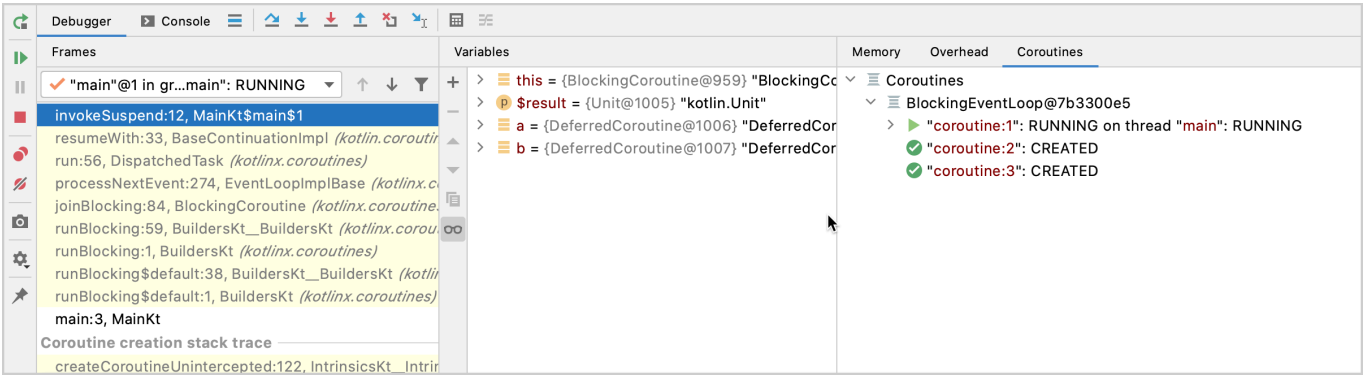
2. Run the code in debug mode by clicking **Debug** next to the run configuration at the top of the screen.



Build a console application

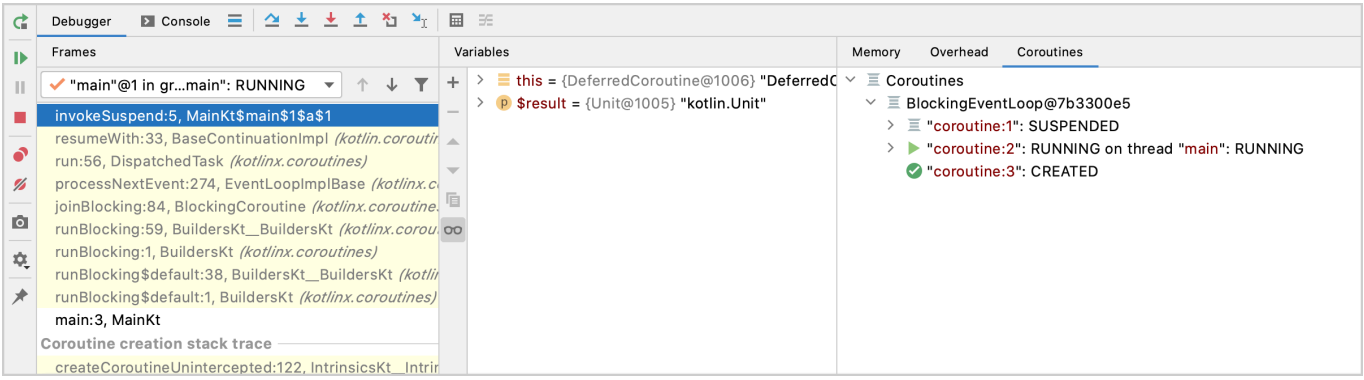
The **Debug** tool window appears:

- The **Frames** tab contains the call stack.
- The **Variables** tab contains variables in the current context.
- The **Coroutines** tab contains information on running or suspended coroutines. It shows that there are three coroutines. The first one has the **RUNNING** status, and the other two have the **CREATED** status.



Debug the coroutine

3. Resume the debugger session by clicking **Resume Program** in the **Debug** tool window:



Debug the coroutine

Now the **Coroutines** tab shows the following:

- The first coroutine has the **SUSPENDED** status – it is waiting for the values so it can multiply them.
- The second coroutine is calculating the `a` value – it has the **RUNNING** status.
- The third coroutine has the **CREATED** status and isn't calculating the value of `b`.

4. Resume the debugger session by clicking **Resume Program** in the **Debug** tool window:



Build a console application

Now the **Coroutines** tab shows the following:

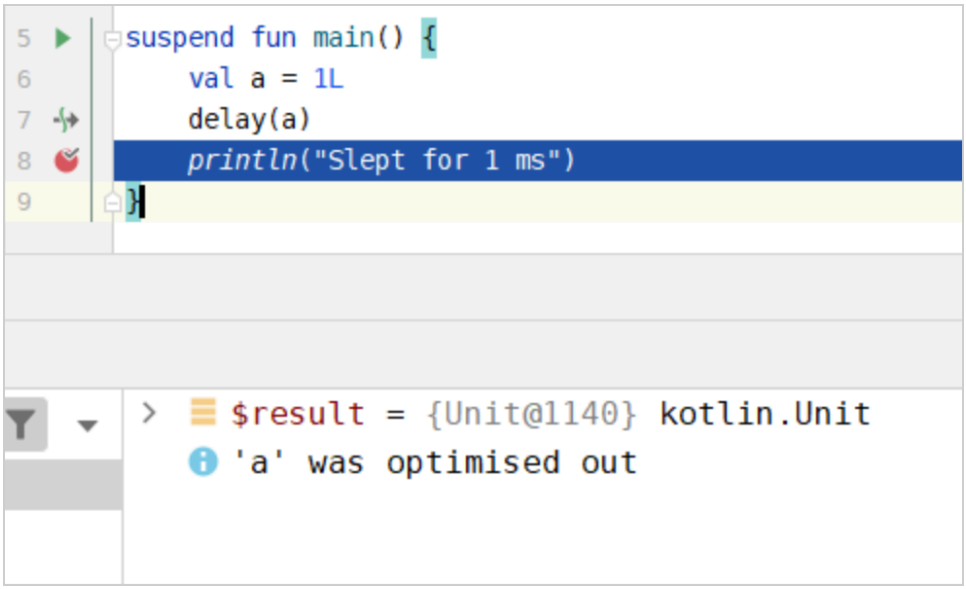
- The first coroutine has the **SUSPENDED** status – it is waiting for the values so it can multiply them.
- The second coroutine has computed its value and disappeared.

- The third coroutine is calculating the value of `b` – it has the **RUNNING** status.

Using IntelliJ IDEA debugger, you can dig deeper into each coroutine to debug your code.

Optimized-out variables

If you use `suspend` functions, in the debugger, you might see the "was optimized out" text next to a variable's name:



Variable "a" was optimized out

This text means that the variable's lifetime was decreased, and the variable doesn't exist anymore. It is difficult to debug code with optimized variables because you don't see their values. You can disable this behavior with the `-Xdebug` compiler option.

Never use this flag in production: `-Xdebug` can cause memory leaks.

Debug Kotlin Flow using IntelliJ IDEA – tutorial

This tutorial demonstrates how to create Kotlin Flow and debug it using IntelliJ IDEA.

The tutorial assumes you have prior knowledge of the coroutines and flow concepts.

Create a Kotlin flow

Create a Kotlin flow with a slow emitter and a slow collector:

1. Open a Kotlin project in IntelliJ IDEA. If you don't have a project, create one.
2. To use the `kotlinx.coroutines` library in a Gradle project, add the following dependency to `build.gradle(.kts)`:

Kotlin

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0")
}
```

Groovy

```
dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0'
}
```

For other build systems, see instructions in the `kotlinx.coroutines` README.

3. Open the `Main.kt` file in `src/main/kotlin`.

   The `src` directory contains Kotlin source files and resources. The `Main.kt` file contains sample code that will print `Hello World!`.

4. Create the `simple()` function that returns a flow of three numbers:

   - Use the `delay()` function to imitate CPU-consuming blocking code. It suspends the coroutine for 100 ms without blocking the thread.
   - Produce the values in the `for` loop using the `emit()` function.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

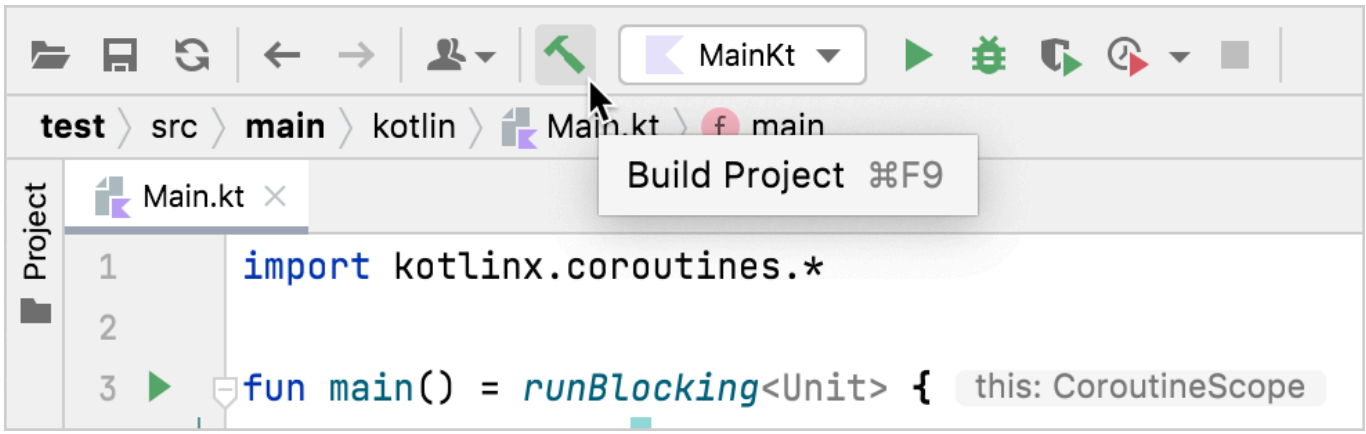
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}
```

5. Change the code in the `main()` function:

   - Use the `runBlocking()` block to wrap a coroutine.
   - Collect the emitted values using the `collect()` function.
   - Use the `delay()` function to imitate CPU-consuming code. It suspends the coroutine for 300 ms without blocking the thread.
   - Print the collected value from the flow using the `println()` function.

```
fun main() = runBlocking {
    simple()
        .collect { value ->
            delay(300)
            println(value)
        }
}
```

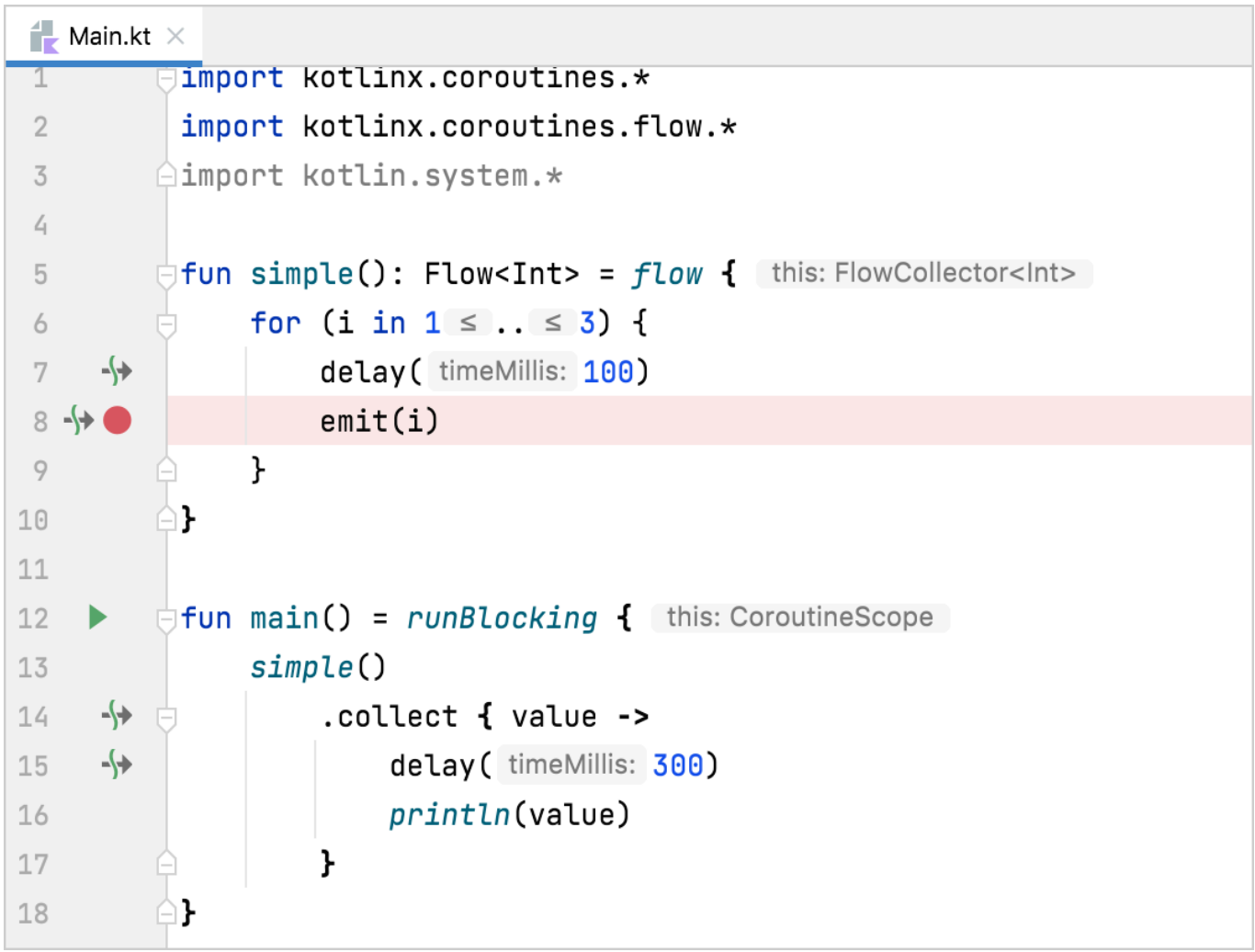
6. Build the code by clicking **Build Project**.



Build an application

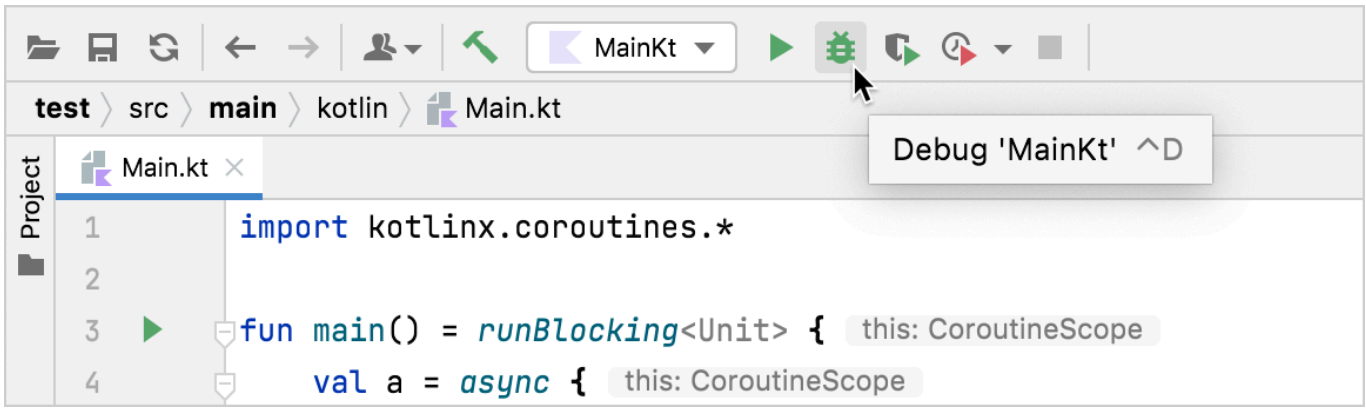
Debug the coroutine

1. Set a breakpoint at the line where the `emit()` function is called:



Build a console application

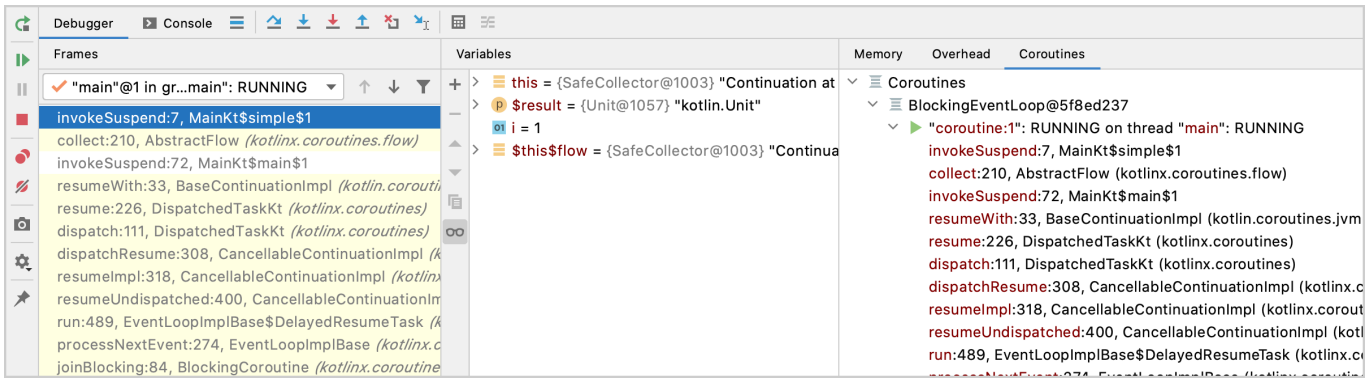
2. Run the code in debug mode by clicking **Debug** next to the run configuration at the top of the screen.



Build a console application

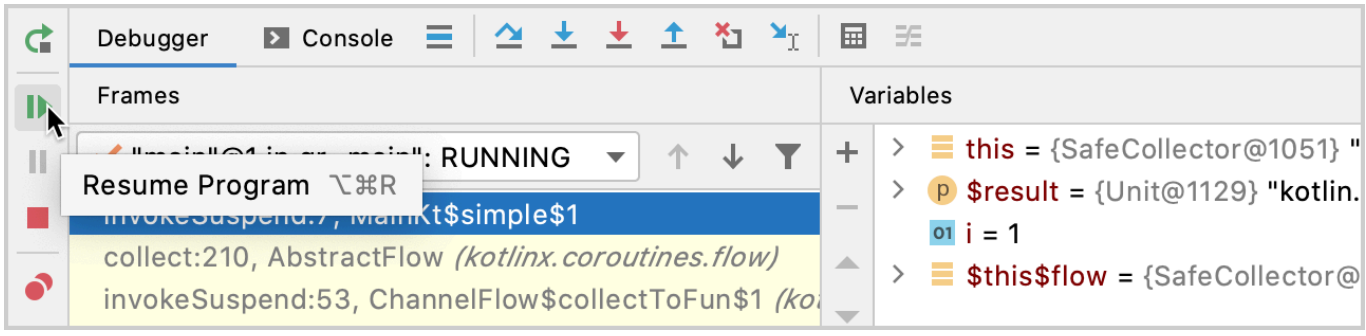
The **Debug** tool window appears:

- The **Frames** tab contains the call stack.
- The **Variables** tab contains variables in the current context. It tells us that the flow is emitting the first value.
- The **Coroutines** tab contains information on running or suspended coroutines.



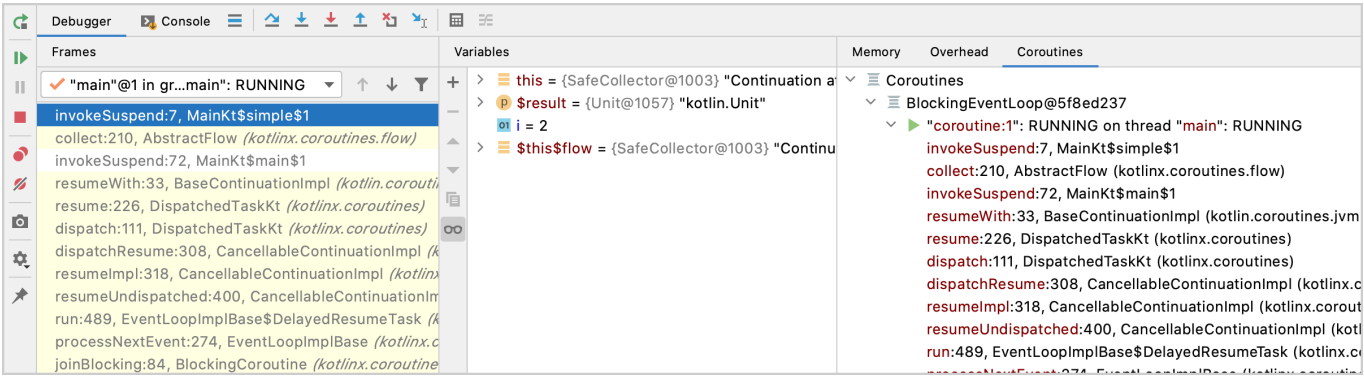
Debug the coroutine

3. Resume the debugger session by clicking **Resume Program** in the **Debug** tool window. The program stops at the same breakpoint.



Debug the coroutine

Now the flow emits the second value.



Debug the coroutine

Optimized-out variables

If you use `suspend` functions, in the debugger, you might see the "was optimized out" text next to a variable's name:



Variable "a" was optimized out

This text means that the variable's lifetime was decreased, and the variable doesn't exist anymore. It is difficult to debug code with optimized variables because you don't see their values. You can disable this behavior with the `-Xdebug` compiler option.

Never use this flag in production: `-Xdebug` can cause memory leaks.

Add a concurrently running coroutine

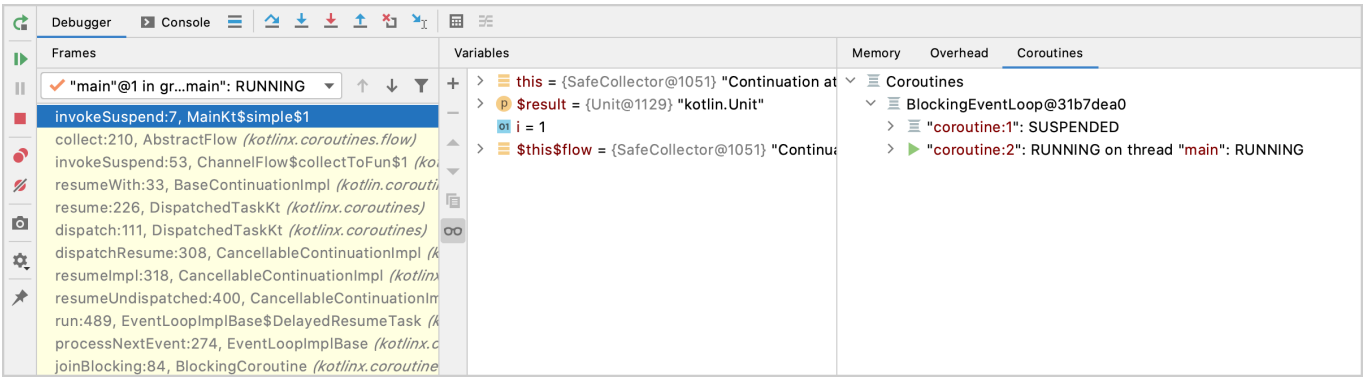
1. Open the `Main.kt` file in `src/main/kotlin`.
2. Enhance the code to run the emitter and collector concurrently:
   - Add a call to the `buffer()` function to run the emitter and collector concurrently. `buffer()` stores emitted values and runs the flow collector in a separate coroutine.

```
fun main() = runBlocking<Unit> {
    simple()
        .buffer()
        .collect { value ->
            delay(300)
            println(value)
        }
}
```

3. Build the code by clicking **Build Project**.

Debug a Kotlin flow with two coroutines

1. Set a new breakpoint at `println(value)`.
2. Run the code in debug mode by clicking **Debug** next to the run configuration at the top of the screen.

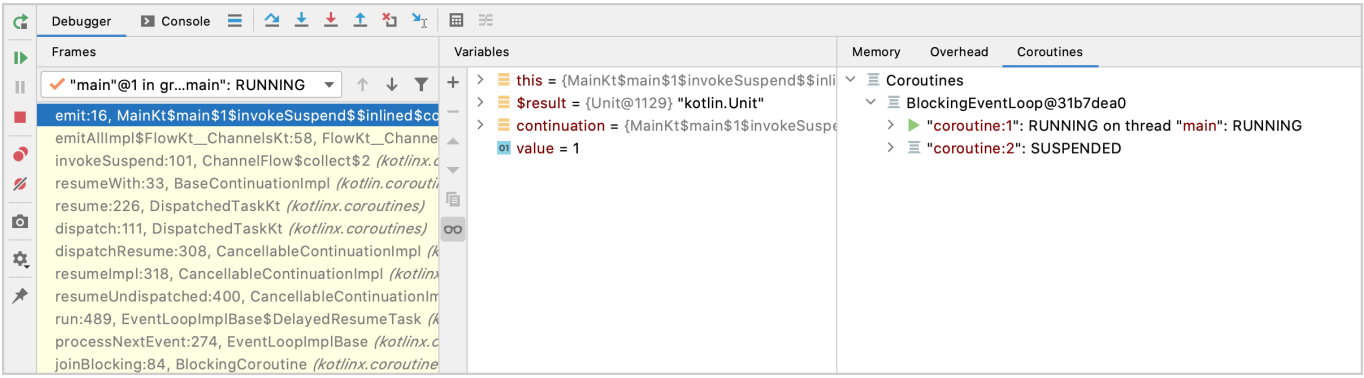


Build a console application

The **Debug** tool window appears.

In the **Coroutines** tab, you can see that there are two coroutines running concurrently. The flow collector and emitter run in separate coroutines because of the `buffer()` function. The `buffer()` function buffers emitted values from the flow. The emitter coroutine has the **RUNNING** status, and the collector coroutine has the **SUSPENDED** status.

3. Resume the debugger session by clicking **Resume Program** in the **Debug** tool window.



Debugging coroutines

Now the collector coroutine has the **RUNNING** status, while the emitter coroutine has the **SUSPENDED** status.

You can dig deeper into each coroutine to debug your code.

Serialization

Serialization is the process of converting data used by an application to a format that can be transferred over a network, or stored in a database or a file. Deserialization is the opposite process of converting external data back into a runtime object. Together, they are essential to most applications that exchange data with third parties.

Some data serialization formats, such as JSON and Protocol Buffers, are particularly common. These formats are language-neutral and platform-neutral, so you can use them to exchange data between systems written in any modern language. Kotlin provides this functionality through the `kotlinx.serialization` libraries, which support multiple platforms and data formats.

If you're new to serialization in Kotlin, we recommend starting with the Get Started with Serialization tutorial. It walks you through adding the Kotlin serialization library to your project and shows you how to serialize and deserialize your first class.



Get started with serialization

Kotlin serialization libraries

Kotlin serialization offers support for all platforms, including JVM, JavaScript, and Native. You can use the same dependency declaration regardless of the target platform.

Kotlin serialization supports various serialization formats, such as JSON, CBOR, and Protocol buffers through different serialization format libraries. These libraries build on the core `kotlinx.serialization` library. For the complete list of supported serialization formats, see Supported serialization formats.

All Kotlin serialization format libraries are part of the `org.jetbrains.kotlinx:` group, with names starting with `kotlinx-serialization-` and suffixes that reflect the serialization format. For example:

- `org.jetbrains.kotlinx:kotlinx-serialization-json` provides JSON serialization.
- `org.jetbrains.kotlinx:kotlinx-serialization-cbor` provides CBOR serialization.

The `kotlinx.serialization` libraries follow their own versioning, independent of Kotlin. You can find the latest release versions on GitHub.

> When you're not using a specific format library, for example, when you're writing your own serialization format, use the `kotlinx-serialization-core` library as the dependency.

Supported serialization formats

`kotlinx.serialization` includes serialization format libraries for various formats:

Format	Artifact ID	Platform	Status
JSON	`kotlinx-serialization-json`	All supported platforms	Stable
HOCON	`kotlinx-serialization-hocon`	JVM only	Experimental
Protocol Buffers	`kotlinx-serialization-protobuf`	All supported platforms	Experimental
CBOR	`kotlinx-serialization-cbor`	All supported platforms	Experimental
Properties	`kotlinx-serialization-properties`	All supported platforms	Experimental

All serialization format libraries, except for the JSON serialization library (`kotlinx-serialization-json`), are Experimental. Their APIs might change at any time. For more details about JSON serialization, see JSON serialization overview.

There are also community-maintained libraries that support more serialization formats, such as YAML or Apache Avro.

You can find out more about experimental serialization formats in Alternative and custom formats.

Supported serialization types

Kotlin serialization supports a variety of built-in types, including all primitive types and most composite types from the Kotlin standard library like the `List` type. For more information, see Serialize built-in types.

Additionally, classes annotated with `@Serializable` are fully supported for serialization, enabling the conversion of class instances to and from formats like JSON. For more information, see Serialize classes.

What's next

- Learn the basics of Kotlin serialization in the Get started with serialization tutorial.
- See how the Kotlin serialization library processes primitives, collections, and other built-in types
- Explore more complex JSON serialization scenarios in the JSON serialization overview.
- Dive into Serialize classes to learn how to serialize classes and modify the default behavior of the `@Serializable` annotation.

- Learn how to obtain generated serializers, create custom serializers, and apply serializers in Create and use serializers.
- See how to serialize different types through a shared base type in Serialize polymorphic classes.
- Explore experimental serialization formats, including CBOR, ProtoBuf, and Properties, and learn how to create custom formats in Alternative and custom formats.

Get started with Kotlin serialization

Serialization converts objects into a format you can store or transmit and later reconstruct.

Kotlin serialization supports multiple formats. This tutorial shows you how to add the necessary plugins and dependencies for Kotlin serialization, and how to serialize and deserialize objects in JSON format.

Add plugins and dependencies

To include the `kotlinx.serialization` library in your project, add the corresponding plugin and dependency configuration based on your build tool:

Gradle Kotlin

```
// build.gradle.kts
plugins {
    kotlin("plugin.serialization") version "2.4.20"
}

dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.11.0")
}
```

Gradle Groovy

```
// build.gradle
plugins {
    id 'org.jetbrains.kotlin.plugin.serialization' version '2.4.20'
}

dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-serialization-json:1.11.0'
}
```

Maven

```
<!-- pom.xml -->
<properties>
    <kotlin.version>2.4.20</kotlin.version>
    <serialization.version>1.11.0</serialization.version>
</properties>

<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <executions>
                <execution>
                    <id>compile</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <compilerPlugins>
                    <plugin>kotlinx-serialization</plugin>
                </compilerPlugins>
            </configuration>
            <dependencies>
                <dependency>
                    <groupId>org.jetbrains.kotlin</groupId>
                    <artifactId>kotlin-maven-serialization</artifactId>
                    <version>${kotlin.version}</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>

<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlinx</groupId>
        <artifactId>kotlinx-serialization-json</artifactId>
        <version>${serialization.version}</version>
    </dependency>
</dependencies>
```

To set up the Kotlin compiler plugin for Bazel, follow the example from the rules_kotlin repository. Bazel isn't officially supported by the Kotlin team, and this repository is maintained independently.

Add the library to a multiplatform project

To use Kotlin serialization for JSON in multiplatform projects, add the JSON serialization library dependency to your common source set:

```
commonMain {
   dependencies {
      implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.11.0")
   }
}
```

This dependency automatically includes the core serialization library as well.

Configure R8 for Kotlin serialization in Android projects

The Kotlin serialization library includes default ProGuard rules, so you don't need additional setup to keep serializers for all serializable classes after shrinking. However, these rules don't apply to classes with named companion objects.

To retain serializers for classes with named companion objects, add rules based on the compatibility mode you use to your `proguard-rules.pro` file:

R8 compatibility mode

```
# Serializer for classes with named companion objects are retrieved using getDeclaredClasses
# If you have any such classes, replace the examples below with your own
-keepattributes InnerClasses # Required for getDeclaredClasses

-if @kotlinx.serialization.Serializable class
com.example.myapplication.HasNamedCompanion, # <-- List serializable classes with named companions
com.example.myapplication.HasNamedCompanion2
{
    static **$* *;
}
-keepnames class <1>$$serializer { # Using -keepnames is enough for the serializer() call to reference
the class correctly
    static <1>$$serializer INSTANCE;
}
```

R8 full mode

```
# Serializer for classes with named companion objects are retrieved using getDeclaredClasses
# If you have any such classes, replace the examples below with your own
-keepattributes InnerClasses # Required for getDeclaredClasses

-if @kotlinx.serialization.Serializable class
com.example.myapplication.HasNamedCompanion, # <-- List serializable classes with named companions
com.example.myapplication.HasNamedCompanion2
{
    static **$* *;
}
-keepnames class <1>$$serializer { # Using -keepnames is enough for the serializer() call to reference
the class correctly
    static <1>$$serializer INSTANCE;
}

# Keep both serializer and serializable classes to save the attribute InnerClasses
-keepclasseswithmembers, allowshrinking, allowobfuscation, allowaccessmodification class
com.example.myapplication.HasNamedCompanion, # <-- List serializable classes with named companions
com.example.myapplication.HasNamedCompanion2
{
    *;
}
```

You can exclude serializable classes that are never serialized at runtime by using custom ProGuard rules with narrower class specifications.

Serialize objects to JSON

In Kotlin, you can serialize objects to JSON using the `kotlinx.serialization` library.

To make a class serializable, you need to mark it with the `@Serializable` annotation. This annotation instructs the compiler to generate the code required for serializing and deserializing instances of the class. For more information, see The `@Serializable` annotation.

Let's look at an example:

1. Import declarations from the necessary serialization libraries:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*
```

2. Make a class serializable by annotating it with `@Serializable`:

```
@Serializable
data class Book(val yearPublished: Int, val title: String)
```

> The `@Serializable` annotation enables default serialization of all properties with backing fields. You can customize serialization behavior with property-level annotations, optional properties, and more.
>
> For more information, see Serialize classes.

3. Use the `Json.encodeToString()` function to serialize an instance of this class:

```
// Imports declarations from the serialization and JSON handling libraries
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Marks the Book class as serializable
@Serializable
data class Book(val yearPublished: Int, val title: String)

fun main() {
    // Serializes an instance of the Book class into a JSON string
    val json = Json.encodeToString(Book(1937, "The Hobbit"))
    println(json)
    // {"yearPublished":1937,"title":"The Hobbit"}
}
```

As a result, you get a string containing the state of this object in JSON format:
`{"yearPublished":1937,"title":"The Hobbit"}`



You can also serialize a collection of objects in a single call:

```
val bookList = listOf(Book(1937, "The Hobbit"), Book(1867, "War and Peace"))
val jsonList = Json.encodeToString(bookList)
```

Deserialize objects from JSON

Deserialization converts a JSON string back into an object.

To deserialize an object from JSON in Kotlin:

1. Import declarations from the necessary serialization libraries:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*
```

2. Make a class serializable by annotating it with `@Serializable`:

```
@Serializable
data class Book(val yearPublished: Int, val title: String)
```

3. Use the `Json.decodeFromString()` function to deserialize an object from JSON:

```
// Imports declarations from the serialization and JSON handling libraries
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Marks the Book class as serializable
@Serializable
data class Book(val yearPublished: Int, val title: String)

fun main() {
    // Deserializes a JSON string into an instance of the Book class
    val obj = Json.decodeFromString<Book>("""{"yearPublished":1937, "title": "The Hobbit"}""")
    println(obj)
    // Book(yearPublished=1937, title=The Hobbit)
}
```

Congratulations! You have successfully serialized an object to JSON and deserialized it back into an object in Kotlin.

What's next

- Learn how to serialize basic types such as primitives and strings, as well as certain standard library classes, in Serialize built-in types.
- Discover how to customize class serialization and adjust the default behavior of the `@Serializable` annotation in Serialize classes.
- Dive deeper into handling JSON data and configuring JSON serialization in the JSON serialization overview.

Serialize built-in types

The Kotlin serialization library supports a variety of built-in types, including basic types such as primitives and strings, as well as certain standard library classes. The following sections describe these types in detail and show how to serialize them.

Basic types

Kotlin serialization provides built-in serializers for types that are represented as a single value in serialized data. This includes primitives, strings, and enums.

For example, here's how you can serialize a `Long` type:

```
// Imports the necessary library declarations
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
class Data(val signature: Long)

fun main() {
    val data = Data(0x1CAFE2FEED0BABE0)
    println(Json.encodeToString(data))
    // {"signature":2067120338512882656}
}
```

Numbers

You can serialize all Kotlin number types, including integers and floating-point numbers, using their natural JSON representations:

```kotlin
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlin.math.PI

@Serializable
class Data(
    val answer: Int,
    val pi: Double
)

fun main() {
    val data = Data(42, PI)
    println(Json.encodeToString(data))
    // {"answer":42,"pi":3.141592653589793}
}
```

Unsigned numbers

Kotlin serialization supports Kotlin's unsigned integer types like `UByte` and `UInt`. In JSON, these values are serialized as regular JSON numbers and preserve their full unsigned range:

```kotlin
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
class Counter(val counted: UByte, val description: String)

fun main() {
    val counted = 239.toUByte()
    println(Json.encodeToString(Counter(counted, "tries")))
    // {"counted":239,"description":"tries"}
}
```

Although JSON preserves the full range of unsigned numbers, other serialization formats may handle them differently. For example, ProtoBuf and CBOR serialize these types using their signed counterparts.

`Long` numbers as strings

You can represent `Long` numbers as strings in JSON. This is useful in JavaScript environments, where JavaScript's `Number` type can't precisely represent all Kotlin `Long` values, which may lead to precision loss.

Use `LongAsStringSerializer` with the `@Serializable` annotation to encode `Long` values as strings in JSON:

```
import kotlinx.serialization.*
import kotlinx.serialization.builtins.*
import kotlinx.serialization.json.*

@Serializable
class Data(
    @Serializable(LongAsStringSerializer::class)
    val signature: Long
)

fun main() {
    val data = Data(0x1CAFE2FEED0BABE0)
    println(Json.encodeToString(data))
    // {"signature":"2067120338512882656"}
}
```

You can also specify serializers like `LongAsStringSerializer` for all properties in a file. For more information, see Specify serializers for a file.

Enum classes

All `enum` classes are serializable by default without the `@Serializable` annotation. When serialized in JSON, an `enum` is encoded as a string:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// The @Serializable annotation isn't required for enum classes
enum class Status { SUPPORTED }

@Serializable
class Project(val name: String, val status: Status)

fun main() {
    val data = Project("kotlinx.serialization", Status.SUPPORTED)
    println(Json.encodeToString(data))
    // {"name":"kotlinx.serialization","status":"SUPPORTED"}
}
```

When targeting Kotlin/JS or Kotlin/Native, you must use the `@Serializable` annotation for an `enum` class to use it as a root object, such as in `encodeToString<Status>(Status.SUPPORTED)`.

Customize serial names of enum entries

To customize the serial names of enum entries, use the `@SerialName` annotation and mark the enum class with `@Serializable`:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Requires the @Serializable annotation because of @SerialName
@Serializable
enum class Status { @SerialName("maintained") SUPPORTED }

@Serializable
class Project(val name: String, val status: Status)

fun main() {
    val data = Project("kotlinx.serialization", Status.SUPPORTED)
    println(Json.encodeToString(data))
    // {"name":"kotlinx.serialization","status":"maintained"}
}
```

For more information on customizing serial names, see Customize serial names.

Standard library types

Kotlin serialization supports several types from the standard library, but some classes, such as ranges and the `Regex` class, aren't supported.

Pair and triple

You can serialize the `Pair` and `Triple` classes from the Kotlin standard library:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
class Project(val name: String)

fun main() {
    val pair = 1 to Project("kotlinx.serialization")
    println(Json.encodeToString(pair))
    // {"first":1,"second":{"name":"kotlinx.serialization"}}
}
```

Collections

Kotlin serialization supports collection types, including both read-only and mutable variants of `List`, `Set`, and `Map`. It also supports their concrete implementations such as `ArrayList` and `LinkedHashSet`, as well as generic and primitive array types. The way these collections are represented depends on the serialization format.

In JSON, lists and sets are serialized as JSON arrays, and maps are represented as JSON objects.

Kotlin uses the declared type to deserialize JSON. During deserialization, the type of the resulting object is determined by the static type specified in the source code. This type can be either the type of the property or the type parameter of the decoding function.

Serialize lists

Kotlin serialization serializes `List` types as JSON arrays. Here's an example with a list of classes:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
class Project(val name: String)

fun main() {
    val list = listOf(
        Project("kotlinx.serialization"),
        Project("kotlinx.coroutines")
    )
    println(Json.encodeToString(list))
    // [{"name":"kotlinx.serialization"},{"name":"kotlinx.coroutines"}]
}
```

Serialize sets

`Set` types are serialized as JSON arrays, just like `List` types:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
class Project(val name: String)

fun main() {
    val set = setOf(
        Project("kotlinx.serialization"),
        Project("kotlinx.coroutines")
    )
    println(Json.encodeToString(set))
    // [{"name":"kotlinx.serialization"},{"name":"kotlinx.coroutines"}]
}
```

By default, you can deserialize sets with duplicate entries. The behavior for handling duplicates is implementation-defined.

Serialize maps

Kotlin serialization supports `Map` types with primitive or enum keys:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
class Project(val name: String)

fun main() {
    // Creates a map with Int keys
    val map = mapOf(
        1 to Project("kotlinx.serialization"),
        2 to Project("kotlinx.coroutines")
    )
    println(Json.encodeToString(map))
    // {"1":{"name":"kotlinx.serialization"},"2":{"name":"kotlinx.coroutines"}}
}
```

Map serialization depends on the format. In JSON, maps are represented as objects. Since JSON object keys are always strings, keys are encoded as strings even if they are numbers in Kotlin. Other formats, such as CBOR, support maps with non-primitive keys and preserve them as such.

> JSON doesn't natively support complex or composite keys. To encode structured objects as map keys, see Allow structured map keys.

Deserialization behavior of collections

Kotlin uses the declared type to deserialize JSON. For example, with collections, a `List` preserves duplicates, while a `Set` enforces uniqueness:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
data class Data(
    val a: List<Int>,
    val b: Set<Int>
)

fun main() {
    val data = Json.decodeFromString<Data>("""
        {
            "a": [42, 42],
            "b": [42, 42]
        }
    """)
    // Duplicates are removed from data.b because the Set type enforces unique elements
    println(data)
    // Data(a=[42, 42], b=[42])
}
```

> For more information about collections in Kotlin, see Collections overview.

Unit and singleton objects

Kotlin's `Unit` type and other singleton objects are serializable. A singleton is a class with only one instance, where the state is defined by the object itself rather than by external properties. In JSON, singleton objects are serialized as empty structures:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
object SerializationVersion {
    val libraryVersion: String = "1.0.0"
}

fun main() {
    println(Json.encodeToString(SerializationVersion))
    // {}
    println(Json.encodeToString(Unit))
    // {}
}
```

You can use serialized singleton objects in closed polymorphic hierarchies to represent cases without additional fields.

Duration and Instant

Kotlin's `Duration` type is serialized to a string using the ISO-8601-2 format:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlin.time.*

fun main() {
    val duration = 1000.toDuration(DurationUnit.SECONDS)
    println(Json.encodeToString(duration))
    // "PT16M40S"
}
```

You can also serialize Kotlin's `Instant` type as a string representing a point in time using the ISO-8601-1 format:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlin.time.*

fun main() {
    val instant = Instant.fromEpochMilliseconds(1607505416124)
    println(Json.encodeToString(instant))
    // "2020-12-09T09:16:56.124Z"
}
```

Nothing

The `Nothing` type is serializable by default. It has no instances, so encoding or decoding it throws an exception. Use `Nothing` when a type is syntactically required, but not involved in serialization, like in a polymorphic hierarchy with a generic base type:

```
import kotlinx.serialization.*
import kotlinx.serialization.builtins.*
import kotlinx.serialization.json.*

@Serializable
sealed class ParametrizedParent<out R> {
    @Serializable
    data class ChildWithoutParameter(val value: Int) : ParametrizedParent<Nothing>()
}

fun main() {
    println(Json.encodeToString(ParametrizedParent.ChildWithoutParameter(42)))
    // {"value":42}
}
```

What's next

- Dive into Serialize classes to learn how to serialize classes and how to modify the default behavior of the `@Serializable` annotation.

- To explore more complex JSON serialization scenarios, see JSON serialization overview.
- Learn more about polymorphism and serializing different types through a shared base in Serialize polymorphic classes.

Serialize classes

The `@Serializable` annotation enables the default serialization of all class properties with backing fields. You can customize this behavior to fit your specific needs. This page covers various serialization techniques, showing you how to specify which properties are serialized, and how the serialization process is managed.

Before starting, make sure you've imported the necessary library dependencies.

The @Serializable annotation

The `@Serializable` annotation enables the automatic serialization of class properties, allowing a class to be converted to and from formats such as JSON.

In Kotlin, only properties with backing fields are serialized:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
class Project(
    // Property with a backing field - serialized
    var name: String
) {
    // Property with a backing field - serialized
    var stars: Int = 0

    // Getter-only property without a backing field - not serialized
    val path: String
        get() = "kotlin/$name"

    // Delegated property - not serialized
    var id by ::name
}

fun main() {
    val data = Project("kotlinx.serialization").apply { stars = 9000 }
    // Prints only the name and the stars properties
    println(Json.encodeToString(data))
    // {"name":"kotlinx.serialization","stars":9000}
}
```

Serialize class references

Classes annotated with `@Serializable` can contain properties that reference other classes. The referenced classes must also be annotated with `@Serializable`. When encoded to JSON, this results in a nested JSON object:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
// The owner property references another serializable class User
class Project(val name: String, val owner: User)

// The referenced class must also be annotated with @Serializable
@Serializable
class User(val name: String)

fun main() {
    val owner = User("kotlin")
    val data = Project("kotlinx.serialization", owner)
    println(Json.encodeToString(data))
    // {"name":"kotlinx.serialization","owner":{"name":"kotlin"}}
}
```

To reference non-serializable classes, annotate the corresponding properties with `@Transient`, or provide a custom serializer for them.

Serialization of repeated object references

Kotlin serialization is designed to encode and decode plain data. It doesn't support reconstruction of arbitrary object graphs with repeated object references. For example, when serializing an object that references the same instance twice, it's encoded twice:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
class Project(val name: String, val owner: User, val maintainer: User)

@Serializable
class User(val name: String)

fun main() {
    val owner = User("kotlin")
    // References owner twice
    val data = Project("kotlinx.serialization", owner, owner)
    println(Json.encodeToString(data))
    // {"name":"kotlinx.serialization","owner":{"name":"kotlin"},"maintainer":{"name":"kotlin"}}
}
```

If you attempt to serialize a circular structure, it results in stack overflow. To exclude references from serialization, use the @Transient annotation.

Serialize generic classes

Generic classes in Kotlin support type-polymorphism, which is enforced by Kotlin serialization at compile-time. For example, consider a generic serializable class `Payload<T>`:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
// The Payload<T> class can be used with built-in types like Int
// or with @Serializable user-defined types like Repository
class Payload<T>(val value: T)

@Serializable
data class Repository(val name: String, val language: String)

@Serializable
class BackupData(
    val issueCount: Payload<Int>,
    val mainRepo: Payload<Repository>
)

fun main() {
    val backup = BackupData(
        Payload(42),
        Payload(Repository("kotlinx.serialization", "Kotlin"))
    )
    println(Json.encodeToString(backup))
    // {"issueCount":{"value":42},"mainRepo":{"value":
{"name":"kotlinx.serialization","language":"Kotlin"}}}
}
```

When you serialize a generic class like `Box<T>`, the JSON output depends on the actual type you specify for `T` at compile time. If that type isn't serializable, you get a compile-time error.

Optional properties

Properties with *default values* are optional during deserialization and can be skipped during serialization, if the format is configured to do so. For example, the default JSON configuration skips encoding default values.

Set default values for optional properties

In Kotlin, an object can be deserialized only when all its properties are present in the input. To make a property optional for serialization, set a default value that's used when no value is provided in the input:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
// Sets a default value for the optional language property
data class Project(val name: String, val language: String = "Kotlin")

fun main() {
    val data = Json.decodeFromString<Project>("""
        {"name":"kotlinx.serialization"}
    """)
    println(data)
    // Project(name=kotlinx.serialization, language=Kotlin)
}
```

Serialize nullable properties

Kotlin serialization natively supports nullable properties. Like other default values, `null` values aren't encoded in the JSON output:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
// Defines a class with the renamedTo nullable property that has a null default value
class Project(val name: String, val renamedTo: String? = null)

fun main() {
    val data = Project("kotlinx.serialization")
    // The renamedTo property isn't encoded because its value is null
    println(Json.encodeToString(data))
    // {"name":"kotlinx.serialization"}
}
```

Additionally, Kotlin's null safety is strongly enforced during deserialization. If a JSON object contains a `null` value for a non-nullable property, an exception is thrown even when the property has a default value:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
data class Project(val name: String, val language: String = "Kotlin")

fun main() {
    val data = Json.decodeFromString<Project>("""
        {"name":"kotlinx.serialization","language":null}
    """)
    println(data)
    // JsonDecodingException
}
```

If you need to handle `null` values from third-party JSON, you can coerce them to a default value.

You can also omit explicit `null` values from the encoded JSON with the `explicitNulls` property.

Initializers in optional properties

When the serialization input includes a value for an optional property, the property's initializer isn't called. For this reason, avoid using code with side effects in property initializers.

Here's an example:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

fun computeLanguage(): String {
    println("Computing")
    return "Kotlin"
}

@Serializable
// Skips the initializer if language is in the input
data class Project(val name: String, val language: String = computeLanguage())

fun main() {
    val data = Json.decodeFromString<Project>("""
        {"name":"kotlinx.serialization","language":"Java"}
    """)
    println(data)
    // Project(name=kotlinx.serialization, language=Java)
}
```

In this example, since the `language` property is specified in the input, the `Computing` string isn't printed in the output.

Manage the serialization of default properties with `@EncodeDefault`

By default, JSON serialization excludes properties that have default values. This reduces the size of the serialized data and avoids unnecessary visual clutter.

Here's an example where the `language` property is excluded from the output because its value equals the default:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
data class Project(val name: String, val language: String = "Kotlin")

fun main() {
    val data = Project("kotlinx.serialization")
    println(Json.encodeToString(data))
    // {"name":"kotlinx.serialization"}
}
```

You can configure a `Json` instance to encode default values for all properties by default.

To always serialize a property, regardless of its value or format settings, use the `@EncodeDefault` annotation. Alternatively, you can set the `EncodeDefault.Mode` parameter to change this behavior.

Let's look at an example where the `language` property is always included in the serialized output, while the `projects` property is excluded when it's an empty list:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.EncodeDefault.Mode.NEVER

@Serializable
data class Project(
    val name: String,
    // Always includes the language property in the serialized output
    // even if it has the default value "Kotlin"
    @EncodeDefault val language: String = "Kotlin"
)

@Serializable
data class User(
    val name: String,
    // Excludes projects when it's an empty list, even if it has a default value
    @EncodeDefault(NEVER) val projects: List<Project> = emptyList()
)

fun main() {
    val adminUser = User("Alice", listOf(Project("kotlinx.serialization")))
    val guestUser = User("Bob")
    // Serializes projects because it contains a value
    // language is always serialized
    println(Json.encodeToString(adminUser))
    // {"name":"Alice","projects":[{"name":"kotlinx.serialization","language":"Kotlin"}]}

    // Excludes projects because it's an empty list
    // and EncodeDefault.Mode is set to NEVER, so it's not serialized
    println(Json.encodeToString(guestUser))
    // {"name":"Bob"}
}
```

Make properties required with the `@Required` annotation

Annotate a property with `@Required` to make it required in the input. This enforces that the input contains the property, even if it has a default value:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.Required

@Serializable
// Marks the language property as required
data class Project(val name: String, @Required val language: String = "Kotlin")

fun main() {
    val data = Json.decodeFromString<Project>("""
        {"name":"kotlinx.serialization"}
    """)
    println(data)
    // MissingFieldException
}
```

Customize class serialization

Kotlin serialization provides several ways to modify how classes are serialized. This section covers techniques for customizing property names, controlling property inclusion, and more.

Customize serial names

By default, property names in the serialization output, such as JSON, match their names in the source code.

You can customize these names, called *serial names*, with the `@SerialName` annotation. Use it to make a property name more descriptive in the serialized output:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
// Changes the lang property to language using @SerialName
class Project(val name: String, @SerialName("language") val lang: String)

fun main() {
    val data = Project("kotlinx.serialization", "Kotlin")
    // Prints the more descriptive property name in the JSON output
    println(Json.encodeToString(data))
    // {"name":"kotlinx.serialization","language":"Kotlin"}
}
```

Define constructor properties for serialization

A class annotated with `@Serializable` must declare all parameters in its primary constructor as properties.

If you need to perform additional initialization logic before assigning values to properties, use a secondary constructor. The primary constructor can remain `private` and handle property initialization.

Here's an example where the secondary constructor parses a string into two values, which are then passed to the primary constructor for serialization:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
class Project private constructor(val owner: String, val name: String) {
    // Creates a Project object using a path string
    constructor(path: String) : this(
        owner = path.substringBefore('/'),
        name = path.substringAfter('/')
    )

    val path: String
        get() = "$owner/$name"
}
fun main() {
    println(Json.encodeToString(Project("kotlin/kotlinx.serialization")))
    // {"owner":"kotlin","name":"kotlinx.serialization"}
}
```

Validate data in the primary constructor

After deserialization, the `kotlinx.serialization` plugin runs the class's initializer blocks, just like when you create an instance. This allows you to validate constructor parameters and reject invalid data during deserialization.

Here's an example:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
class Project(val name: String) {
    // Validates that the name is not empty
    init {
        require(name.isNotEmpty()) { "name cannot be empty" }
    }
}

fun main() {
    val data = Json.decodeFromString<Project>("""
        {"name":""}
    """)
    println(data)
    // Exception in thread "main" java.lang.IllegalArgumentException: name cannot be empty
}
```

Exclude properties with the `@Transient` annotation

You can exclude a property from serialization with the `@Transient` annotation. Transient properties must have a default value.

If you explicitly specify a value for a transient property in the input, even if it matches the default value, a `JsonDecodingException` is thrown:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.Transient
import kotlinx.serialization.json.*

@Serializable
// Excludes the language property from serialization
data class Project(val name: String, @Transient val language: String = "Kotlin")

fun main() {
    // Throws an exception even though input matches the default value
    val data = Json.decodeFromString<Project>("""
        {"name":"kotlinx.serialization","language":"Kotlin"}
    """)
    println(data)
    // JsonDecodingException
}
```

> To avoid exceptions from unknown keys in JSON, including those marked with the `@Transient` annotation, enable the `ignoreUnknownKeys` configuration property. For more information, see the Ignore unknown keys section.

What's next

- Explore more complex JSON serialization scenarios in the JSON serialization overview.
- Learn how to obtain generated serializers, create custom serializers, and apply serializers in Create and use serializers.

JSON serialization overview

The Kotlin serialization library allows you to easily convert Kotlin objects to JSON and back. The `Json` class is the primary tool for this, offering flexibility in how JSON is generated and parsed. You can configure `Json` instances to handle specific JSON behaviors or use its default instance for basic tasks.

With the `Json` class, you can:

- Serialize Kotlin objects to JSON strings using the `encodeToString()` function.
- Deserialize JSON strings back to Kotlin objects with the `decodeFromString()` function.
- Work directly with the `JsonElement` when handling complex JSON structures using the `encodeToJsonElement()` and the `decodeFromJsonElement()` functions.
- Use Experimental extension functions to serialize and deserialize I/O sources without creating intermediate strings, including JVM streams as well as `kotlinx-io` and Okio types.

Before you start, import the following declarations from the serialization library:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*
```

Here's a simple example that uses the default `Json` instance to show how JSON serialization works in Kotlin:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
data class User(val name: String, val age: Int)

fun main() {
    // Uses the default Json instance
    val json = Json

    // Creates a User object
    val user = User("Alice", 30)

    // Converts the User object to a JSON string
    val jsonString = json.encodeToString(user)
    println(jsonString)
    // {"name":"Alice","age":30}

    // Converts the JSON string back to a User object
    val deserializedUser = json.decodeFromString<User>(jsonString)
    println(deserializedUser)
    // User(name=Alice, age=30)
}
```

In addition to using the default configuration, you can customize the `Json` instance for specific use cases, such as ignoring unknown keys:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
data class Project(val name: String)

// Configures a Json instance to ignore unknown keys
val customJson = Json {
    ignoreUnknownKeys = true
}

fun main() {
    val data = customJson.decodeFromString<Project>("""
        {"name":"kotlinx.serialization","language":"Kotlin"}
    """)
    println(data)
    // Project(name=kotlinx.serialization)
}
```

What's next

- Learn how to customize `Json` instances to address different use cases for serialization and deserialization.
- Explore advanced JSON element handling to manipulate and work with JSON data before it is parsed or serialized.
- Discover how to transform JSON during serialization and deserialization for more control over your data.

Customize the Json instance

The default `Json` instance strictly follows the JSON specification and the declarations in Kotlin classes.

You can use more flexible JSON features or type conversions by creating a custom `Json` instance with the `Json()` builder function:

```
// Creates a Json instance based on the default configuration, allowing special floating-point values
val customJson = Json {
    allowSpecialFloatingPointValues = true
}

// Use the customJson instance with the same syntax as the default one to encode a string
val jsonString = customJson.encodeToString(Data(Double.NaN))
println(jsonString)
```

`Json` instances created this way are immutable and thread-safe, which makes them safe to store in a top-level property for reuse.

> Reusing custom `Json` instances improves performance by allowing them to cache class-specific information.

You can also base a new `Json` instance on an existing one and modify its settings using the same builder syntax:

```
// Creates a new instance based on an existing Json
val lenientJson = Json(customJson) {
    isLenient = true
    prettyPrint = true
}
```

Customize JSON structure

You can customize how a `Json` instance structures data during encoding and decoding. This allows you to control which values appear in the output and how specific types are represented.

Encode default values

By default, the JSON encoder omits default property values because they are automatically applied to missing properties during decoding. This behavior is especially useful for nullable properties with null defaults, as it avoids writing unnecessary `null` values. For more details, see the Manage serialization of default properties section.

To change this default behavior, set the `encodeDefaults` property to `true` in a `Json` instance:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Configures a Json instance to encode default values
val format = Json { encodeDefaults = true }

@Serializable
class Project(
    val name: String,
    val language: String = "Kotlin",
    val website: String? = null
)

fun main() {
    val data = Project("kotlinx.serialization")

    // Encodes all the property values, including the default ones
    println(format.encodeToString(data))
    // {"name":"kotlinx.serialization","language":"Kotlin","website":null}
}
```

Omit explicit nulls

By default, all `null` values are encoded into JSON output. To omit `null` values, set the `explicitNulls` property to `false` in a `Json` instance:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Configures a Json instance to omit null values during serialization
val format = Json { explicitNulls = false }

@Serializable
data class Project(
    val name: String,
    val language: String,
    val version: String? = "1.2.2",
    val website: String?,
    val description: String? = null
)

fun main() {
    val data = Project("kotlinx.serialization", "Kotlin", null, null, null)
    val json = format.encodeToString(data)

    // Omits version, website, and description properties from the JSON output
    println(json)
    // {"name":"kotlinx.serialization","language":"Kotlin"}

    // Treats missing nullable properties without defaults as null
    // Fills properties that have defaults with their default values
    println(format.decodeFromString<Project>(json))
    // Project(name=kotlinx.serialization, language=Kotlin, version=1.2.2, website=null,
description=null)
}
```

When `explicitNulls` is set to `false`, encoding and decoding can become asymmetrical. In this example, the `version` property is `null` before encoding, but it decodes to `1.2.2`.

> You can configure the decoder to handle certain invalid input values by treating them as missing properties with the `coerceInputValues` property. For more information, see the Coerce input values section.

Allow structured map keys

The JSON format doesn't natively support maps with structured keys, because JSON keys are strings that represent only primitives or enums. To serialize and deserialize maps with user-defined class keys, use the `allowStructuredMapKeys` property:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Configures a Json instance to encode maps with structured keys
val format = Json { allowStructuredMapKeys = true }

@Serializable
data class Project(val name: String)

fun main() {
    val map = mapOf(
        Project("kotlinx.serialization") to "Serialization",
        Project("kotlinx.coroutines") to "Coroutines"
    )
    // Serializes the map with structured keys as a JSON array:
    // [key1, value1, key2, value2,...]
    println(format.encodeToString(map))
    // [{"name":"kotlinx.serialization"},"Serialization",{"name":"kotlinx.coroutines"},"Coroutines"]
}
```

Allow special floating-point values

By default, special floating-point values like `Double.NaN` and infinities aren't supported in JSON because the JSON specification prohibits them.

To enable their encoding and decoding, set the `allowSpecialFloatingPointValues` property to `true` in a `Json` instance:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Configures a Json instance to allow special floating-point values
val format = Json { allowSpecialFloatingPointValues = true }

@Serializable
class Data(
    val value: Double
)

fun main() {
    val data = Data(Double.NaN)
    // Produces a non-standard JSON output used for representing special floating-point values
    println(format.encodeToString(data))
    // {"value":NaN}
}
```

Specify class discriminator for polymorphism

When working with polymorphic data, you can use the `classDiscriminator` property to specify a key name that identifies the type of the serialized polymorphic object. Combined with an explicit serial name defined with the `@SerialName` annotation, this approach gives you full control over the resulting JSON structure:

```kotlin
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Configures a Json instance to use a custom class discriminator
val format = Json { classDiscriminator = "#class" }

@Serializable
sealed class Project {
    abstract val name: String
}

// Specifies a custom serial name for the OwnedProject class
@Serializable
@SerialName("owned")
class OwnedProject(override val name: String, val owner: String) : Project()

// Specifies a custom serial name for the SimpleProject class
@Serializable
@SerialName("simple")
class SimpleProject(override val name: String) : Project()

fun main() {
    val simpleProject: Project = SimpleProject("kotlinx.serialization")
    val ownedProject: Project = OwnedProject("kotlinx.coroutines", "kotlin")

    // Serializes SimpleProject with #class: "simple"
    println(format.encodeToString(simpleProject))
    // {"#class":"simple","name":"kotlinx.serialization"}

    // Serializes OwnedProject with #class: "owned"
    println(format.encodeToString(ownedProject))
    // {"#class":"owned","name":"kotlinx.coroutines","owner":"kotlin"}
}
```

While the `classDiscriminator` property in a `Json` instance lets you specify a single discriminator key for all polymorphic types, the Experimental `@JsonClassDiscriminator` annotation offers more flexibility. It allows you to define a custom discriminator directly on the base class, which is automatically inherited by all its subclasses.

> To learn more about inheritable serial annotations, see `@InheritableSerialInfo`.

Here's an example:

```kotlin
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// The @JsonClassDiscriminator annotation is inheritable, so all subclasses of Base will have the same
discriminator
@Serializable
@OptIn(ExperimentalSerializationApi::class)
@JsonClassDiscriminator("message_type")
sealed class Base

// Inherits the discriminator from Base
@Serializable
sealed class ErrorClass: Base()

// Defines a class that combines a message and an optional error
@Serializable
data class Message(val message: Base, val error: ErrorClass?)

@Serializable
@SerialName("my.app.BaseMessage")
data class BaseMessage(val message: String) : Base()

@Serializable
@SerialName("my.app.GenericError")
data class GenericError(@SerialName("error_code") val errorCode: Int) : ErrorClass()

val format = Json { classDiscriminator = "#class" }

fun main() {
    val data = Message(BaseMessage("not found"), GenericError(404))
    // Uses the discriminator from Base for all subclasses
    println(format.encodeToString(data))
    // {"message":{"message_type":"my.app.BaseMessage","message":"not found"},"error":
{"message_type":"my.app.GenericError","error_code":404}}
}
```

You can't specify different class discriminators in subclasses of a sealed base class. Only hierarchies with distinct, non-overlapping subclasses can define their own discriminators.

When both specify a discriminator, `@JsonClassDiscriminator` takes priority over the one in the `Json` configuration.

Set class discriminator output mode

Use the `JsonBuilder.classDiscriminatorMode` property to control how class discriminators are added to your JSON output. By default, the discriminator is only added for polymorphic types, which is useful when working with polymorphic class hierarchies.

To adjust this behavior, set the `ClassDiscriminatorMode` property to one of these options:

- `POLYMORPHIC` : (default) adds the class discriminator only for polymorphic types.
- `ALL_JSON_OBJECTS` : adds the class discriminator to all JSON objects, wherever possible.
- `NONE` : omits the class discriminator entirely.

Here's an example with the `ClassDiscriminatorMode` property set to `NONE` :

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Configures a Json instance to omit the class discriminator from the output
val format = Json { classDiscriminatorMode = ClassDiscriminatorMode.NONE }

@Serializable
sealed class Project {
    abstract val name: String
}

@Serializable
class OwnedProject(override val name: String, val owner: String) : Project()

fun main() {
    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")
    // Serializes without a discriminator
    println(format.encodeToString(data))
    // {"name":"kotlinx.coroutines","owner":"kotlin"}
}
```

Without the discriminator, the Kotlin serialization library can't deserialize the output back into the appropriate type.

Pretty printing

By default, `Json` produces a compact, single-line output.

You can add indentations and line breaks for better readability by enabling pretty printing in the output. To do so, set the `prettyPrint` property to `true` in a `Json` instance:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Creates a custom Json format
val format = Json { prettyPrint = true }

@Serializable
data class Project(val name: String, val language: String)

fun main() {
    val data = Project("kotlinx.serialization", "Kotlin")

    // Prints the JSON output with line breaks and indentations
    println(format.encodeToString(data))
}
```

This example prints the following result:

```
{
    "name": "kotlinx.serialization",
    "language": "Kotlin"
}
```

You can use the `prettyPrintIndent` option to customize the indentation used in pretty-printed JSON.

For example, you can replace the default four spaces with any allowed whitespace characters, such as `\t` or `\n`.

Customize JSON deserialization

Kotlin's `Json` parser offers several settings that allow you to customize the parsing and deserialization of JSON data.

Ignore unknown keys

When working with JSON data from third-party services or other dynamic sources, new properties may be added to the JSON objects over time.

By default, unknown keys (the property names in the JSON input) result in an error during deserialization. To prevent this, set the `ignoreUnknownKeys` property to `true` in a `Json` instance:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Creates a Json instance to ignore unknown keys
val format = Json { ignoreUnknownKeys = true }

@Serializable
data class Project(val name: String)

fun main() {
    val data = format.decodeFromString<Project>("""
        {"name":"kotlinx.serialization","language":"Kotlin"}
    """)
    // The language key is ignored because it's not in the Project class
    println(data)
    // Project(name=kotlinx.serialization)
}
```

Experimental

Ignore unknown keys for specific classes

Instead of enabling `ignoreUnknownKeys` for all classes, you can ignore unknown keys only for specific classes by using the `@JsonIgnoreUnknownKeys` annotation. This allows you to keep strict deserialization by default while allowing leniency only where needed.

The `@JsonIgnoreUnknownKeys` annotation is Experimental. To opt in, use the `@OptIn(ExperimentalSerializationApi::class)` annotation or the compiler option `-opt-in=kotlinx.serialization.ExperimentalSerializationApi`:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@OptIn(ExperimentalSerializationApi::class)
@Serializable
// Unknown properties in Outer are ignored during deserialization
@JsonIgnoreUnknownKeys
data class Outer(val a: Int, val inner: Inner)

@Serializable
data class Inner(val x: String)

fun main() {
    val outer = Json.decodeFromString<Outer>(
        """{"a":1,"inner":{"x":"value"},"unknownKey":42}"""
    )
    println(outer)
    // Outer(a=1, inner=Inner(x=value))

    // Throws an exception
    // unknownKey inside inner is NOT ignored because Inner is not annotated
    println(
        Json.decodeFromString<Outer>(
            """{"a":1,"inner":{"x":"value","unknownKey":"unexpected"}}"""
        )
    )
}
```

In this example, `Inner` throws a `SerializationException` for unknown keys because it isn't annotated with `@JsonIgnoreUnknownKeys`.

Coerce input values

When working with JSON data from third-party services or other dynamic sources, the format can evolve. This may cause exceptions during decoding when actual values don't match the expected types.

The default `Json` implementation is strict about input types. To relax this restriction, set the `coerceInputValues` property to `true` in a `Json` instance:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

val format = Json { coerceInputValues = true }

@Serializable
data class Project(val name: String, val language: String = "Kotlin")

fun main() {
    val data = format.decodeFromString<Project>("""
        {"name":"kotlinx.serialization","language":null}
    """)

    // Coerces the invalid null value for language to its default value
    println(data)
    // Project(name=kotlinx.serialization, language=Kotlin)
}
```

The `coerceInputValues` property only affects decoding. It treats certain invalid input values as if the corresponding property were missing. Currently, it applies to:

- `null` inputs for non-nullable types
- unknown values for enums

> This list may be expanded in future versions, making `Json` instances with this property even more permissive by replacing invalid values with defaults or `null`.

If a value is missing, it's replaced with a default property value if one exists.

For enums, the value is replaced with `null` only if:

- No default is defined.
- The `explicitNulls` property is set to `false`.
- The property is nullable.

You can combine `coerceInputValues` with the `explicitNulls` property to handle invalid enum values:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

enum class Color { BLACK, WHITE }

@Serializable
data class Brush(val foreground: Color = Color.BLACK, val background: Color?)

val json = Json {
  coerceInputValues = true
  explicitNulls = false
}

fun main() {

    // Coerces the unknown foreground value to its default and background to null
    val brush = json.decodeFromString<Brush>("""{"foreground":"pink", "background":"purple"}""")
    println(brush)
    // Brush(foreground=BLACK, background=null)
}
```

Allow trailing commas

To allow trailing commas in JSON input, set the `allowTrailingComma` property to `true`:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Allows trailing commas in JSON objects and arrays
val format = Json { allowTrailingComma = true }

fun main() {
    val numbers = format.decodeFromString<List<Int>>(
        """
            [1, 2, 3,]
        """
    )
    println(numbers)
    // [1, 2, 3]
}
```

Allow comments in JSON

Use the `allowComments` property to allow comments in JSON input. When this property is enabled, the parser accepts the following comment forms in the input:

- `//` line comments that end at a newline `\n`
- `/* */` block comments

Here's an example:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Allows comments in JSON input
val format = Json { allowComments = true }

fun main() {
    val numbers = format.decodeFromString<List<Int>>(
        """
            [
                // first element
                1,
                /* second element */
                2
            ]
        """
    )
    println(numbers)
    // [1, 2]
}
```

Lenient parsing

By default, the `Json` parser enforces strict JSON rules to ensure compliance with the RFC-8259 specification, which requires keys and string literals to be quoted.

To relax these restrictions, set the `isLenient` property to `true` in a `Json` instance:

```kotlin
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

val format = Json { isLenient = true }

enum class Status { SUPPORTED }

@Serializable
data class Project(val name: String, val status: Status, val votes: Int)

fun main() {
    // Decodes a JSON string with lenient parsing
    // Lenient parsing allows unquoted keys, string, and enum values
    val data = format.decodeFromString<Project>("""
        {
            name   : kotlinx.serialization,
            status : SUPPORTED,
            votes  : "9000"
        }
    """)
    println(data)
    // Project(name=kotlinx.serialization, status=SUPPORTED, votes=9000)
}
```

Customize name mapping between JSON and Kotlin

Some JSON data may not perfectly align with Kotlin's naming conventions or expected formats. To address these challenges, the Kotlin serialization library provides several tools to manage naming discrepancies, handle multiple JSON property names, and ensure consistent naming strategies across serialized data.

Accept alternative JSON property names for a single Kotlin property

When JSON property names change between schema versions, you can use the `@SerialName` annotation to rename a JSON property.

However, this prevents decoding data that still uses previous property names. To accept alternative JSON property names for a single property, use the `@JsonNames` annotation:

```kotlin
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
// Maps both name and title JSON properties to the name property
data class Project(@JsonNames("title") val name: String)

fun main() {
    val project = Json.decodeFromString<Project>("""{"name":"kotlinx.serialization"}""")
    println(project)
    // Project(name=kotlinx.serialization)

    val oldProject = Json.decodeFromString<Project>("""{"title":"kotlinx.coroutines"}""")
    // Both name and title Json properties correspond to name property
    println(oldProject)
    // Project(name=kotlinx.coroutines)
}
```

The `useAlternativeNames` property in `JsonBuilder` enables the `@JsonNames` annotation. This property is set to `true` by default and allows `Json` to recognize and decode multiple names for a single property.

If you aren't using `@JsonNames` and want to improve performance, especially when skipping many unknown properties with `ignoreUnknownKeys`, you can set this property to `false`.

Decode enums in a case-insensitive manner

Kotlin's naming conventions recommend naming enum values in uppercase with underscores or in upper camel case. By default, `Json` uses the exact names of Kotlin enum constants when decoding.

However, JSON data from external sources might use lowercase or mixed-case names. To handle such cases, configure the `Json` instance to decode enum values case-insensitively with the `JsonBuilder.decodeEnumsCaseInsensitive` property:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Configures a Json instance to decode enum values in a case-insensitive way
val format = Json { decodeEnumsCaseInsensitive = true }

enum class Cases { VALUE_A, @JsonNames("Alternative") VALUE_B }

@Serializable
data class CasesList(val cases: List<Cases>)

fun main() {
    // Decodes enum values regardless of their case, including alternative names
    println(format.decodeFromString<CasesList>("""{"cases":["value_A", "alternative"]}"""))
    // CasesList(cases=[VALUE_A, VALUE_B])
}
```

This property applies to both serial names and alternative names specified with the `@JsonNames` annotation, ensuring that all values are decoded successfully. This property doesn't affect encoding.

Apply a global naming strategy

When property names in JSON input differ from those in Kotlin, you can specify the name of each property explicitly using the `@SerialName` annotation. However, when migrating from other frameworks or a legacy codebase, you might need to transform every serial name in the same way.

For these scenarios, you can specify a global naming strategy using the JsonBuilder.namingStrategy property in a `Json` instance. The Kotlin serialization library provides built-in strategies, such as JsonNamingStrategy.SnakeCase:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
data class Project(val projectName: String, val projectOwner: String)

// Configures a Json instance to apply SnakeCase naming strategy
val format = Json { namingStrategy = JsonNamingStrategy.SnakeCase }

fun main() {
    val project = format.decodeFromString<Project>("""{"project_name":"kotlinx.coroutines",
"project_owner":"Kotlin"}""")
    // Serializes and deserializes as if all serial names are transformed from camel case to snake
case
    println(format.encodeToString(project.copy(projectName = "kotlinx.serialization")))
    // {"project_name":"kotlinx.serialization","project_owner":"Kotlin"}
}
```

When using a global naming strategy with `JsonNamingStrategy`, keep the following in mind:

- *The transformation applies to all properties*, whether the serial name is derived from the property name or explicitly defined with the `@SerialName` annotation. You can't exclude a property from transformation by specifying a serial name. To keep specific names unchanged during serialization, use the `@JsonNames` annotation instead.
- If a transformed name conflicts with other transformed property names or with alternative names specified by the `@JsonNames` annotation, deserialization fails with an exception.
- Global naming strategies are implicit. It makes it difficult to determine the serialized names by looking at the class definition. This can make tasks like **Find Usages**, and **Rename** in IDEs, or full-text searches using tools like `grep`, more difficult, potentially increasing the risk of bugs and maintenance costs.

Given these factors, consider the trade-offs carefully before implementing global naming strategies in your application.

What's next

- Explore advanced JSON element handling to manipulate and work with JSON data before it's parsed or serialized.
- Discover how to transform JSON during serialization and deserialization for more control over your data.

JSON elements

The Kotlin serialization library also supports working with JSON at a structural level. You can use the `JsonElement` API, to inspect, modify, and build JSON structures directly before converting them into a Kotlin type or a string.

`JsonElement` has three direct subtypes that represent the core JSON structures:

- `JsonPrimitive` handles primitive JSON elements such as strings, numbers, booleans, and `null`. The `null` value is represented by a special subclass of `JsonPrimitive`, called `JsonNull`. Each `JsonPrimitive` stores a string representation of its value, which you can access through its `JsonPrimitive.content` property.
- `JsonArray` represents a JSON array. It's a Kotlin `List` of `JsonElement` items.
- `JsonObject` represents a JSON object. It's a Kotlin `Map` with `String` keys and `JsonElement` values.

Parse strings to JSON elements

You can parse a string into a `JsonElement` to work with the JSON structure before converting it into a Kotlin type or a string.

Use the `Json.parseToJsonElement()` function to parse the input into a JSON element tree without decoding or deserializing it:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

fun main() {
    val element = Json.parseToJsonElement("""
        {"name":"kotlinx.serialization","language":"Kotlin"}
    """)
    // JsonElement.toString() gives you a valid JSON string
    println(element)
    // {"name":"kotlinx.serialization","language":"Kotlin"}
}
```

Access the contents of JSON elements

You can access the contents of a JSON element directly through the extension properties of the `JsonElement` API. These extension properties cast the element to a specific subtype and throw an `IllegalArgumentException` if the element doesn't have the expected JSON structure.

The available extension properties are:

- `jsonPrimitive` returns a `JsonPrimitive`.
- `jsonArray` returns a `JsonArray`.
- `jsonObject` returns a `JsonObject`.

Similarly, `JsonPrimitive` has extension properties for parsing the value as Kotlin primitive types, such as `int`, `intOrNull`, `long`, and `longOrNull`.

Here's an example of how you can use these extension properties when processing JSON data with a known structure:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

fun main() {
    val element = Json.parseToJsonElement("""
        {
            "name": "kotlinx.serialization",
            "forks": [{"votes": 42}, {"votes": 9000}, {}]
        }
    """)
    val sum = element
        // Accesses the forks key from the JsonObject
        .jsonObject["forks"]!!

        // Accesses the value as a JsonArray and sums the votes values from each JsonObject as Int
        .jsonArray.sumOf { it.jsonObject["votes"]?.jsonPrimitive?.int ?: 0 }
    println(sum)
    // 9042
}
```

If you don't know the JSON structure in advance, you can check the element type and handle each `JsonElement` subtype explicitly. For example, you can use a helper function with a `when` expression:

```
fun processElement(element: JsonElement): String = when (element) {
    is JsonObject -> "JsonObject with keys: ${element.keys}"
    is JsonArray -> "JsonArray with ${element.size} elements"
    is JsonPrimitive -> "JsonPrimitive with content: ${element.content}"
}
```

Create JSON elements

You can create instances of specific `JsonElement` subtypes directly.

To create a `JsonPrimitive`, use the `JsonPrimitive()` function:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

fun main() {
    // Creates JsonPrimitive values from different Kotlin primitives
    val number = JsonPrimitive(42)
    val text = JsonPrimitive("kotlinx.serialization")

    println(number)
    // 42
    println(text)
    // "kotlinx.serialization"
}
```

You can create `JsonArray` and `JsonObject` elements either by directly calling their constructors or by using the builder functions:

- To create a `JsonArray` from a `List`, use `JsonArray()`, or use the `buildJsonArray()` builder function.
- To create a `JsonObject` from a `Map`, use `JsonObject()`, or use the `buildJsonObject()` builder function.

The builder functions provide a DSL similar to Kotlin's standard library collection builders with JSON-specific overloads and inner builder functions.

Here's an example that highlights the key features of the JSON builder DSLs:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

fun main() {
    val element = buildJsonObject {
        // Adds a simple key-value pair to the JsonObject
        put("name", "kotlinx.serialization")
        // Adds a nested JsonObject under the owner key
        putJsonObject("owner") {
            put("name", "kotlin")
        }
        // Adds a JsonArray with multiple JsonObjects
        putJsonArray("forks") {
            // Adds a JsonObject to the JsonArray
            addJsonObject {
                put("votes", 42)
            }
            addJsonObject {
                put("votes", 9000)
            }
        }
    }
    // Prints the resulting JSON string
    println(element)
    // {"name":"kotlinx.serialization","owner":{"name":"kotlin"},"forks":[{"votes":42},
{"votes":9000}]}
}
```

Encode literal JSON content

While the JSON specification doesn't restrict the size or precision of numbers, serializing numbers of arbitrary size with the `JsonPrimitive()` function might lead to some issues.

For example, if you use `Double` for large numbers, the value might get truncated and you lose precision. If you use Kotlin/JVM `BigDecimal`, the value stays precise, but `JsonPrimitive()` encodes the value as a string rather than as a number:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import java.math.BigDecimal

val format = Json { prettyPrint = true }

fun main() {
    val pi = BigDecimal("3.14159265358979323846264338327 9")

    // Converts the BigDecimal to a Double, causing potential truncation
    val piJsonDouble = JsonPrimitive(pi.toDouble())
    // Converts the BigDecimal to a String, preserving the precision but treating it as a string in
JSON
    val piJsonString = JsonPrimitive(pi.toString())

    val piObject = buildJsonObject {
        put("pi_double", piJsonDouble)
        put("pi_string", piJsonString)
    }

    println(format.encodeToString(piObject))
    // "pi_double": 3.141592653589793,
    // "pi_string": "3.14159265358979323846264338327 9"
}
```

In this example, even though `pi` is defined as a number with 30 decimal places, the resulting JSON doesn't preserve that precision. The `Double` value is truncated to 15 decimal places, and the `String` value is wrapped in quotes, so it becomes a JSON string rather than a number.

To avoid these issues, you can encode an arbitrary unquoted value, such as the string value of `pi` in this example, using the `JsonUnquotedLiteral()` function:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import java.math.BigDecimal

val format = Json { prettyPrint = true }

fun main() {
    val pi = BigDecimal("3.14159265358979323846264338327 9")

    // Encodes the raw JSON content using JsonUnquotedLiteral()
    @OptIn(ExperimentalSerializationApi::class)
    val piJsonLiteral = JsonUnquotedLiteral(pi.toString())

    // Converts to Double and String
    val piJsonDouble = JsonPrimitive(pi.toDouble())
    val piJsonString = JsonPrimitive(pi.toString())

    val piObject = buildJsonObject {
        put("pi_literal", piJsonLiteral)
        put("pi_double", piJsonDouble)
        put("pi_string", piJsonString)
    }

    // pi_literal now accurately matches the value defined
    println(format.encodeToString(piObject))
    // "pi_literal": 3.14159265358979323846264338327 9,
    // "pi_double": 3.141592653589793,
    // "pi_string": "3.14159265358979323846264338327 9"
}
```

To decode `pi` back to a `BigDecimal`, extract the string content of the `JsonPrimitive`:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import java.math.BigDecimal

fun main() {
    val piObjectJson = """
          {
              "pi_literal": 3.14159265358979323846264338327 9
          }
      """.trimIndent()

    // Decodes the JSON string into a JsonObject
    val piObject: JsonObject = Json.decodeFromString(piObjectJson)

    // Extracts the string content from the JsonPrimitive
    val piJsonLiteral = piObject["pi_literal"]!!.jsonPrimitive.content

    // Converts the string to a BigDecimal
    val pi = BigDecimal(piJsonLiteral)
    // Prints the decoded value of pi, preserving all 30 decimal places
    println(pi)
    // 3.14159265358979323846264338327 9
}
```



This example uses a `JsonPrimitive` for simplicity. For more reusable approaches, see Json Transformations.

JSON null literal

To avoid creating an inconsistent state, you can't encode the string `"null"` with the `JsonUnquotedLiteral()` function. If you try to do so, an exception is thrown:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@OptIn(ExperimentalSerializationApi::class)
fun main() {
    JsonUnquotedLiteral("null")
    // Exception in thread "main" kotlinx.serialization.json.internal.JsonEncodingException
}
```

To represent a JSON `null` literal value, use `JsonNull`:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

fun main() {
    val possiblyNull = JsonNull

    println(possiblyNull)
    // null
}
```

Decode Json elements

To decode an instance of the `JsonElement` class into a serializable object, use the `Json.decodeFromJsonElement()` function:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
data class Project(val name: String, val language: String)

fun main() {
    val element = buildJsonObject {
        put("name", "kotlinx.serialization")
        put("language", "Kotlin")
    }

    // Decodes the JsonElement into a Project object
    val data = Json.decodeFromJsonElement<Project>(element)
    println(data)
    // Project(name=kotlinx.serialization, language=Kotlin)
}
```

What's next

- Discover how to transform JSON during serialization and deserialization for more control over your data.
- Learn how to serialize classes and how to modify the default behavior of the `@Serializable` annotation.

Transform JSON structure

To control the structure and content of the JSON generated during serialization, you can create custom serializers. For smaller adjustments, such as wrapping values or unwrapping arrays, the `JsonTransformingSerializer` class provides a simpler way to modify JSON by working directly with the JSON element tree instead of using the `Encoder` or `Decoder` manually.

> These sections are based on concepts explained in Create and use serializers. If you're not familiar with custom serializers, we recommend reading that page first.

The `JsonTransformingSerializer` is a JSON-specific abstract serializer that implements the `KSerializer` interface. It provides the `transformSerialize()` and `transformDeserialize()` functions that you can override to adjust the JSON element tree before serialization or deserialization.

In addition to transforming JSON structures, you can also use `JsonContentPolymorphicSerializer` to select the appropriate polymorphic class based on the JSON content.

Modify JSON structure

You can adjust the JSON structure by transforming the JSON element tree. The following examples demonstrate common use cases, such as wrapping or unwrapping arrays and omitting specific properties.

Wrap a single object in an array during deserialization

Some APIs return a single JSON object for one item and a JSON array for multiple items. To deserialize both cases into a `List`:

1. Create a subclass of `JsonTransformingSerializer` and specify a serializer in its constructor. To use the standard conversion logic, pass the default serializer for the target type, such as `ListSerializer()` for lists.
2. Override the `transformDeserialize()` function.

Here's an example:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.builtins.*
import kotlinx.serialization.json.*

@Serializable
data class Project(
    val name: String,
    // Specifies UserListSerializer to handle the serialization of the users property
    @Serializable(with = UserListSerializer::class)
    val users: List<User>
)

@Serializable
data class User(val name: String)

// Creates a serializer that transforms the results of the default List serializer
object UserListSerializer : JsonTransformingSerializer<List<User>>(ListSerializer(User.serializer()))
{
    override fun transformDeserialize(element: JsonElement): JsonElement =
        // If the element is not a JsonArray, wraps it into a single-element array
        if (element !is JsonArray) JsonArray(listOf(element)) else element
}

fun main() {
    println(Json.decodeFromString<Project>("""
        {"name":"kotlinx.serialization","users":{"name":"kotlin"}}
    """))
    // Project(name=kotlinx.serialization, users=[User(name=kotlin)])

    println(Json.decodeFromString<Project>("""
        {"name":"kotlinx.serialization","users":[{"name":"kotlin"},{"name":"jetbrains"}]}
    """))
    // Project(name=kotlinx.serialization, users=[User(name=kotlin), User(name=jetbrains)])
}
```

Unwrap a single-element array during serialization

To unwrap a single-element list into a single JSON object during serialization, override the `transformSerialize()` function:

```kotlin
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.builtins.*
import kotlinx.serialization.json.*

@Serializable
data class Project(
    val name: String,
    // Specifies UserListSerializer to handle serialization of the users property
    @Serializable(with = UserListSerializer::class)
    val users: List<User>
)

@Serializable
data class User(val name: String)

// Creates a serializer that transforms the results of the default List serializer
object UserListSerializer : JsonTransformingSerializer<List<User>>(ListSerializer(User.serializer()))
{

    override fun transformSerialize(element: JsonElement): JsonElement {
        require(element is JsonArray)
        // Unwraps single-element lists into a single JSON object
        return element.singleOrNull() ?: element
    }
}

fun main() {
    val data = Project("kotlinx.serialization", listOf(User("kotlin")))
    println(Json.encodeToString(data))
    // {"name":"kotlinx.serialization","users":{"name":"kotlin"}}
}
```

Omit specific properties during serialization

If you can't specify a default value, but still want to omit a property when it has a specific value, use `JsonTransformingSerializer`:

1. Create a subclass of `JsonTransformingSerializer` and specify a serializer in its constructor.
2. Override the `transformSerialize()` function.

Here's an example where the `Project` class has a `language` property, which is omitted when its value is `"Kotlin"`:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.builtins.*
import kotlinx.serialization.json.*

@Serializable
class Project(val name: String, val language: String)

// Creates a custom serializer that omits the language property if it's equal to "Kotlin"
object ProjectSerializer : JsonTransformingSerializer<Project>(Project.serializer()) {
    override fun transformSerialize(element: JsonElement): JsonElement =
        // Omits the language property if its value is "Kotlin"
        JsonObject(element.jsonObject.filterNot {
            (k, v) -> k == "language" && v.jsonPrimitive.content == "Kotlin"
        })
}

fun main() {
    val data = Project("kotlinx.serialization", "Kotlin")

    // Uses the default serializer
    println(Json.encodeToString(data))
    // {"name":"kotlinx.serialization","language":"Kotlin"}

    // Applies the custom serializer to omit the language property
    println(Json.encodeToString(ProjectSerializer, data))
    // {"name":"kotlinx.serialization"}
}
```

When serializing an object directly, you need to explicitly pass the custom serializer to the `encodeToString()` function to ensure that the custom serialization logic is applied. For more information, see Pass serializers manually.

Select the appropriate polymorphic class based on the JSON content

In polymorphic serialization, JSON often contains a dedicated *class discriminator* property that identifies the concrete subtype during deserialization.

If the JSON input doesn't contain a class discriminator, you can use `JsonContentPolymorphicSerializer` to infer the type from the structure of the JSON. This serializer lets you override the `selectDeserializer()` function to choose the correct deserializer based on the JSON content.

Here's an example where all values share a common base type with a `name` property:

```
@Serializable
abstract class Project {
    abstract val name: String
}

@Serializable
data class BasicProject(override val name: String): Project()

@Serializable
data class OwnedProject(override val name: String, val owner: String) : Project()
```

In this example, the serializer chooses which subtype to use based on the JSON content, so you don't need a `sealed` class for the class hierarchy.

To distinguish between `BasicProject` and `OwnedProject`, override the `selectDeserializer()` function. You can use this function to check whether the JSON object contains the `owner` key, and return the corresponding serializer:

```
// Creates a custom serializer that selects deserializer based on the presence of "owner"
object ProjectSerializer : JsonContentPolymorphicSerializer<Project>(Project::class) {
    override fun selectDeserializer(element: JsonElement) = when {
        // Selects the OwnedProject serializer if the JSON object contains an "owner" key
        "owner" in element.jsonObject -> OwnedProject.serializer()
        else -> BasicProject.serializer()
    }
}
```

When you use this serializer to serialize data, Kotlin serialization uses the serializer of the value's actual runtime type. This could be the serializer specified in a `SerializersModule` or the default serializer:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.builtins.*
import kotlinx.serialization.json.*


@Serializable
abstract class Project {
    abstract val name: String
}

@Serializable
data class BasicProject(override val name: String): Project()

@Serializable
data class OwnedProject(override val name: String, val owner: String) : Project()

// Creates a custom serializer that selects deserializer based on the presence of "owner"
object ProjectSerializer : JsonContentPolymorphicSerializer<Project>(Project::class) {
    override fun selectDeserializer(element: JsonElement) = when {
        // Selects the OwnedProject serializer if the JSON object contains an "owner" key
        "owner" in element.jsonObject -> OwnedProject.serializer()
        else -> BasicProject.serializer()
    }
}

fun main() {
    val data = listOf(
        OwnedProject("kotlinx.serialization", "kotlin"),
        BasicProject("example")
    )
    // No class discriminator in the JSON output
    val string = Json.encodeToString(ListSerializer(ProjectSerializer), data)

    println(string)
    // [{"name":"kotlinx.serialization","owner":"kotlin"},{"name":"example"}]

    println(Json.decodeFromString(ListSerializer(ProjectSerializer), string))
    // [OwnedProject(name=kotlinx.serialization, owner=kotlin), BasicProject(name=example)]
}
```

Experimental

Add custom behavior to the default serializer

You can add custom behavior to the default serializer that Kotlin serialization generates, by using the default serializer as a delegate.

To do so, annotate a serializable class with the Experimental `@KeepGeneratedSerializer` and use the automatically generated `generatedSerializer()` as the base serializer in your custom `JsonTransformingSerializer`.

Here's an example that updates the JSON structure during deserialization by combining multiple input properties into a single `name` property expected by the target class:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@OptIn(ExperimentalSerializationApi::class)
@KeepGeneratedSerializer
@Serializable(with = UserNameSerializer::class)
// Defines a type with a name property
data class User(val name: String)

// Adds custom logic to the default serializer to combine input properties during deserialization
object UserNameSerializer : JsonTransformingSerializer<User>(User.generatedSerializer()) {
    override fun transformDeserialize(element: JsonElement): JsonElement {
        val jsonObject = element.jsonObject
        val first = jsonObject["firstName"]?.jsonPrimitive?.content
        val last = jsonObject["lastName"]?.jsonPrimitive?.content

        // Combines input properties into the name property
        // if the input doesn't match the expected structure
        return if (first ≠ null && last ≠ null) {
            JsonObject(mapOf("name" to JsonPrimitive("$first $last")))
        } else {
            jsonObject
        }
    }
}

fun main() {
    // Deserializes JSON where the name property is split across multiple input properties
   val fromExternalData = Json.decodeFromString<User>(
        """{"firstName":"John","lastName":"Smith"}"""
    )
    println(fromExternalData)
    // User(name=John Smith)

    // Deserializes JSON where the name property matches the expected structure
    val fromInternalData = Json.decodeFromString<User>(
        """{"name":"John Smith"}"""
    )
    println(fromInternalData)
    // User(name=John Smith)
}
```

Experimental

Implement custom serialization logic in JSON

If the transformation functions provided by `JsonTransformingSerializer` or `JsonContentPolymorphicSerializer` aren't enough, you can implement custom serialization logic by defining your own `KSerializer` class.

You can get full control over how values are serialized and deserialized by overriding the `serialize()` and `deserialize()` functions directly.

When you implement custom serialization logic for JSON, you can cast `Encoder` to `JsonEncoder` and `Decoder` to `JsonDecoder` to call the JSON-specific functions `decodeJsonElement()` and `encodeToJsonElement()`. These functions allow you to retrieve JSON elements from the value the decoder is currently handling, or insert JSON elements into it.

Both `JsonDecoder` and `JsonEncoder` expose a `json` property that gives access to the active `Json` instance, which controls how values are encoded and decoded. Through that instance, you can use `encodeToJsonElement()` and `decodeFromJsonElement()` to convert between `JsonElement` instances and Kotlin objects.

Using these APIs, you can implement two-stage conversions:

- Decode the input into a `JsonElement` first and then convert that element into a Kotlin value.
- Convert a Kotlin value into a `JsonElement` first and then encode that element with the encoder.

Let's look at an example of a custom `KSerializer` that fully controls how values of a `Response` type are encoded and decoded in JSON. This serializer encodes an `Ok` response directly as a JSON value and an `Error` response as a JSON object that contains the error message:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.builtins.*
import kotlinx.serialization.json.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.encoding.*

// Defines a sealed class for API responses
@Serializable(with = ResponseSerializer::class)
sealed class Response<out T> {
    data class Ok<out T>(val data: T) : Response<T>()
    data class Error(val message: String) : Response<Nothing>()
}

// Implements custom serialization logic for Response
class ResponseSerializer<T>(
    private val dataSerializer: KSerializer<T>
) : KSerializer<Response<T>> {
    override val descriptor: SerialDescriptor = buildClassSerialDescriptor("Response") {
        element("Ok", dataSerializer.descriptor)
        element("Error", buildClassSerialDescriptor("Error") {
            element<String>("message")
        })
    }
    // Deserializes a Response value from JSON
    override fun deserialize(decoder: Decoder): Response<T> {
        // Ensures that the decoder is a JsonDecoder
        require(decoder is JsonDecoder)

        // Decodes the input into a JsonElement
        val element = decoder.decodeJsonElement()

        // Converts the JsonElement into the corresponding Response value
        return if (element is JsonObject && "error" in element) {
            Response.Error(element["error"]!!.jsonPrimitive.content)
        } else {
            Response.Ok(
                decoder.json.decodeFromJsonElement(dataSerializer, element)
            )
        }
    }

    // Serializes a Response value to JSON
    override fun serialize(encoder: Encoder, value: Response<T>) {
        // Ensures that the encoder is a JsonEncoder
        require(encoder is JsonEncoder)

        // Converts the Response value into a JsonElement
        val element = when (value) {
            is Response.Ok ->
                encoder.json.encodeToJsonElement(dataSerializer, value.data)
            is Response.Error ->
                buildJsonObject { put("error", value.message) }
        }

        // Encodes the JsonElement using the encoder
        encoder.encodeJsonElement(element)
    }
}

@Serializable
data class Project(val name: String)
```

```
fun main() {
    val responses = listOf(
        Response.Ok(Project("kotlinx.serialization")),
        Response.Error("Not found")
    )

    val json = Json.encodeToString(responses)
    println(json)
    // [{"name":"kotlinx.serialization"},{"error":"Not found"}]

    println(Json.decodeFromString<List<Response<Project>>>(json))
    // [Ok(data=Project(name=kotlinx.serialization)), Error(message=Not found)]
}
```

Preserve unknown JSON attributes

A common use case for a custom JSON-specific serializer is preserving JSON properties from the input that your serializable class doesn't define. By default, these properties are ignored during deserialization.

To preserve these JSON properties, implement a custom JSON-specific serializer that collects all properties not defined in the target class into a dedicated `JsonObject` field during deserialization. This allows you to preserve these properties in the serializable class without modifying the original JSON structure.

Here's an example:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.builtins.*
import kotlinx.serialization.json.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.encoding.*

data class UnknownProject(val name: String, val details: JsonObject)

object UnknownProjectSerializer : KSerializer<UnknownProject> {
    override val descriptor: SerialDescriptor = buildClassSerialDescriptor("UnknownProject") {
        element<String>("name")
        element<JsonElement>("details")
    }

    override fun deserialize(decoder: Decoder): UnknownProject {
        // Ensures the decoder is JSON-specific
        val jsonInput = decoder as? JsonDecoder ?: error("Can be deserialized only by JSON")

        // Reads the entire content as JSON
        val json = jsonInput.decodeJsonElement().jsonObject

        // Extracts and removes the name property
        val name = json.getValue("name").jsonPrimitive.content
        val details = json.toMutableMap()
        details.remove("name")
        return UnknownProject(name, JsonObject(details))
    }

    override fun serialize(encoder: Encoder, value: UnknownProject) {
        error("Serialization is not supported")
    }
}

fun main() {
    // Deserializes JSON with properties not defined in the serializable class into UnknownProject
    println(Json.decodeFromString(UnknownProjectSerializer, """
{"type":"unknown","name":"example","maintainer":"Unknown","license":"Apache 2.0"}"""))
    // UnknownProject(name=example, details={"type":"unknown","maintainer":"Unknown","license":"Apache
2.0"})

}
```

In this example, the preserved JSON properties remain at the same level within the input JSON object as the properties defined in the serializable class.

What's next

- Learn how to serialize polymorphic classes and handle objects of various types within a shared hierarchy.
- Discover other serialization formats, such as CBOR and ProtoBuf.

Experimental

Serialize JSON with I/O sources

The Kotlin serialization library provides APIs for working with JVM streams and `kotlinx-io` or Okio sources and sinks.

You can use these APIs to serialize and deserialize JSON directly from I/O sources without creating intermediate strings. These APIs use UTF-8 encoding and throw `SerializationException` for invalid JSON data and `IOException` for I/O failures.

When working with I/O resources, it's important to close them properly to prevent resource leaks. You can do this with the `.use()` function, which closes the resource automatically when the operation completes.

Serialize JSON to JVM output streams

Use the `.encodeToStream()` extension function to serialize JSON directly to a JVM `OutputStream`:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import java.io.FileOutputStream

@Serializable
data class Project(val name: String, val stars: Int)

fun main() {
    val project = Project("kotlinx.serialization", 9000)

    // Creates an OutputStream for the project.json file
    FileOutputStream("project.json").use { output ->

        // Serializes the project instance into the OutputStream
        Json.encodeToStream(project, output)
    }
}
```

In this example, the JSON representation of `Project` is serialized into the `project.json` file.

Deserialize JSON from JVM input streams

To deserialize JSON directly from a JVM `InputStream`, use the `.decodeFromStream()` extension function:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import java.io.FileInputStream

@Serializable
data class Project(val name: String, val stars: Int)

fun main() {
    // Opens an InputStream
    FileInputStream("project.json").use { input ->

        // Deserializes the JSON contents of the InputStream into a Project instance
        val project = Json.decodeFromStream<Project>(input)

        // Prints the deserialized Project instance
        println(project)
    }
}
```

In this example, the JSON contents of the input stream are deserialized into a single `Project` instance.

If your input contains multiple JSON objects in a top-level JSON array or as whitespace-separated objects, you can use `.decodeToSequence()` to process the elements lazily. This lets you handle each value as it is parsed, for example:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import java.io.FileInputStream

@Serializable
data class Project(val name: String, val stars: Int)

fun main() {
    // Opens an InputStream for the projects.json file containing a JSON array of Project objects
    FileInputStream("projects.json").use { input ->

        // Lazily deserializes each Project from the InputStream
        val projects = Json.decodeToSequence<Project>(input)

        // Processes elements one by one
        for (project in projects) {
            println(project)
        }
    }
}
```

You can iterate through sequences returned by `.decodeToSequence()` only once because they are tied to the underlying stream.

Accessing a sequence may result in an exception if its underlying stream is closed before the sequence is completely evaluated.

JSON serialization with kotlinx-io and Okio

In addition to JVM streams, you can work with JSON using I/O types, such as `kotlinx.io.Sink` and `kotlinx.io.Source` from the `kotlinx-io` library (currently in Alpha), and `okio.BufferedSink` and `okio.BufferedSource` from the Okio library.

You can use the following `Json` extension functions to read and write JSON directly through these I/O types:

- `.encodeToSink()` and `.encodeToBufferedSink()` to write JSON to a `kotlinx.io.Sink` or `okio.BufferedSink`.
- `.decodeFromSource()` and `.decodeFromBufferedSource()` to read a single JSON value from a `kotlinx.io.Source` or `okio.BufferedSource`.
- `.decodeSourceToSequence()` and `.decodeBufferedSourceToSequence()` to lazily decode multiple JSON values as a `Sequence<T>`.

The next sections cover examples using `kotlinx-io` types with these APIs.
You can use Okio types similarly with their corresponding `okio.BufferedSink` and `okio.BufferedSource` APIs.

Add dependencies for kotlinx-io and Okio

To use the extension functions with `kotlinx-io` or Okio types, add the corresponding dependencies:

Add dependencies for `kotlinx-io`

Gradle

```
// build.gradle(.kts)
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-json-io:1.11.0")
    implementation("org.jetbrains.kotlinx:kotlinx-io-core:0.9.1")
}
```

Maven

```
<!-- pom.xml -->
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlinx</groupId>
        <artifactId>kotlinx-serialization-json-io</artifactId>
        <version>1.11.0</version>
    </dependency>
    <dependency>
        <groupId>org.jetbrains.kotlinx</groupId>
        <artifactId>kotlinx-io-core</artifactId>
        <version>0.9.1</version>
    </dependency>
</dependencies>
```

Add dependencies for Okio

Gradle

```
// build.gradle(.kts)
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-json-okio:1.11.0")
    implementation("com.squareup.okio:okio:3.16.2")
}
```

Maven

```
<!-- pom.xml -->
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlinx</groupId>
        <artifactId>kotlinx-serialization-json-okio</artifactId>
        <version>1.11.0</version>
    </dependency>
    <dependency>
        <groupId>com.squareup.okio</groupId>
        <artifactId>okio</artifactId>
        <version>3.16.2</version>
    </dependency>
</dependencies>
```

Serialize JSON to Sinks

To serialize JSON to a `Sink`, use the `.encodeToSink()` function:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Imports declarations for kotlinx-io types and JSON I/O support
import kotlinx.serialization.json.io.*
import kotlinx.io.*
import kotlinx.io.files.*

@Serializable
data class Project(val name: String, val stars: Int)

@OptIn(ExperimentalSerializationApi::class)
fun main() {
    val project = Project("kotlinx.serialization", 9000)

    // Creates a Sink for the project.json file
    val path = Path("project.json")
    SystemFileSystem.sink(path).buffered().use { sink: Sink →

        // Serializes the Project instance directly into a Sink
        Json.encodeToSink(project, sink)
    }
}
```

Deserialize JSON from Sources

To deserialize JSON from a `Source`, use the `.decodeFromSource()` function:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Imports declarations for kotlinx-io types and JSON I/O support
import kotlinx.serialization.json.io.*
import kotlinx.io.*
import kotlinx.io.files.*

@Serializable
data class Project(val name: String, val stars: Int)

@OptIn(ExperimentalSerializationApi::class)
fun main() {
    // Opens a Source for the project.json file
    val path = Path("project.json")
    SystemFileSystem.source(path).buffered().use { source: Source →

        // Deserializes a Project instance directly from a Source
        val project = Json.decodeFromSource<Project>(source)

        println(project)
    }
}
```

If your input contains a large JSON array or multiple top-level JSON objects, you can turn a `Source` into a lazily decoded `Sequence<T>` with the `.decodeSourceToSequence()` function.

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Imports declarations for kotlinx-io types and JSON I/O support
import kotlinx.serialization.json.io.*
import kotlinx.io.*
import kotlinx.io.files.*

@Serializable
data class Project(val name: String, val stars: Int)

@OptIn(ExperimentalSerializationApi::class)
fun main() {
    // Opens a Source for the projects.json file containing multiple JSON objects
    val path = Path("projects.json")
    SystemFileSystem.source(path).buffered().use { source: Source ->

        // Lazily deserializes each Project as it is read from the Source
        val projects: Sequence<Project> = Json.decodeSourceToSequence(source)

        for (project in projects) {
            println(project)
        }
    }
}
```

You can iterate through sequences returned by `.decodeSourceToSequence()` only once because they are tied to the underlying `Source`.

Accessing a sequence may result in an exception if its underlying source is closed before the sequence is completely evaluated.

What's next

- Learn how to customize Json instances to address different use cases for serialization and deserialization.
- Explore advanced JSON element handling to manipulate and work with JSON data before it's parsed or serialized.
- Discover how to transform JSON during serialization and deserialization for more control over your data.

Serialize polymorphic classes

Polymorphism allows working with objects of different types through a common interface or base class. Kotlin serialization supports subtype polymorphism, allowing different types to be serialized through a common declared supertype.

For an overview of how interfaces and inheritance work in Kotlin, see Interfaces and Inheritance.

Kotlin serialization is *static* by default. It uses the *static type* of the serialized value, which is the declared type of the variable, to determine which properties are encoded.

This means that only the properties defined in that static type are serialized, even if the value is an instance of a subclass at runtime.

Here's an example that uses a polymorphic class hierarchy:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
open class Project(val name: String)

class OwnedProject(name: String, val owner: String) : Project(name)

fun main() {
    // Uses Project as the declared static type
    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")

    // Serializes only properties defined in the static type
    println(Json.encodeToString(data))
    // {"name":"kotlinx.coroutines"}
}
```

In this example, even though the runtime value is the `OwnedProject` subclass, only the properties of the declared `Project` static type are serialized.

To serialize values in a polymorphic class hierarchy, Kotlin serialization provides two approaches:

- Closed polymorphism, where a `sealed` base class or interface ensures that all subclasses are known at compile time.
- Open polymorphism, where you specify subclasses explicitly for an `open` or `abstract` base class.

Serialize closed polymorphic classes

A class hierarchy is considered to use closed polymorphism when all possible subclasses are guaranteed to be known at compile time.

To provide this guarantee, use a `sealed class` or `sealed interface` as the base type. Mark it as `@Serializable`, and mark all subclasses with `@Serializable`:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Defines a sealed class as the base type
@Serializable
sealed class Project {
    abstract val name: String
}

// Marks the subclass as serializable
@Serializable
class OwnedProject(override val name: String, val owner: String) : Project()

// Serializes data using the base type as the static type
fun main() {
    // Uses the base type as the static type
    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")

    // A type property is added to identify the serialized subclass
    println(Json.encodeToString(data))
    // {"type":"OwnedProject","name":"kotlinx.coroutines","owner":"kotlin"}
}
```

By default, the `type` property in the JSON output acts as the class discriminator and identifies the serialized subclass. This property is only included if you use the base class as the static type during serialization.

> You can also configure JSON to use a different key name for the class discriminator. For more information, see the Specify class discriminator for polymorphism section.

Here's an example where a subclass is the static type, so the `type` property is omitted:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
sealed class Project {
    abstract val name: String
}

@Serializable
class OwnedProject(override val name: String, val owner: String) : Project()

fun main() {
    // The inferred static type is OwnedProject
    val data = OwnedProject("kotlinx.coroutines", "kotlin")

    // The type property is omitted because the static type is OwnedProject
    println(Json.encodeToString(data))
    // {"name":"kotlinx.coroutines","owner":"kotlin"}
}
```

You can explicitly specify the base type when serializing objects to ensure the `type` property is included in the output. To do so, pass the base type as a type argument to the `encodeToString()` function:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*


@Serializable
sealed class Project {
    abstract val name: String
}

@Serializable
class OwnedProject(override val name: String, val owner: String) : Project()

fun main() {
    // The inferred static type is OwnedProject
    val data = OwnedProject("kotlinx.coroutines", "kotlin")

    // Specifies the base type explicitly to make use of polymorphism
    println(Json.encodeToString<Project>(data))
    // {"type":"OwnedProject","name":"kotlinx.coroutines","owner":"kotlin"}
}
```

Define custom serial names for subclasses

By default, Kotlin serialization uses the fully qualified class name as the class discriminator value for polymorphic subclasses in both closed and open polymorphic hierarchies. This value can change if you refactor the class.

To keep the class discriminator value stable, define a custom serial name with the `@SerialName` annotation.

> Each serializable class must have a unique `@SerialName`. Don't reuse the same name for different classes, even across separate polymorphic hierarchies.
>
> If you reuse the same name within the same hierarchy, encoding or decoding functions throw an `IllegalStateException`. Reusing the same name across separate hierarchies can lead to subtle serialization issues.

Use `@SerialName` to define a stable identifier for a subclass that doesn't depend on its source code:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
sealed class Project {
    abstract val name: String
}

// Assigns a custom serial name to the subclass
@Serializable
@SerialName("owned")
class OwnedProject(override val name: String, val owner: String) : Project()

fun main() {
    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")
    println(Json.encodeToString(data))
    // {"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"}
}
```

Base class properties with backing fields

In a polymorphic class hierarchy, the base class can define properties with backing fields that are serialized together with subclass properties, for example:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
sealed class Project {
    abstract val name: String
    var status = "open"
}

@Serializable
@SerialName("owned")
class OwnedProject(override val name: String, val owner: String) : Project()

fun main() {
    // Configures a Json instance to encode default values
    val json = Json { encodeDefaults = true }

    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")
    // Serializes base class properties together with subclass properties
    println(json.encodeToString(data))
    // {"type":"owned","status":"open","name":"kotlinx.coroutines","owner":"kotlin"}
}
```

Serialize objects in polymorphic class hierarchies

A polymorphic class hierarchy can have objects as subclasses. During serialization, an object is treated similarly to a class without properties, and its class name is used as the value of the `type` property by default.

To include these objects in serialization, annotate them with `@Serializable`:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Defines a sealed base class
@Serializable
sealed class Response

// Defines an object subclass
@Serializable
object EmptyResponse : Response()

// Defines a class that extends Response
@Serializable
class TextResponse(val text: String) : Response()

// Serializes a list containing different subclasses
fun main() {
    val list = listOf(EmptyResponse, TextResponse("OK"))
    println(Json.encodeToString(list))
    // [{"type":"EmptyResponse"},{"type":"TextResponse","text":"OK"}]
}
```

Serialize open polymorphic classes

Kotlin serialization supports open polymorphism for `open` and `abstract` classes and interfaces. In this model, subclasses can be defined anywhere in the codebase, even in other modules. Since the compiler can't determine all subclasses at compile time, you must specify them explicitly.

Use open polymorphism when not all subclasses are known at compile time. To register all subclasses at runtime, you have to specify the base type and every subclass in a `SerializersModule` class:

1. Create a `SerializersModule` with the `SerializersModule()` builder function.
2. Inside the `SerializersModule`, use the `polymorphic()` function to specify the base type.
3. Inside the `polymorphic()` block, *register* each subclass using the `subclass()` function. Registering a subclass adds it to the set of subclasses associated with the base type, which makes it available for polymorphic serialization and deserialization.
4. Add the `SerializersModule` to a `Json` instance with the `serializersModule` property.
5. Optionally, use the `@SerialName` annotation to define a stable identifier for a subclass instead of using its fully qualified class name.

To include the `type` property, use the base type as the static type, for example:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.modules.*

// Defines a SerializersModule
val module = SerializersModule {

    // Specifies Project as the base type
    polymorphic(Project::class) {

        // Registers OwnedProject as a subclass of Project
        subclass(OwnedProject::class)
    }
}

// Adds the SerializersModule to a Json instance
val format = Json { serializersModule = module }

// Defines the base type used as the static type during serialization
@Serializable
abstract class Project {
    abstract val name: String
}

// Defines a serializable subclass of Project
@Serializable
@SerialName("owned")
class OwnedProject(override val name: String, val owner: String) : Project()

fun main() {
    // Uses Project as the static type to make use of polymorphism
    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")

    println(format.encodeToString(data))
    // {"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"}
}
```

This configuration produces the same JSON structure as the example in the Serialize closed polymorphic classes section, but it supports `open` and `abstract` classes.

> This example works only on the JVM because of `serializer()` function restrictions. In Kotlin/JS and Kotlin/Native, use an explicit serializer: `format.encodeToString(PolymorphicSerializer(Project::class), data)`.
>
> You can keep track of this issue on GitHub.

Serialize interfaces in open polymorphic hierarchies

You can't annotate interfaces with `@Serializable`. Kotlin serialization treats interfaces as polymorphic, using `PolymorphicSerializer` as their serializer by default.

To use an interface as the base type in open polymorphic serialization, mark the classes that implement it with `@Serializable` and register them in a `SerializersModule`:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.modules.*

// Defines a SerializersModule
val module = SerializersModule {
    polymorphic(Project::class) {
        subclass(OwnedProject::class)
    }
}

val format = Json { serializersModule = module }

// Defines an interface for polymorphic serialization
interface Project {
    val name: String
}

// OwnedProject implements the Project interface
@Serializable
@SerialName("owned")
class OwnedProject(override val name: String, val owner: String) : Project

fun main() {
    // Uses the interface for serialization
    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")
    println(format.encodeToString(data))
    // {"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"}
}
```

> In Kotlin/JS and Kotlin/Native, you must explicitly specify the serializer with `format.encodeToString(PolymorphicSerializer(Project::class), data)` because of the platform's limited reflection capabilities.

You can also use an interface as a property in a serializable class. In this case, Kotlin serialization also applies `PolymorphicSerializer` to that property:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.modules.*

val module = SerializersModule {
    polymorphic(Project::class) {
        subclass(OwnedProject::class)
    }
}

val format = Json { serializersModule = module }

interface Project {
    val name: String
}

@Serializable
@SerialName("owned")
class OwnedProject(override val name: String, val owner: String) : Project

// Defines a serializable class with an interface property
@Serializable
class Data(val project: Project)

fun main() {
    val data = Data(OwnedProject("kotlinx.coroutines", "kotlin"))
    println(format.encodeToString(data))
    // {"project":{"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"}}
}
```

Select the base type of a polymorphic hierarchy

Kotlin serialization is fully static. It determines the base type of a polymorphic hierarchy at compile time. You can select any base type, including `Any`, as long as you configure it explicitly.

To do so:

1. Specify the base type and its subclasses in a `SerializersModule`.
2. If the base type doesn't have its own serializer, pass `PolymorphicSerializer` for that type when calling the `encodeToString()` function.

Here's an example:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.modules.*

val module = SerializersModule {
    // Registers OwnedProject for the Any base type
    polymorphic(Any::class) {
        subclass(OwnedProject::class)
    }
}

val format = Json { serializersModule = module }

@Serializable
abstract class Project {
    abstract val name: String
}

@Serializable
@SerialName("owned")
class OwnedProject(override val name: String, val owner: String) : Project()

fun main() {
    // Uses Any as the static type
    val data: Any = OwnedProject("kotlinx.coroutines", "kotlin")

    // Specifies a PolymorphicSerializer for the Any base type
    println(format.encodeToString(PolymorphicSerializer(Any::class), data))
    // {"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"}
}
```

If you don't provide a `PolymorphicSerializer`, the following exception is thrown:

```
Exception in thread "main" kotlinx.serialization.SerializationException: Serializer for class 'Any' is
not found.
Please ensure that class is marked as '@Serializable' and that the serialization compiler plugin is
applied.
```

You can also use multiple base types for the same subclass, as long as you register the subclass for each base type explicitly.

To avoid repeating `subclass()` calls for each base type, you can define a helper function that registers the subclasses and use it in each `polymorphic()` block.

Here's an example that uses both `Project` and `BaseProject` as base types:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.modules.*

val module = SerializersModule {
    // Defines a helper function that registers subclasses
    fun PolymorphicModuleBuilder<Project>.registerProjectSubclasses() {
        subclass(OwnedProject::class)
    }

    // Registers the same subclass for each base type Project and  BaseProject
    polymorphic(BaseProject::class) { registerProjectSubclasses() }
    polymorphic(Project::class) { registerProjectSubclasses() }
}

val format = Json { serializersModule = module }

interface BaseProject {
    val name: String
}

interface Project : BaseProject

@Serializable
@SerialName("owned")
class OwnedProject(override val name: String, val owner: String) : Project

@Serializable
class Data(
    val project: Project,
    val baseProject: BaseProject
)

fun main() {
    val project = OwnedProject("kotlinx.coroutines", "kotlin")
    val data = Data(project, project)
    println(format.encodeToString(data))
    // {"project":{"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"},"baseProject":
{"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"}}
}
```

Serialize properties polymorphically

Interfaces and abstract classes use polymorphic serialization by default.

To serialize a property with a non-serializable type, such as `Any`, or when the property's type is an `open` class, annotate the property with `@Polymorphic`.

This applies `PolymorphicSerializer` to the property.

Here's an example:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.modules.*

val module = SerializersModule {
    polymorphic(Any::class) {
        subclass(OwnedProject::class)
    }
}

val format = Json { serializersModule = module }

interface Project {
    val name: String
}

@Serializable
@SerialName("owned")
class OwnedProject(override val name: String, val owner: String) : Project

@Serializable
class Data(
    // Applies PolymorphicSerializer to the property
    @Polymorphic
    val project: Any
)

fun main() {
    val data = Data(OwnedProject("kotlinx.coroutines", "kotlin"))
    println(format.encodeToString(data))
    // {"project":{"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"}}
}
```

Serialize open classes polymorphically

To serialize an `open` class polymorphically, annotate it with both `@Serializable` and `@Polymorphic`:

```kotlin
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.modules.*

val module = SerializersModule {
    polymorphic(Project::class) {
        subclass(OwnedProject::class)
    }
}

val format = Json { serializersModule = module }

// Applies PolymorphicSerializer to the open class
@Serializable
@Polymorphic
open class Project

@Serializable
@SerialName("owned")
class OwnedProject(val name: String, val owner: String) : Project()

@Serializable
class Data(val project: Project)

fun main() {
    val project = OwnedProject("kotlinx.coroutines", "kotlin")

    val data = Data(project)
    println(format.encodeToString(data))
    // {"project":{"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"}}
}
```

Experimental

Register implementations of a sealed class or interface

You can combine open and closed polymorphism in the same hierarchy, for example when the root of the hierarchy is an `interface` and various sub-hierarchies are represented as sealed classes. Instead of registering each sealed subclass individually, you can register all implementations of a sealed class or interface with the `subclassesOfSealed()` function.

This function also registers subclasses recursively when a subclass is itself a sealed serializable class. All subclasses of that sealed class or interface must be either concrete or sealed. Otherwise, an `IllegalArgumentException` is thrown.

Here's an example:

```kotlin
@file:OptIn(ExperimentalSerializationApi::class)

// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.modules.*

interface Base

@Serializable
sealed interface Sub: Base

@Serializable
class Sub1(val data: String): Sub

val module1 = SerializersModule {
    polymorphic(Base::class) {
        subclassesOfSealed(Sub.serializer())
    }
}

val format1 = Json { serializersModule = module1 }

val module2 = SerializersModule {
    polymorphic(Base::class) {
        // Uses the reified version of subclassesOfSealed() to specify the same sealed type
        subclassesOfSealed<Sub>()
    }
}

val format2 = Json { serializersModule = module2 }


fun main() {
    val data: Base = Sub1("kotlin")
    println(format1.encodeToString(data))
    // {"type":"Sub1","data":"kotlin"}

    println(format2.encodeToString(data))
    // {"type":"Sub1","data":"kotlin"}
}
```

Serialize generic subtypes in a polymorphic hierarchy

Kotlin serialization can't automatically determine the concrete type of a generic type parameter at runtime. As a result, it can't pick a serializer for that parameter without explicit configuration.

To provide this configuration for a generic polymorphic subtype, register the subtype in a `SerializersModule` with an explicit serializer. `PolymorphicSerializer(Any::class)` has the broadest scope, but you can use a more specific serializer when you know which concrete types the generic value can have.

> Sealed base types don't need this configuration because the compiler plugin can infer the subtype serializer when the base type provides the serializer for its type argument.

Here's an example:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.modules.*

@Serializable
abstract class Response<out T>

// Defines a generic polymorphic subtype
@Serializable
@SerialName("OkResponse")
data class OkResponse<out T>(val data: T) : Response<T>()

@Serializable
abstract class Project {
    abstract val name: String
}

// Defines a subtype used as a generic value
@Serializable
@SerialName("OwnedProject")
data class OwnedProject(override val name: String, val owner: String) : Project()

// Defines serializers for a polymorphic hierarchy with generic subtypes
val responseModule = SerializersModule {
    polymorphic(Response::class) {
        // Registers the generic subtype
        // with a serializer that specifies a PolymorphicSerializer
        subclass(OkResponse.serializer(PolymorphicSerializer(Any::class)))
    }
    polymorphic(Any::class) {
        // Registers the subtype used as the generic value
        subclass(OwnedProject::class)
    }
    polymorphic(Project::class) {
        // Registers the same subtype for the static base type
        subclass(OwnedProject::class)
    }
}

// Creates a Json instance with the registered serializers
val format = Json { serializersModule = responseModule }

fun main() {
    // Uses a generic polymorphic type with a concrete subtype
    val data: Response<Project> = OkResponse(OwnedProject("kotlinx.serialization", "kotlin"))

    val jsonString = format.encodeToString(data)
    println(jsonString)
    // {"type":"OkResponse","data":
{"type":"OwnedProject","name":"kotlinx.serialization","owner":"kotlin"}}

    val deserializedData = format.decodeFromString<Response<Project>>(jsonString)
    println(deserializedData)
    // OkResponse(data=OwnedProject(name=kotlinx.serialization, owner=kotlin))
}
```

In this example, `PolymorphicSerializer(Any::class)` allows the generic subtype `OkResponse` to be serialized with any value that's registered polymorphically as a subtype of `Any`.

Merge multiple `SerializersModule` instances

As an application grows and splits into multiple source code modules, managing all class hierarchies within a single `SerializersModule` can become difficult.

You can merge multiple `SerializersModule` instances using the `plus` operator, allowing them to be used together in the same `Json` format instance.

Here's an example:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.modules.*

@Serializable
abstract class Response<out T>

@Serializable
data class OkResponse<out T>(val data: T) : Response<T>()

val responseModule = SerializersModule {
    polymorphic(Response::class) {
        subclass(OkResponse.serializer(PolymorphicSerializer(Any::class)))
    }
}

@Serializable
abstract class Project {
   abstract val name: String
}

@Serializable
data class OwnedProject(override val name: String, val owner: String) : Project()


val projectModule = SerializersModule {
    fun PolymorphicModuleBuilder<Project>.registerProjectSubclasses() {
        subclass(OwnedProject::class)
    }
    polymorphic(Any::class) { registerProjectSubclasses() }
    polymorphic(Project::class) { registerProjectSubclasses() }
}
// Merges the SerializersModule instances from both hierarchies
val format = Json { serializersModule = projectModule + responseModule }

fun main() {
    val data: Response<Project> = OkResponse(OwnedProject("kotlinx.serialization", "kotlin"))
    val string = format.encodeToString(data)
    println(string)
    // {"type":"OkResponse","data":
{"type":"OwnedProject","name":"kotlinx.serialization","owner":"kotlin"}}

    println(format.decodeFromString<Response<Project>>(string))
    // OkResponse(data=OwnedProject(name=kotlinx.serialization, owner=kotlin))
}
```

To merge modules inside a `SerializersModule` block, use the `include()` function:

```
// Merges multiple SerializersModule instances using include()
val combinedModule = SerializersModule {
    include(projectModule)
    include(responseModule)
}
```

If you expose the `SerializersModule` of your library or shared module, consumers can merge it with their own `SerializersModule`.

Deserialize unknown polymorphic subtypes with a default deserializer

When deserializing polymorphic data, Kotlin serialization resolves the subtype from the `type` property. If the subtype isn't registered, deserialization fails with a `SerializationException`.

You can define a default deserializer to handle unregistered or unknown polymorphic subtypes.

Let's look at the following example where `Project` is the base type, `OwnedProject` is a registered subtype, and `BasicProject` represents unknown project subtypes:

```
@Serializable
abstract class Project {
    abstract val name: String
}

// Represents unknown project types
@Serializable
data class BasicProject(override val name: String, val type: String): Project()

@Serializable
@SerialName("OwnedProject")
data class OwnedProject(override val name: String, val owner: String) : Project()
```

To handle unknown polymorphic subtypes of `Project`, configure a default deserializer for the base type. Use the `defaultDeserializer()` function inside the `polymorphic()` block to define a fallback for these unregistered subtypes:

```
val module = SerializersModule {
    polymorphic(Project::class) {
        subclass(OwnedProject::class)
        defaultDeserializer { BasicProject.serializer() }
    }
}
```

You can use this `SerializersModule` configuration to deserialize both registered and unregistered subtypes:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.modules.*

@Serializable
abstract class Project {
    abstract val name: String
}

// Represents unknown project types
@Serializable
data class BasicProject(override val name: String, val type: String): Project()

@Serializable
@SerialName("OwnedProject")
data class OwnedProject(override val name: String, val owner: String) : Project()

// Registers a default deserializer for unknown Project subtypes
val module = SerializersModule {
    polymorphic(Project::class) {
        subclass(OwnedProject::class)
        defaultDeserializer { BasicProject.serializer() }
    }
}

val format = Json { serializersModule = module }

fun main() {
    // Deserializes both a known and an unknown Project subtype
    println(format.decodeFromString<List<Project>>("""
        [
            {"type":"unknown","name":"example"},
            {"type":"OwnedProject","name":"kotlinx.serialization","owner":"kotlin"}
        ]
    """))
    // [BasicProject(name=example, type=unknown), OwnedProject(name=kotlinx.serialization,
owner=kotlin)]
}
```

In this example, deserialization doesn't rely on the `type` field to differentiate subtypes and instead uses the plugin-generated serializer for `BasicProject`. This approach assumes that unknown subtypes follow a known structure.

For JSON, you can also configure the format to ignore unknown keys when unknown subtypes contain extra properties but still match the structure expected by the default deserializer. If unknown input varies in structure, use a custom serializer instead.

> For more flexible handling of JSON input during deserialization, see Modify JSON structure.

Serialize polymorphic types with a default serializer

You can serialize values of a polymorphic base type without registering every concrete subtype. Use this when you don't have access to the full type hierarchy, or when it changes a lot.

To do so:

1. Use the `polymorphicDefaultSerializer()` function in a `SerializersModule` block.

2. Specify a lambda in `polymorphicDefaultSerializer()` that returns a `SerializationStrategy` for the runtime value.

Let's look at an example with two private classes, `CatImpl` and `DogImpl`. To avoid raising their visibility, register a default serializer for `Animal` that selects a serializer based on the runtime type through public interfaces:

```
interface Animal

interface Cat : Animal {
    val catType: String
}

interface Dog : Animal {
    val dogType: String
}

private class CatImpl : Cat {
    override val catType: String = "Tabby"
}

private class DogImpl : Dog {
    override val dogType: String = "Husky"
}

object AnimalProvider {
    fun createCat(): Cat = CatImpl()
    fun createDog(): Dog = DogImpl()
}

// Registers a default serializer for unknown Animal subtypes
val module = SerializersModule {
    polymorphicDefaultSerializer(Animal::class) { instance ->
        @Suppress("UNCHECKED_CAST")
        // Determines the appropriate serializer using a when block
        when (instance) {
            is Cat -> CatSerializer as SerializationStrategy<Animal>
            is Dog -> DogSerializer as SerializationStrategy<Animal>
            else -> null
        }
    }
}
```

Define serializers for `Cat` and `Dog`, then create a `Json` instance that uses the `SerializersModule` with the `serializersModule` property to enable polymorphic serialization of `Animal` values:

```
// Imports declarations from the serialization library
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.modules.*

interface Animal

interface Cat : Animal {
    val catType: String
}

interface Dog : Animal {
    val dogType: String
}

private class CatImpl : Cat {
    override val catType: String = "Tabby"
}

private class DogImpl : Dog {
    override val dogType: String = "Husky"
}

object AnimalProvider {
    fun createCat(): Cat = CatImpl()
    fun createDog(): Dog = DogImpl()
}

// Registers a default serializer for unknown Animal subtypes
val module = SerializersModule {
    polymorphicDefaultSerializer(Animal::class) { instance ->
        @Suppress("UNCHECKED_CAST")
        // Determines the appropriate serializer using a when block based on the runtime value
        when (instance) {
            is Cat -> CatSerializer as SerializationStrategy<Animal>
            is Dog -> DogSerializer as SerializationStrategy<Animal>
            else -> null
        }
    }
}

// Defines custom serializers
object CatSerializer : SerializationStrategy<Cat> {
    override val descriptor = buildClassSerialDescriptor("Cat") {
        element<String>("catType")
    }

    override fun serialize(encoder: Encoder, value: Cat) {
        encoder.encodeStructure(descriptor) {
            encodeStringElement(descriptor, 0, value.catType)
        }
    }
}

object DogSerializer : SerializationStrategy<Dog> {
    override val descriptor = buildClassSerialDescriptor("Dog") {
        element<String>("dogType")
    }

    override fun serialize(encoder: Encoder, value: Dog) {
        encoder.encodeStructure(descriptor) {
```

```
            encodeStringElement(descriptor, 0, value.dogType)
        }
    }
}

val format = Json { serializersModule = module }

fun main() {
    // Serializes an instance of Cat
    println(format.encodeToString<Animal>(AnimalProvider.createCat()))
    // {"type":"Cat","catType":"Tabby"}
}
```

What's next

- Learn how to obtain generated serializers, create custom serializers, and apply serializers in Create and use serializers.

Create and use serializers

A serializer defines the structure of a Kotlin type in its serialized form, while format implementations such as `Json` control how that structure is encoded.

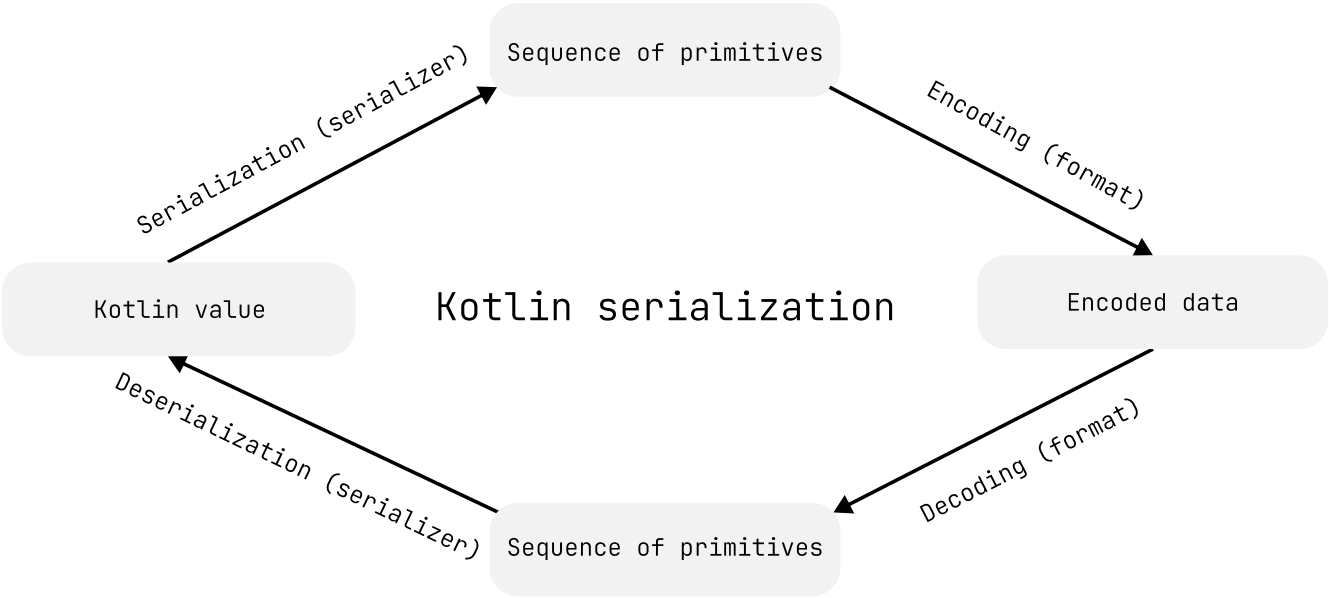


Diagram where a Kotlin value is serialized by a serializer into a sequence of primitives, encoded by a format into encoded data, decoded back into a sequence of primitives, and deserialized by a serializer into a Kotlin value

Serializers define serialization and deserialization strategies for a type through the `KSerializer` interface. Kotlin serialization provides serializers for built-in types, collections, and more.

You can use these serializers to serialize values or inspect the structure of their serialized form, while custom serializers let you define that structure yourself.

Obtain serializers

When you annotate a class with `@Serializable`, the Kotlin serialization plugin generates a `KSerializer` for that class.

To retrieve this automatically generated serializer, call the generated `.serializer()` function. You can use a serializer directly with functions such as `Json.encodeToString()`, or access its `descriptor` property to inspect the structure of the type's serialized form.

Here's an example that defines a `Color` class with a single integer property and inspects the structure of its serialized form:

```
import kotlinx.serialization.*

@Serializable
data class Color(val rgb: Int)

fun main() {
    // Retrieves the generated serializer for the Color class
    val colorSerializer: KSerializer<Color> = Color.serializer()

    println(colorSerializer.descriptor)
    // Color(rgb: kotlin.Int)
}
```

Built-in primitive types and `String` also provide serializers through the `.serializer()` function, such as `Int.serializer()` and `String.serializer()`.

You can use the top-level `serializer<T>()` function to obtain a serializer for any type, including parameterized ones:

```
import kotlinx.serialization.*

@Serializable
@SerialName("Color")
class Color(val rgb: Int)

fun main() {
    // Retrieves the serializer for the Map<String, Color>
    val stringToColorMapSerializer: KSerializer<Map<String, Color>> = serializer()

    // Prints: kotlin.collections.LinkedHashMap(PrimitiveDescriptor(kotlin.String), Color(rgb:
kotlin.Int))
    println(stringToColorMapSerializer.descriptor)
}
```

For generic classes, if you want to use the generated `.serializer()` function, provide one `KSerializer` argument for each type parameter:

```
import kotlinx.serialization.*

@Serializable
@SerialName("Color")
class Color(val rgb: Int)

@Serializable
@SerialName("Box")
class Box<T>(val contents: T)

fun main() {
    // Calls .serializer() using a KSerializer for the type parameter
    val boxedColorSerializer = Box.serializer(Color.serializer())

    println(boxedColorSerializer.descriptor)
    // Box(contents: Color)
}
```

Obtain serializers for collection types

Unlike classes annotated with `@Serializable`, collection types like `List<T>` don't have a generated `.serializer()` function.

To obtain a serializer for a collection type, create one with `ListSerializer()`, `SetSerializer()`, or `MapSerializer()` and specify serializers for the collection's type parameters.

Here's an example that uses the `ListSerializer()` function to create a serializer for `List<String>`:

```
import kotlinx.serialization.*
import kotlinx.serialization.builtins.*

fun main() {
    val stringListSerializer: KSerializer<List<String>> = ListSerializer(String.serializer())
    println(stringListSerializer.descriptor)
    // kotlin.collections.ArrayList(PrimitiveDescriptor(kotlin.String))
}
```

Create custom serializers

If you want more control over the structure of your serialized data, you can create a custom serializer. A custom serializer lets you define how a type is represented in its serialized form.

Like generated serializers, custom serializers define both serialization and deserialization for a type through the `KSerializer` interface. To support both, `KSerializer` extends `SerializationStrategy` and `DeserializationStrategy`.

> For JSON serialization, you can also modify the JSON output produced by an existing serializer without redefining its structure with `JsonTransformingSerializer`.

Create a custom primitive serializer

Use a primitive serializer to represent a class as a single primitive value, such as a string or integer.

To create a custom primitive serializer:

1. Create a custom serializer as an `object` that implements `KSerializer` for the serialized class:

```
object YourSerializer : KSerializer<Type>
```

2. Override the `descriptor` property to define the schema for the serialized data.

   Use the `PrimitiveSerialDescriptor(serialName, kind)` to define the structure of the serialized form as a single primitive value. Specify a **unique** `serialName`, such as a fully qualified name, and use a `PrimitiveKind` that matches the encoder and decoder functions used in the serializer:

```
override val descriptor: SerialDescriptor =
    PrimitiveSerialDescriptor("com.example.Type", PrimitiveKind.STRING)
```

   If the `descriptor` doesn't match the encoding and decoding functions, updates to `kotlinx.serialization` may cause the serializer to behave unpredictably in some formats.

3. Implement the `serialize()` function to define how values are converted to their serialized form. Choose an `Encoder` function that matches the `PrimitiveKind` in the `descriptor`:

```
override fun serialize(encoder: Encoder, value: Type) {
    val encodedValue: String = // convert value to a primitive representation
    encoder.encodeString(encodedValue)
}
```

4. Implement the `deserialize()` function to define how to convert the serialized data back into an instance of your class. The `Decoder` provides functions to read the data, such as `decodeString()`.

```
override fun deserialize(decoder: Decoder): Type {
    val decodedValue: String = decoder.decodeString()
    // Converts decodedValue back to Type
    return ...
}
```

5. Use the `@Serializable` annotation to specify a custom serializer for the class:

```
@Serializable(YourSerializer::class)
data class Type(val stringValue: String)
```

Here's an example for a custom primitive serializer that serializes `Color` as a hexadecimal string:

```
import kotlinx.serialization.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.json.*

// Creates the custom serializer for the Color class
object ColorAsStringSerializer : KSerializer<Color> {
    // Defines the schema for the serialized data as a single string
    override val descriptor: SerialDescriptor =
        // Specifies a unique name and a PrimitiveKind
        PrimitiveSerialDescriptor("my.app.Color", PrimitiveKind.STRING)

    // Defines how a Color value is serialized as a string
    override fun serialize(encoder: Encoder, value: Color) {
        // Converts the RGB value to a hexadecimal string
        val hexValue = value.rgb.toString(16).padStart(6, '0')
        // Encodes the serialized value as a string
        encoder.encodeString(hexValue)
    }

    // Defines how a Color value is deserialized from a string
    override fun deserialize(decoder: Decoder): Color {
        // Decodes the serialized string value
        val hexValue = decoder.decodeString()
        // Converts the decoded value back into a Color
        return Color(hexValue.toInt(16))
    }
}
// Specifies ColorAsStringSerializer as the custom serializer for the Color class
@Serializable(ColorAsStringSerializer::class)
data class Color(val rgb: Int)

fun main() {
    val color = Color(0x00FF00)
    // Serializes a Color value to JSON
    val jsonString = Json.encodeToString(color)

    println(jsonString)
    // "00ff00"

    // Deserializes the JSON string into a Color value
    val deserializedColor = Json.decodeFromString<Color>(jsonString)

    println(deserializedColor.rgb)
    // 65280
}
```

Serialize binary data as Base64 strings

You can use a custom primitive serializer to solve common serialization tasks beyond simple value conversion. One such task is representing binary data as a Base64 string.

Different APIs use different Base64 variants by default. Choose a Kotlin Base64 encoder that matches the expected variant, such as `Base64.Default` or `Base64.Mime`.

Here's an example that serializes a `ByteArray` as a Base64 string with `Base64.Default` in JSON format:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.encoding.Encoder
import kotlinx.serialization.encoding.Decoder
import kotlinx.serialization.descriptors.*
import kotlin.io.encoding.*

// Creates a custom primitive serializer,
// which represents ByteArray as a Base64 string
object ByteArrayAsBase64Serializer : KSerializer<ByteArray> {

    // Uses the default Base64 variant
    private val base64 = Base64.Default

    // Defines the serialized form as a single STRING primitive
    override val descriptor: SerialDescriptor
        get() = PrimitiveSerialDescriptor(
            "ByteArrayAsBase64Serializer",
            PrimitiveKind.STRING
        )

    // Encodes the ByteArray as a Base64 string
    override fun serialize(encoder: Encoder, value: ByteArray) {
        val base64Encoded = base64.encode(value)
        encoder.encodeString(base64Encoded)
    }

    // Decodes the Base64 string back into a ByteArray
    override fun deserialize(decoder: Decoder): ByteArray {
        val base64Decoded = decoder.decodeString()
        return base64.decode(base64Decoded)
    }
}

@Serializable
data class Value(
    // Specifies the custom serializer for this property
    @Serializable(ByteArrayAsBase64Serializer::class)
    val base64Input: ByteArray
) {

    // Implements value-based equality for ByteArray
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (javaClass ≠ other?.javaClass) return false
        other as Value
        return base64Input.contentEquals(other.base64Input)
    }

    // Computes hashCode based on array contents
    override fun hashCode(): Int {
        return base64Input.contentHashCode()
    }
}

fun main() {
    val string = "PNG_IMAGE_DATA"
    val value = Value(string.toByteArray())

    // Serializes Value to JSON
    val encoded = Json.encodeToString(value)

    println(encoded)
```

```
    // {"base64Input":"UE5HX0lNQUdFX0RBVEE="}

    // Deserializes JSON back into Value
    val decoded = Json.decodeFromString<Value>(encoded)

    println(decoded.base64Input.decodeToString())
    // PNG_IMAGE_DATA
}
```

Delegate serialization to another serializer

You can serialize a class as another type by delegating the serialization logic to the serializer for that type. For example, you can use this approach to serialize a class as a non-primitive type, such as an `IntArray`.

To delegate serialization, create a custom serializer that defines a property for the serializer of the delegated type:

1. Override the `descriptor` property to wrap the delegated serializer's `descriptor`.

   > When you delegate serialization, you can't use the original class `descriptor` or the delegated type's `descriptor` directly. Instead, create a new `descriptor` that reuses the delegated serializer's structure.

2. Override the `serialize()` function to convert an instance of your class to the delegated type and use `encoder.encodeSerializableValue()` function to encode it with the delegated serializer.
3. Override the `deserialize()` function to use the `decoder.decodeSerializableValue()` function with the delegated serializer to decode the value and convert it back into an instance of your class.

Here's an example that serializes a `Color` class as an `IntArray` by delegating the serialization logic to `IntArraySerializer`:

```
import kotlinx.serialization.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.builtins.IntArraySerializer
import kotlinx.serialization.json.*

// Creates a custom serializer that delegates to IntArraySerializer
class ColorIntArraySerializer : KSerializer<Color> {
    private val delegateSerializer = IntArraySerializer()
    override val descriptor = SerialDescriptor("my.app.Color", delegateSerializer.descriptor)

    // Delegates serialization logic to IntArraySerializer
    override fun serialize(encoder: Encoder, value: Color) {
        val data = intArrayOf(
            (value.rgb shr 16) and 0xFF,
            (value.rgb shr 8) and 0xFF,
            value.rgb and 0xFF
        )
        encoder.encodeSerializableValue(delegateSerializer, data)
    }

    // Delegates deserialization and converts IntArray back to Color
    override fun deserialize(decoder: Decoder): Color {
        val array = decoder.decodeSerializableValue(delegateSerializer)
        return Color((array[0] shl 16) or (array[1] shl 8) or array[2])
    }
}

@Serializable(ColorIntArraySerializer::class)
class Color(val rgb: Int)

fun main() {
    val green = Color(0x00ff00)

    println(Json.encodeToString(green))
    // [0,255,0]
}
```

While using the array representation isn't conventional in JSON, it can reduce the size of serialized data when used with a `ByteArray` and a binary format.

> For more information on how non-JSON serialization formats treat arrays, see Alternative and custom serialization formats.

Serialize classes with a surrogate class

You can use a *surrogate class*, a class that matches the serialized form of another class, when you want to:

- Change how a class is serialized without modifying the class itself.
- Avoid creating a composite serializer.
- Validate the serialized form before creating the original class.
- Handle cases where direct serialization doesn't fit the class's rules.

You can make surrogate classes `private`, and use an `init` block to enforce constraints on the serial representation of the class. You can also define a custom serial name to keep the serialized type name unchanged.

Let's look at an example that serializes `Color` as a JSON object with specific ranges for the `r`, `g`, and `b` properties:

```
// Defines a surrogate that matches the serialized form
@Serializable
@SerialName("Color")
private class ColorSurrogate(val r: Int, val g: Int, val b: Int) {
    init {
        // Enforces constraints on the serialized form
        require(r in 0..255 && g in 0..255 && b in 0..255)
    }
}
```

Similarly to delegating serialization to another serializer, the custom serializer converts the original class to another representation and wraps that representation's automatically generated `SerialDescriptor`.

> Surrogate classes define a `SerialDescriptor` with their own `serialName`. You can reuse the structure of the surrogate class's `SerialDescriptor`, but not its `serialName`, because each `SerialDescriptor` must have a unique `serialName`.

To retrieve the generated serializer for the surrogate class, use `ColorSurrogate.serializer()`:

```
object ColorSerializer : KSerializer<Color> {
    // The serialNames of descriptors must be unique
    override val descriptor: SerialDescriptor = SerialDescriptor("my.app.Color",
ColorSurrogate.serializer().descriptor)

    // Converts the original class to the surrogate representation
    override fun serialize(encoder: Encoder, value: Color) {
        val surrogate = ColorSurrogate((value.rgb shr 16) and 0xff, (value.rgb shr 8) and 0xff,
value.rgb and 0xff)
        encoder.encodeSerializableValue(ColorSurrogate.serializer(), surrogate)
    }

    // Converts the surrogate representation back to the original class
    override fun deserialize(decoder: Decoder): Color {
        val surrogate = decoder.decodeSerializableValue(ColorSurrogate.serializer())
        return Color((surrogate.r shl 16) or (surrogate.g shl 8) or surrogate.b)
    }
}
```

Finally, specify the custom serializer for the class:

```kotlin
import kotlinx.serialization.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.builtins.IntArraySerializer
import kotlinx.serialization.json.*

// Defines a private surrogate class with custom properties
@Serializable
@SerialName("Color")
private class ColorSurrogate(val r: Int, val g: Int, val b: Int) {
    init {
        // Enforces constraints on the serialized form
        require(r in 0..255 && g in 0..255 && b in 0..255)
    }
}

// Creates a custom serializer that converts to and from the surrogate
object ColorSerializer : KSerializer<Color> {
    // Defines a unique serialName for the wrapped SerialDescriptor
    override val descriptor: SerialDescriptor =
        SerialDescriptor("my.app.Color", ColorSurrogate.serializer().descriptor)

    // Converts the original class to the surrogate representation
    override fun serialize(encoder: Encoder, value: Color) {
        val surrogate = ColorSurrogate((value.rgb shr 16) and 0xff, (value.rgb shr 8) and 0xff,
value.rgb and 0xff)
        encoder.encodeSerializableValue(ColorSurrogate.serializer(), surrogate)
    }

    // Converts the surrogate representation back to the original class
    override fun deserialize(decoder: Decoder): Color {
        val surrogate = decoder.decodeSerializableValue(ColorSurrogate.serializer())
        return Color((surrogate.r shl 16) or (surrogate.g shl 8) or surrogate.b)
    }
}

// Specifies ColorSerializer as the custom serializer for the class
@Serializable(ColorSerializer::class)
class Color(val rgb: Int)

fun main() {
    val green = Color(0x00ff00)

    println(Json.encodeToString(green))
    // {"r":0,"g":255,"b":0}
}
```

Create a custom composite serializer

You can use composite serializers to represent complex data structures, such as classes with multiple properties.

Compared to using a surrogate class, a custom composite serializer lets you define the serialized structure of the original class directly without an additional conversion step. This can also improve performance in some cases, but requires writing more of the serialization logic manually.

To create a custom composite serializer:

1. Create a serializer as an `object` that implements `KSerializer` for the target class:

```kotlin
object YourSerializer : KSerializer<Type>
```

2. Override the `descriptor` property and define the serialization schema with the `buildClassSerialDescriptor()` function:

```
override val descriptor: SerialDescriptor =
    buildClassSerialDescriptor("com.example.Type") {
        // ...
    }
```

3. Specify each property in the `buildClassSerialDescriptor()` function with the `element()` function. The order of elements determines their indices, starting from `0`.

   The `SerialKind` of the serializer's `descriptor` determines what each `element()` represents. In a class `descriptor`, an `element()` represents a property, while in an enum `descriptor`, an `element()` represents constants such as `RED` or `GREEN`:

```
override val descriptor: SerialDescriptor =
    buildClassSerialDescriptor("com.example.Type") {
        element<Int>("first")
        element<Int>("second")
    }
```

4. Implement the `serialize()` function using the `encodeStructure()` DSL. Inside its block, the lambda receiver is a `CompositeEncoder`, which you use to call functions such as `encodeIntElement()` for each field, following the order defined in the `descriptor`:

```
override fun serialize(encoder: Encoder, value: Type) =
    encoder.encodeStructure(descriptor) {
        encodeIntElement(descriptor, 0, value.first)
        encodeIntElement(descriptor, 1, value.second)
    }
```

5. Implement the `deserialize()` function using the `decodeStructure()` DSL. Inside its block, the lambda receiver is a `CompositeDecoder`, which you use to call functions like `decodeIntElement()` to decode each property.

   Most formats allow data to be encoded in an arbitrary order, which can differ from the order of elements in the serializer's `descriptor`. Use the `decodeElementIndex()` function to identify which `element()` to decode. It returns `CompositeDecoder.DECODE_DONE` when no more elements are left, which you use to stop decoding the current structure:

```
override fun deserialize(decoder: Decoder): Type =
    decoder.decodeStructure(descriptor) {
        var first = 0
        var second = ""

        // Uses decodeElementIndex to ensure correct decoding regardless of order
        while (true) {
            when (val index = decodeElementIndex(descriptor)) {
                0 -> first = decodeStringElement(descriptor, 0)
                1 -> second = decodeIntElement(descriptor, 1)
                CompositeDecoder.DECODE_DONE -> break
                else -> error("Unexpected index: $index")
            }
        }

        Type(first, second)
    }
```

6. Use the `@Serializable(YourSerializer::class)` annotation to specify the custom serializer for the class.

> If you only need to transform or restructure data without manually handling each element, consider delegating serialization to another serializer or a surrogate class for a simpler approach.

Let's look at an example of how to serialize a `Color` class with multiple properties:

```
import kotlinx.serialization.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.json.*

// Creates a custom serializer for the Color class with multiple properties
object ColorAsObjectSerializer : KSerializer<Color> {
    // Defines the schema for the Color class
    override val descriptor: SerialDescriptor =
        buildClassSerialDescriptor("my.app.Color") {
            // Specifies each property with its type and name with the element() function
            element<Int>("r")
            element<Int>("g")
            element<Int>("b")
        }

    // Serializes the Color in the order specified in the descriptor
    override fun serialize(encoder: Encoder, value: Color) =
        encoder.encodeStructure(descriptor) {
            encodeIntElement(descriptor, 0, (value.rgb shr 16) and 0xff)
            encodeIntElement(descriptor, 1, (value.rgb shr 8) and 0xff)
            encodeIntElement(descriptor, 2, value.rgb and 0xff)
        }

    // Deserializes the data back into a Color object
    override fun deserialize(decoder: Decoder): Color =
        decoder.decodeStructure(descriptor) {
            // Temporary variables to hold the decoded values
            var r = -1
            var g = -1
            var b = -1
            // Uses decodeElementIndex() since element order may vary by format
            while (true) {
                when (val index = decodeElementIndex(descriptor)) {
                    0 -> r = decodeIntElement(descriptor, 0)
                    1 -> g = decodeIntElement(descriptor, 1)
                    2 -> b = decodeIntElement(descriptor, 2)
                    CompositeDecoder.DECODE_DONE -> break
                    else -> error("Unexpected index: $index")
                }
            }
            // Validates values and reconstructs Color
            require(r in 0..255 && g in 0..255 && b in 0..255)
            Color((r shl 16) or (g shl 8) or b)
        }
}

// Specifies the custom serializer for Color
@Serializable(ColorAsObjectSerializer::class)
data class Color(val rgb: Int)

fun main() {
    val color = Color(0x00ff00)
    val string = Json.encodeToString(color)

    println(string)
    // {"r":0,"g":255,"b":0}

    require(Json.decodeFromString<Color>(string) == color)
}
```

Encode default values in custom serializers

Plugin-generated serializers check whether the encoder needs to encode values that are equal to their defaults. For example, in JSON, this is controlled by the `encodeDefaults` property.

To achieve the same behavior in a custom serializer, use the `shouldEncodeElementDefault()` function with the serializer's `descriptor` and the index of the element you're encoding.

> If a property is annotated with `@EncodeDefault`, plugin-generated serializers don't call `shouldEncodeElementDefault()`.

Here's an example where a custom serializer encodes default `Color` values when `encodeDefaults` is enabled:

```
import kotlinx.serialization.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.json.*

// Creates a custom serializer for the Color class with multiple properties
object ColorAsObjectSerializer : KSerializer<Color> {
    // Defines the schema for the Color class
    override val descriptor: SerialDescriptor =
        buildClassSerialDescriptor("my.app.Color") {
            // Specifies each property with its type and name with the element() function
            element<Int>("r", isOptional = true)
            element<Int>("g", isOptional = true)
            element<Int>("b", isOptional = true)
        }

    override fun serialize(encoder: Encoder, value: Color) =
        encoder.encodeStructure(descriptor) {
            val r = (value.rgb shr 16) and 0xff
            val g = (value.rgb shr 8) and 0xff
            val b = value.rgb and 0xff

            // Encodes r if it differs from its default value,
            // or if the encoder needs to encode the first element's default value
            if (r != 0 || shouldEncodeElementDefault(descriptor, 0)) {
                encodeIntElement(descriptor, 0, r)
            }
            if (g != 255 || shouldEncodeElementDefault(descriptor, 1)) {
                encodeIntElement(descriptor, 1, g)
            }
            if (b != 0 || shouldEncodeElementDefault(descriptor, 2)) {
                encodeIntElement(descriptor, 2, b)
            }
        }

    // Deserializes the data back into a Color object
    override fun deserialize(decoder: Decoder): Color =
        decoder.decodeStructure(descriptor) {
            var r = 0
            var g = 255
            var b = 0

            while (true) {
                when (val index = decodeElementIndex(descriptor)) {
                    0 -> r = decodeIntElement(descriptor, 0)
                    1 -> g = decodeIntElement(descriptor, 1)
                    2 -> b = decodeIntElement(descriptor, 2)
                    CompositeDecoder.DECODE_DONE -> break
                    else -> error("Unexpected index: $index")
                }
            }
            require(r in 0..255 && g in 0..255 && b in 0..255)
            Color((r shl 16) or (g shl 8) or b)
        }
}

// Specifies the custom serializer for Color
@Serializable(ColorAsObjectSerializer::class)
data class Color(val rgb: Int = 0x00ff00)

fun main() {
    val color = Color()
    val stringWithDefaults = Json { encodeDefaults = true }.encodeToString(color)
```

```
    println(stringWithDefaults)
    // {"r":0,"g":255,"b":0}
}
```

Experimental

Optimize deserialization with sequential decoding

Some formats use a strictly ordered schema and can support sequential decoding. For these formats, you can optimize deserialization with the `decodeSequentially()` function. This function returns `true` if the current structure can be decoded in order. Handle that case separately to skip the more complex logic of decoding individual elements out of order.

Serializers generated by the `kotlinx.serialization` plugin use this optimization.

Here's an example that uses `decodeSequentially()` to optimize deserialization when possible:

```
import kotlinx.serialization.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.json.*


object ColorAsObjectSerializer : KSerializer<Color> {

    override val descriptor: SerialDescriptor =
        buildClassSerialDescriptor("my.app.Color") {
            element<Int>("r")
            element<Int>("g")
            element<Int>("b")
        }

    override fun serialize(encoder: Encoder, value: Color) =
        encoder.encodeStructure(descriptor) {
            encodeIntElement(descriptor, 0, (value.rgb shr 16) and 0xff)
            encodeIntElement(descriptor, 1, (value.rgb shr 8) and 0xff)
            encodeIntElement(descriptor, 2, value.rgb and 0xff)
        }

    override fun deserialize(decoder: Decoder): Color =
        decoder.decodeStructure(descriptor) {
            var r = -1
            var g = -1
            var b = -1
            // Decodes values directly in order if the format stores data sequentially
            @OptIn(ExperimentalSerializationApi::class)
            if (decodeSequentially()) {
                r = decodeIntElement(descriptor, 0)
                g = decodeIntElement(descriptor, 1)
                b = decodeIntElement(descriptor, 2)
            } else while (true) {
                // Ensures correct decoding for formats where elements may be unordered
                when (val index = decodeElementIndex(descriptor)) {
                    0 -> r = decodeIntElement(descriptor, 0)
                    1 -> g = decodeIntElement(descriptor, 1)
                    2 -> b = decodeIntElement(descriptor, 2)
                    CompositeDecoder.DECODE_DONE -> break
                    else -> error("Unexpected index: $index")
                }
            }
            require(r in 0..255 && g in 0..255 && b in 0..255)
            Color((r shl 16) or (g shl 8) or b)
        }
}

@Serializable(ColorAsObjectSerializer::class)
data class Color(val rgb: Int)

fun main() {
    val color = Color(0x00ff00)
    val string = Json.encodeToString(color)

    println(string)
    // {"r":0,"g":255,"b":0}

    require(Json.decodeFromString<Color>(string) == color)
}
```

Create a custom serializer for generic types

To create a custom serializer for a generic class, declare the serializer as a `class` instead of an `object`, with one `KSerializer` constructor parameter for each generic type parameter.

You can delegate the serialization logic for each type parameter to the corresponding `KSerializer`, so it's encoded according to its own serialization rules.

Let's look at an example using a generic `Box<T>` class:

```
import kotlinx.serialization.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.json.*

@Serializable(BoxSerializer::class)
data class Box<T>(val contents: T)

// Creates a custom serializer as a class for Box<T>
class BoxSerializer<T>(private val dataSerializer: KSerializer<T>) : KSerializer<Box<T>> {
    // Defines a unique serialName for the Box<T> descriptor
    override val descriptor: SerialDescriptor =
        SerialDescriptor("my.app.Box", dataSerializer.descriptor)

    // Delegates serialization and deserialization
    override fun serialize(encoder: Encoder, value: Box<T>) = dataSerializer.serialize(encoder,
value.contents)
    override fun deserialize(decoder: Decoder) = Box(dataSerializer.deserialize(decoder))
}

@Serializable
data class Project(val name: String)

fun main() {
    val box = Box(Project("kotlinx.serialization"))
    val string = Json.encodeToString(box)

    println(string)
    // {"name":"kotlinx.serialization"}

    println(Json.decodeFromString<Box<Project>>(string))
    // Box(contents=Project(name=kotlinx.serialization))
}
```

Experimental

Use a plugin-generated serializer together with a custom serializer

By default, the Kotlin serialization plugin doesn't generate a serializer if you specify a custom serializer with `@Serializable(YourSerializer::class)`.

You might still want to use the plugin-generated serializer, for example:

- Using the plugin-generated serializer as a fallback strategy.
- Inspecting the plugin-generated `descriptor` to access the default structure.
- Reusing default serialization behavior in subclasses that don't use a custom serializer.

You can keep the automatically generated serializer alongside your custom one by annotating the serializable class with `@KeepGeneratedSerializer`. To access the plugin-generated serializer, use the `.generatedSerializer()` function on the companion object of the serializable class.

This can also be useful when using a `JsonTransformingSerializer` to adjust the JSON structure and reuse the plugin-generated serializer for the default serialization logic.

Here's an example that uses both a custom serializer and the plugin-generated serializer:

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*

object ColorAsStringSerializer : KSerializer<Color> {
    override val descriptor: SerialDescriptor = PrimitiveSerialDescriptor("my.app.ColorAsString",
PrimitiveKind.STRING)

    override fun serialize(encoder: Encoder, value: Color) {
        val string = value.rgb.toString(16).padStart(6, '0')
        encoder.encodeString(string)
    }

    override fun deserialize(decoder: Decoder): Color {
        val string = decoder.decodeString()
        return Color(string.toInt(16))
    }
}

@OptIn(ExperimentalSerializationApi::class)
@KeepGeneratedSerializer
@Serializable(ColorAsStringSerializer::class)
class Color(val rgb: Int)

fun main() {
    val green = Color(0x00ff00)

    // Uses the custom serializer
    println(Json.encodeToString(green))
    // "00ff00"

    // Uses the plugin-generated serializer
    println(Json.encodeToString(Color.generatedSerializer(), green))
    // {"rgb":65280}

}
```

Apply serializers

You can apply custom serializers to your own classes and to third-party types. Third-party types, such as java.util.Date, can't be directly annotated with `@Serializable` because their source code can't be modified.

Pass a serializer manually

To serialize a type with a custom serializer, create the serializer and pass it explicitly to overloads of functions such as `Json.encodeToString()` and `Json.decodeFromString()`.

Here's an example that serializes `Date` values as the number of milliseconds since the Unix epoch:

```
import kotlinx.serialization.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.json.*
import java.util.Date
import java.text.SimpleDateFormat

// Can't use @Serializable on Date without access to its source code
object DateAsLongSerializer : KSerializer<Date> {
    override val descriptor: SerialDescriptor = PrimitiveSerialDescriptor("my.app.DateAsLong",
PrimitiveKind.LONG)
    override fun serialize(encoder: Encoder, value: Date) = encoder.encodeLong(value.time)
    override fun deserialize(decoder: Decoder): Date = Date(decoder.decodeLong())
}

fun main() {
    val kotlin10ReleaseDate = SimpleDateFormat("yyyy-MM-ddX").parse("2016-02-15+00")

    // Serializes Date as a Long in milliseconds
    println(Json.encodeToString(DateAsLongSerializer, kotlin10ReleaseDate))
    // 1455494400000
}
```

Specify a serializer on a property

When a type is used as a property in a serializable class, specify its custom serializer on that property with the `@Serializable` annotation:

```
import kotlinx.serialization.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.json.*
import java.util.Date
import java.text.SimpleDateFormat

object DateAsLongSerializer : KSerializer<Date> {
    override val descriptor: SerialDescriptor = PrimitiveSerialDescriptor("my.app.DateAsLong",
PrimitiveKind.LONG)
    override fun serialize(encoder: Encoder, value: Date) = encoder.encodeLong(value.time)
    override fun deserialize(decoder: Decoder): Date = Date(decoder.decodeLong())
}

@Serializable
class ProgrammingLanguage(
    val name: String,
    // Specifies the custom serializer for the Date property
    @Serializable(DateAsLongSerializer::class)
    val stableReleaseDate: Date
)

fun main() {
    val data = ProgrammingLanguage("Kotlin", SimpleDateFormat("yyyy-MM-ddX").parse("2016-02-15+00"))

    println(Json.encodeToString(data))
    // {"name":"Kotlin","stableReleaseDate":1455494400000}
}
```

Specify a serializer on a type

You can also apply the `@Serializable` annotation directly to a type. You can use this to specify a custom serializer for a type when it's used as a generic type argument, for example in `List<Date>`:

```
import kotlinx.serialization.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.json.*
import java.util.Date
import java.text.SimpleDateFormat

object DateAsLongSerializer : KSerializer<Date> {
    override val descriptor: SerialDescriptor = PrimitiveSerialDescriptor("my.app.DateAsLong",
PrimitiveKind.LONG)
    override fun serialize(encoder: Encoder, value: Date) = encoder.encodeLong(value.time)
    override fun deserialize(decoder: Decoder): Date = Date(decoder.decodeLong())
}

@Serializable
class ProgrammingLanguage(
    val name: String,
     // Specifies the custom serializer for Date as a generic type argument
    val releaseDates: List<@Serializable(DateAsLongSerializer::class) Date>
)

fun main() {
    val df = SimpleDateFormat("yyyy-MM-ddX")
    val data = ProgrammingLanguage("Kotlin", listOf(df.parse("2023-07-06+00"), df.parse("2023-04-
25+00"), df.parse("2022-12-28+00")))

   println(Json.encodeToString(data))
    // {"name":"Kotlin","releaseDates":[1688601600000,1682380800000,1672185600000]}
}
```

Specify a serializer for a file

To apply a serializer to all properties of a given type in a source file, add the `@UseSerializers` annotation at the beginning of the file:

```
@file:UseSerializers(DateAsLongSerializer::class)
```

This applies the `DateAsLongSerializer` to all instances of that type within the file, so you don't need to annotate each property separately.

Here's an example:

```
// Applies the custom serializer to all properties of that type in the file
@file:UseSerializers(DateAsLongSerializer::class)

import kotlinx.serialization.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.json.*
import java.util.Date
import java.text.SimpleDateFormat

object DateAsLongSerializer : KSerializer<Date> {
    override val descriptor: SerialDescriptor = PrimitiveSerialDescriptor("my.app.DateAsLong",
PrimitiveKind.LONG)
    override fun serialize(encoder: Encoder, value: Date) = encoder.encodeLong(value.time)
    override fun deserialize(decoder: Decoder): Date = Date(decoder.decodeLong())
}

// Uses the file-level serializer for the Date property
@Serializable
class ProgrammingLanguage(val name: String, val stableReleaseDate: Date)

fun main() {
    val data = ProgrammingLanguage("Kotlin", SimpleDateFormat("yyyy-MM-ddX").parse("2016-02-15+00"))

   println(Json.encodeToString(data))
    // {"name":"Kotlin","stableReleaseDate":1455494400000}
}
```

Specify serializers with type aliases

In Kotlin serialization, you usually specify serialization strategies explicitly with the `@Serializable` annotation. It doesn't provide a global serializer configuration, except for contextual serialization.

If you use the same serializer repeatedly in many places, you can define a `typealias` with an attached serializer annotation.

This lets you reuse the annotated type without adding the `@Serializable` annotation at each usage site.

Here's an example of using `typealias` to apply `DateAsLongSerializer` and `DateAsSimpleTextSerializer` to `Date`:

```kotlin
import kotlinx.serialization.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.json.*
import java.util.Date
import java.text.SimpleDateFormat
import java.util.TimeZone

object DateAsLongSerializer : KSerializer<Date> {
    override val descriptor: SerialDescriptor = PrimitiveSerialDescriptor("my.app.DateAsLong",
PrimitiveKind.LONG)
    override fun serialize(encoder: Encoder, value: Date) = encoder.encodeLong(value.time)
    override fun deserialize(decoder: Decoder): Date = Date(decoder.decodeLong())
}

// Defines a serializer that encodes Date as a formatted string (yyyy-MM-dd)
object DateAsSimpleTextSerializer: KSerializer<Date> {
    override val descriptor: SerialDescriptor = PrimitiveSerialDescriptor("my.app.DateAsSimpleText",
PrimitiveKind.LONG)
    private val format = SimpleDateFormat("yyyy-MM-dd").apply {
        // Sets the time zone to UTC for consistent output
        setTimeZone(TimeZone.getTimeZone("UTC"))
    }
    override fun serialize(encoder: Encoder, value: Date) = encoder.encodeString(format.format(value))
    override fun deserialize(decoder: Decoder): Date = format.parse(decoder.decodeString())
}

// Applies global serializers using typealias to avoid annotating each occurrence
typealias DateAsLong = @Serializable(DateAsLongSerializer::class) Date

typealias DateAsText = @Serializable(DateAsSimpleTextSerializer::class) Date

// Uses typealiases to apply custom serializers for Date properties
@Serializable
class ProgrammingLanguage(val stableReleaseDate: DateAsText, val lastReleaseTimestamp: DateAsLong)

fun main() {
    val format = SimpleDateFormat("yyyy-MM-ddX")
    val data = ProgrammingLanguage(format.parse("2016-02-15+00"), format.parse("2022-07-07+00"))

    println(Json.encodeToString(data))
    // {"stableReleaseDate":"2016-02-15","lastReleaseTimestamp":1657152000000}
}
```

Implement contextual serialization

By default, serialization strategies are defined at compile time. *Contextual serialization* allows you to adjust the serialization strategy for specific types at runtime, even when they're nested deep within an object tree.

For example, you can use contextual serialization to serialize `java.util.Date` in JSON format either as an ISO 8601 `String` or as a `Long`, depending on the protocol version being used. This approach is supported by the built-in `ContextualSerializer` class.

Contextual serialization selects a custom serializer for a type at runtime from a `SerializersModule`.

To implement contextual serialization:

1. Mark the property in your serializable class with the `@Contextual` annotation.

> The `@Contextual` annotation is a shortcut for the `ContextualSerializer` class. To apply contextual serialization across multiple properties in the same file, use `@UseContextualSerialization` annotation.

2. Create a `SerializersModule` with the `SerializersModule()` builder function. Use the `contextual()` function to register the custom serializer for contextual serialization.

   Without the `SerializersModule`, a `SerializationException` is thrown during the serialization or deserialization of contextually annotated types that don't have a default serializer.

3. Create a `Json` instance and pass the `SerializersModule` to the `serializersModule` property.

Here's an example:

```
import kotlinx.serialization.*
import kotlinx.serialization.encoding.*
import kotlinx.serialization.descriptors.*
import kotlinx.serialization.json.*
import java.util.Date
import java.text.SimpleDateFormat

// Creates a custom serializer for Date
object DateAsLongSerializer : KSerializer<Date> {
    override val descriptor: SerialDescriptor = PrimitiveSerialDescriptor("my.app.Date",
PrimitiveKind.LONG)
    override fun serialize(encoder: Encoder, value: Date) = encoder.encodeLong(value.time)
    override fun deserialize(decoder: Decoder): Date = Date(decoder.decodeLong())
}

@Serializable
class ProgrammingLanguage(
    val name: String,
    // Specifies contextual serialization for Date
    @Contextual
    val stableReleaseDate: Date
)

// Defines a SerializersModule and registers the contextual serializer for Date
private val module = SerializersModule {
    contextual(DateAsLongSerializer)
}

// Creates a Json instance with the custom SerializersModule
val format = Json { serializersModule = module }

fun main() {
    val data = ProgrammingLanguage("Kotlin", SimpleDateFormat("yyyy-MM-ddX").parse("2016-02-15+00"))

   println(format.encodeToString(data))
    // {"name":"Kotlin","stableReleaseDate":1455494400000}
}
```

Serialize generic classes contextually

To serialize generic classes contextually, you can register a function in the `SerializersModule`. This function receives the serializers for the generic type arguments and creates the corresponding serializer at runtime.

You can't use a single serializer instance for generic classes because generic type arguments can vary. For example, the following approach only works for `Box<Int>`, but not for other types like `Box<String>`:

```
val incorrectModule = SerializersModule {
    // This only works for Box<Int>, but not for Box<String> or other types
    contextual(BoxSerializer(Int.serializer()))
}
```

Instead, register a function that creates a serializer for generic types such as `Box<T>` based on the serializers of their type arguments. Here's an example:

```
val correctModule = SerializersModule {
    // args[0] is the serializer for T,
    // for example Int.serializer() or String.serializer()
    contextual(Box::class) { args -> BoxSerializer(args[0]) }
}
```

You can combine multiple `SerializersModule` instances with the `plus` operator, such as modules for generic and non-generic classes. For more information, see Merge multiple `SerializersModule` instances.

What's next

- Discover how to transform JSON structure by modifying the JSON element tree instead of creating a custom serializer.
- Learn about alternative and custom serialization formats to implement format-specific representations for your data.

Kotlin Metadata JVM library

The `kotlin-metadata-jvm` library provides tools to read, modify, and generate metadata from Kotlin classes compiled for the JVM. This metadata, stored in the `@Metadata` annotation within `.class` files, is used by libraries and tools such as `kotlin-reflect` to inspect Kotlin-specific constructs such as properties, functions, and classes at runtime.

The `kotlin-reflect` library relies on metadata to retrieve Kotlin-specific class details at runtime. Any inconsistencies between the metadata and the actual `.class` file may lead to incorrect behavior when using reflection.

You can also use the Kotlin Metadata JVM library to inspect various declaration attributes such as visibility or modality, or to generate and embed metadata into `.class` files.

Add the library to your project

To include the Kotlin Metadata JVM library in your project, add the corresponding dependency configuration based on your build tool.

The Kotlin Metadata JVM library follows the same versioning as the Kotlin compiler and standard library. Ensure that the version you use matches your project's Kotlin version.

Gradle

Add the following dependency to your `build.gradle(.kts)` file:

Kotlin

```
// build.gradle.kts
repositories {
    mavenCentral()
}

dependencies {
    implementation("org.jetbrains.kotlin:kotlin-metadata-jvm:2.4.20")
}
```

Groovy

```
// build.gradle
repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-metadata-jvm:2.4.20'
}
```

Maven

Add the following dependency to your `pom.xml` file.

```
<project>
    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-metadata-jvm</artifactId>
            <version>2.4.20</version>
        </dependency>
    </dependencies>
    ...
</project>
```

Read and parse metadata

The `kotlin-metadata-jvm` library extracts structured information from compiled Kotlin `.class` files, such as class names, visibility, and signatures. You can use it in projects that need to analyze compiled Kotlin declarations. For example, the Binary Compatibility Validator (BCV) relies on `kotlin-metadata-jvm` to print public API declarations.

You can start exploring Kotlin class metadata by retrieving the `@Metadata` annotation from a compiled class using reflection:

```
fun main() {
    // Specifies the fully qualified name of the class
    val clazz = Class.forName("org.example.SampleClass")

    // Retrieves the @Metadata annotation
    val metadata = clazz.getAnnotation(Metadata::class.java)

    // Checks if the metadata is present
    if (metadata ≠ null) {
        println("This is a Kotlin class with metadata.")
    } else {
        println("This is not a Kotlin class.")
    }
}
```

After retrieving the `@Metadata` annotation, use either the `readLenient()` or the `readStrict()` function from the `KotlinClassMetadata` API to parse it. These functions extract detailed information about classes or files, while addressing different compatibility requirements:

- `readLenient()` : Use this function to read metadata, including metadata generated by newer Kotlin compiler versions. This function doesn't support modifying or writing metadata.
- `readStrict()` : Use this function when you need to modify and write metadata. The `readStrict()` function only works with metadata generated by Kotlin compiler versions fully supported by your project.

> The `readStrict()` function supports metadata formats up to one version beyond `JvmMetadataVersion.LATEST_STABLE_SUPPORTED` , which corresponds to the latest Kotlin version used in the project. For example, if your project depends on `kotlin-metadata-jvm:2.1.0` , `readStrict()` can process metadata up to Kotlin `2.2.x` ; otherwise, it throws an error to prevent mishandling unknown formats.
>
> For more information, see the Kotlin Metadata GitHub repository.

When parsing metadata, the `KotlinClassMetadata` instance provides structured information about class or file-level declarations. For classes, use the `kmClass` property to analyze detailed class-level metadata, such as the class name, functions, properties, and attributes like visibility. For file-level declarations, the metadata is represented by the `kmPackage` property, which includes top-level functions and properties from file facades generated by the Kotlin compiler.

The following code example demonstrates how to use `readLenient()` to parse metadata, analyze class-level details with `kmClass` , and retrieve file-level declarations with `kmPackage` :

```
// Imports the necessary libraries
import kotlin.metadata.jvm.*
import kotlin.metadata.*

fun main() {
    // Specifies the fully qualified class name
    val className = "org.example.SampleClass"

    try {
        // Retrieves the class object for the specified name
        val clazz = Class.forName(className)

        // Retrieves the @Metadata annotation
        val metadataAnnotation = clazz.getAnnotation(Metadata::class.java)
        if (metadataAnnotation ≠= null) {
            println("Kotlin Metadata found for class: $className")

            // Parses metadata using the readLenient() function
            val metadata = KotlinClassMetadata.readLenient(metadataAnnotation)
            when (metadata) {
                is KotlinClassMetadata.Class → {
                    val kmClass = metadata.kmClass
                    println("Class name: ${kmClass.name}")

                    // Iterates over functions and checks visibility
                    kmClass.functions.forEach { function →
                        val visibility = function.visibility
                        println("Function: ${function.name}, Visibility: $visibility")
                    }
                }
                is KotlinClassMetadata.FileFacade → {
                    val kmPackage = metadata.kmPackage

                    // Iterates over functions and checks visibility
                    kmPackage.functions.forEach { function →
                        val visibility = function.visibility
                        println("Function: ${function.name}, Visibility: $visibility")
                    }
                }
                else → {
                    println("Unsupported metadata type: $metadata")
                }
            }
        } else {
            println("No Kotlin Metadata found for class: $className")
        }
    } catch (e: ClassNotFoundException) {
        println("Class not found: $className")
    } catch (e: Exception) {
        println("Error processing metadata: ${e.message}")
        e.printStackTrace()
    }
}
```

Write and read annotations in metadata

Kotlin stores annotations in both bytecode and Kotlin metadata. If you use the `kotlin-metadata-jvm` library to read or write annotations, you work with their metadata representation.

Kotlin stores annotations in Kotlin metadata starting with Kotlin 2.4.0. If you inspect class files compiled with earlier versions, the annotations aren't present in the metadata.

When you change the annotations in metadata, make sure they stay consistent with the annotations stored in bytecode. If they get out of sync, tools that rely on reflection or bytecode analysis may report different results than tools that read Kotlin metadata.

The `kotlin-metadata-jvm` library provides the following APIs for accessing annotations:

- `KmClass.annotations`
- `KmFunction.annotations`
- `KmProperty.annotations`
- `KmConstructor.annotations`
- `KmPropertyAccessorAttributes.annotations`
- `KmValueParameter.annotations`
- `KmFunction.extensionReceiverAnnotations`
- `KmProperty.extensionReceiverAnnotations`
- `KmProperty.backingFieldAnnotations`
- `KmProperty.delegateFieldAnnotations`
- `KmEnumEntry.annotations`

Here's an example of reading annotations from Kotlin metadata:

```
import kotlin.metadata.ExperimentalAnnotationsInMetadata
import kotlin.metadata.jvm.KotlinClassMetadata

annotation class Label(val value: String)

@Label("Message class")
class Message

fun main() {
    val metadata = Message::class.java.getAnnotation(Metadata::class.java)
    val kmClass = (KotlinClassMetadata.readStrict(metadata) as KotlinClassMetadata.Class).kmClass
    println(kmClass.annotations)
    // [@Label(value = StringValue("Message class"))]
}
```

Extract metadata from bytecode

While you can retrieve metadata using reflection, another approach is to extract it from bytecode using a bytecode manipulation framework such as ASM.

You can do this by following these steps:

1. Read the bytecode of a `.class` file using the ASM library's `ClassReader` class. This class processes the compiled file and populates a `ClassNode` object, which represents the class structure.
2. Extract the `@Metadata` from the `ClassNode` object. The example below uses a custom extension function `findAnnotation()` for this.

3. Parse the extracted metadata using the `KotlinClassMetadata.readLenient()` function.
4. Inspect the parsed metadata with the `kmClass` and `kmPackage` properties.

Here's an example:

```
// Imports the necessary libraries
import kotlin.metadata.jvm.*
import kotlin.metadata.*
import org.objectweb.asm.*
import org.objectweb.asm.tree.*
import java.io.File

// Checks if an annotation refers to a specific name
fun AnnotationNode.refersToName(name: String) =
    desc.startsWith('L') && desc.endsWith(';') && desc.regionMatches(1, name, 0, name.length)

// Retrieves annotation values by key
private fun List<Any>.annotationValue(key: String): Any? {
    for (index in (0 until size / 2)) {
        if (this[index * 2] == key) {
            return this[index * 2 + 1]
        }
    }
    return null
}

// Defines a custom extension function to locate an annotation by its name in a ClassNode
fun ClassNode.findAnnotation(annotationName: String, includeInvisible: Boolean = false):
AnnotationNode? {
    val visible = visibleAnnotations?.firstOrNull { it.refersToName(annotationName) }
    if (!includeInvisible) return visible
    return visible ?: invisibleAnnotations?.firstOrNull { it.refersToName(annotationName) }
}

// Operator to simplify retrieving annotation values
operator fun AnnotationNode.get(key: String): Any? = values.annotationValue(key)

// Extracts Kotlin metadata from a class node
fun ClassNode.readMetadataLenient(): KotlinClassMetadata? {
    val metadataAnnotation = findAnnotation("kotlin/Metadata", false) ?: return null
    @Suppress("UNCHECKED_CAST")
    val metadata = Metadata(
        kind = metadataAnnotation["k"] as Int?,
        metadataVersion = (metadataAnnotation["mv"] as List<Int>?)?.toIntArray(),
        data1 = (metadataAnnotation["d1"] as List<String>?)?.toTypedArray(),
        data2 = (metadataAnnotation["d2"] as List<String>?)?.toTypedArray(),
        extraString = metadataAnnotation["xs"] as String?,
        packageName = metadataAnnotation["pn"] as String?,
        extraInt = metadataAnnotation["xi"] as Int?
    )
    return KotlinClassMetadata.readLenient(metadata)
}

// Converts a file to a ClassNode for bytecode inspection
fun File.toClassNode(): ClassNode {
    val node = ClassNode()
    this.inputStream().use { ClassReader(it).accept(node, ClassReader.SKIP_CODE) }
    return node
}

fun main() {
    val classFilePath = "build/classes/kotlin/main/org/example/SampleClass.class"
    val classFile = File(classFilePath)

    // Reads the bytecode and processes it into a ClassNode object
    val classNode = classFile.toClassNode()

    // Locates the @Metadata annotation and reads it leniently
```

```
    val metadata = classNode.readMetadataLenient()
    if (metadata ≠= null && metadata is KotlinClassMetadata.Class) {
        // Inspects the parsed metadata
        val kmClass = metadata.kmClass

        // Prints class details
        println("Class name: ${kmClass.name}")
        println("Functions:")
        kmClass.functions.forEach { function →
            println("- ${function.name}, Visibility: ${function.visibility}")
        }
    }
}
```

Modify metadata

When using tools like ProGuard to shrink and optimize bytecode, some declarations may be removed from `.class` files. ProGuard automatically updates metadata to keep it consistent with the modified bytecode.

However, if you're developing a custom tool that modifies Kotlin bytecode in a similar way, you need to ensure that metadata is adjusted accordingly. With the `kotlin-metadata-jvm` library, you can update declarations, adjust attributes, and remove specific elements.

For example, if you use a JVM tool that deletes private methods from Java class files, you must also delete private functions from Kotlin metadata to maintain consistency:

1. Parse the metadata by using the `readStrict()` function to load the `@Metadata` annotation into a structured `KotlinClassMetadata` object.
2. Apply modifications by adjusting the metadata, such as filtering functions or altering attributes, directly within `kmClass` or other metadata structures.
3. Use the `write()` function to encode the modified metadata into a new `@Metadata` annotation.

Here's an example where private functions are removed from a class's metadata:

```
// Imports the necessary libraries
import kotlin.metadata.jvm.*
import kotlin.metadata.*

fun main() {
    // Specifies the fully qualified class name
    val className = "org.example.SampleClass"

    try {
        // Retrieves the class object for the specified name
        val clazz = Class.forName(className)

        // Retrieves the @Metadata annotation
        val metadataAnnotation = clazz.getAnnotation(Metadata::class.java)
        if (metadataAnnotation ≠ null) {
            println("Kotlin Metadata found for class: $className")

            // Parses metadata using the readStrict() function
            val metadata = KotlinClassMetadata.readStrict(metadataAnnotation)
            if (metadata is KotlinClassMetadata.Class) {
                val kmClass = metadata.kmClass

                // Removes private functions from the class metadata
                kmClass.functions.removeIf { it.visibility == Visibility.PRIVATE }
                println("Removed private functions. Remaining functions: ${kmClass.functions.map {
it.name }}")

                // Serializes the modified metadata back
                val newMetadata = metadata.write()
                // After modifying the metadata, you need to write it into the class file
                // To do so, you can use a bytecode manipulation framework such as ASM

                println("Modified metadata: ${newMetadata}")
            } else {
                println("The metadata is not a class.")
            }
        } else {
            println("No Kotlin Metadata found for class: $className")
        }
    } catch (e: ClassNotFoundException) {
        println("Class not found: $className")
    } catch (e: Exception) {
        println("Error processing metadata: ${e.message}")
        e.printStackTrace()
    }
}
```

Instead of separately calling `readStrict()` and `write()`, you can use the `transform()` function. This function parses metadata, applies transformations through a lambda, and writes the modified metadata automatically.

Create metadata from scratch

To create metadata for a Kotlin class file from scratch using the Kotlin Metadata JVM library:

1. Create an instance of `KmClass`, `KmPackage`, or `KmLambda`, depending on the type of metadata you want to generate.
2. Add attributes to the instance, such as the class name, visibility, constructors, and function signatures.

> You can use the `apply()` scope function to reduce boilerplate code while setting properties.

3. Use the instance to create a `KotlinClassMetadata` object, which can generate a `@Metadata` annotation.
4. Specify the metadata version, such as `JvmMetadataVersion.LATEST_STABLE_SUPPORTED`, and set flags (`0` for no flags, or copy flags from existing files if necessary).
5. Use the `ClassWriter` class from ASM to embed metadata fields, such as `kind`, `data1` and `data2` into a `.class` file.

The following example demonstrates how to create metadata for a simple Kotlin class:

```
// Imports the necessary libraries
import kotlin.metadata.*
import kotlin.metadata.jvm.*
import org.objectweb.asm.*

fun main() {
    // Creates a KmClass instance
    val klass = KmClass().apply {
        name = "Hello"
        visibility = Visibility.PUBLIC
        constructors += KmConstructor().apply {
            visibility = Visibility.PUBLIC
            signature = JvmMethodSignature("<init>", "()V")
        }
        functions += KmFunction("hello").apply {
            visibility = Visibility.PUBLIC
            returnType = KmType().apply {
                classifier = KmClassifier.Class("kotlin/String")
            }
            signature = JvmMethodSignature("hello", "()Ljava/lang/String;")
        }
    }

    // Serializes a KotlinClassMetadata.Class instance, including the version and flags, into a
@kotlin.Metadata annotation
    val annotationData = KotlinClassMetadata.Class(
        klass, JvmMetadataVersion.LATEST_STABLE_SUPPORTED, 0
    ).write()

    // Generates a .class file with ASM
    val classBytes = ClassWriter(0).apply {
        visit(Opcodes.V1_6, Opcodes.ACC_PUBLIC, "Hello", null, "java/lang/Object", null)
        // Writes @kotlin.Metadata instance to the .class file
        visitAnnotation("Lkotlin/Metadata;", true).apply {
            visit("mv", annotationData.metadataVersion)
            visit("k", annotationData.kind)
            visitArray("d1").apply {
                annotationData.data1.forEach { visit(null, it) }
                visitEnd()
            }
            visitArray("d2").apply {
                annotationData.data2.forEach { visit(null, it) }
                visitEnd()
            }
            visitEnd()
        }
        visitEnd()
    }.toByteArray()

    // Writes the generated .class file to disk
    java.io.File("Hello.class").writeBytes(classBytes)

    println("Metadata and .class file created successfully.")
}
```

For a more detailed example, see the Kotlin Metadata JVM GitHub repository.

What's next?

- See the API reference for the Kotlin Metadata JVM library.
- Check out the Kotlin Metadata JVM GitHub repository.
- Learn about module metadata and working with `.kotlin_module` files.

Overview

Lincheck is a framework for testing concurrent code on the JVM. When running tests, Lincheck explores the potential thread interleavings of the program and reports the ones that lead to incorrect behavior.

> In Kotlin Multiplatform projects, you can use Lincheck to test code only on the JVM.

A concurrency test in Lincheck only requires you to list the operations for each thread and the expected assertions. Lincheck handles the rest:

```
class CounterTest {
    @Test // Test function declaration
    fun test() = Lincheck.runConcurrentTest {
        var counter = 0

        // Increments the counter concurrently
        val t1 = thread { counter++ }
        val t2 = thread { counter++ }

        // Waits for the threads to finish
        t1.join()
        t2.join()

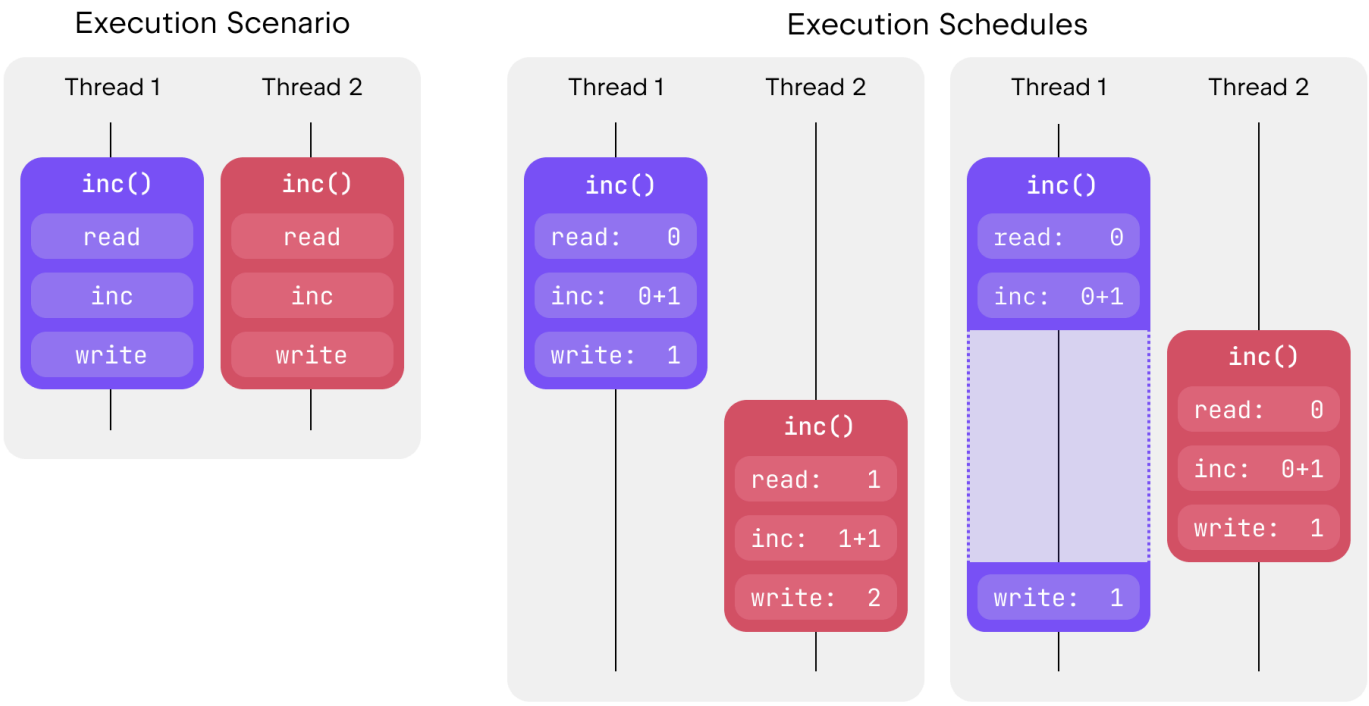
        // Checks that both increments have been applied
        assertEquals(2, counter)
    }
}
```

If the test fails, Lincheck provides the thread interleaving and the thread switch points that led to an error:

```
| ------------------------------------------------------------------------- |
|                  Main Thread                   |   Thread 1    |   Thread 2    |
| ------------------------------------------------------------------------- |
| thread(block = Lambda#2): Thread#1             |               |               |
| thread(block = Lambda#3): Thread#2             |               |               |
| switch (reason: waiting for Thread 1 to finish) |               |               |
|                                                |               | run()         |
|                                                |               |   counter → 0 |
|                                                |               |   switch      |
|                                                | run()         |               |
|                                                |   counter → 0 |               |
|                                                |   counter = 1 |               |
|                                                |               |   counter = 1 |
| Thread#1.join()                                |               |               |
| Thread#2.join()                                |               |               |
| counter.element → 1                            |               |               |
| assertEquals(2, 1): threw AssertionFailedError |               |               |
| ------------------------------------------------------------------------- |
```

How Lincheck works

Each time the JVM runs concurrent code, the execution order of the operations across threads might change. For example, an operation can be interrupted by another operation in a different thread. This is not an error by itself, but can lead to an error if the code has concurrency bugs.



The image compares the execution scenario of a program to execution schedules. In the first execution schedule, operations are performed one after the other. In the second execution schedule, the first operation is interrupted by the second operation.

> An *execution scenario* defines how operations are distributed across the threads and the order of execution within each thread.
>
> An *execution schedule* (also called *thread interleaving*) defines the order of execution for all operations across all threads.

Lincheck implements two testing strategies to find execution schedules that lead to incorrect behavior:

- **Model checking**. Lincheck controls scheduling by inserting explicit thread switch instructions into the program. These instructions are placed at synchronization points or shared memory accesses. Model checking enables Lincheck to generate an exact execution trace that leads to an error.
- **Stress testing**. The operating system controls scheduling. Lincheck executes each scenario multiple times to increase the chances of finding an error.

Explore Lincheck

- Learn Lincheck features step-by-step in Getting started with Lincheck.
- Learn about the declarative approach to testing concurrent data structures in Lincheck.

Learn more

- "How we test concurrent algorithms in Kotlin Coroutines" by Nikita Koval: Video. KotlinConf 2023
- "Lincheck: Testing concurrency on the JVM" workshop by Maria Sokolova: Part 1, Part 2. Hydra 2021
- "Lincheck: A Practical Framework for Testing Concurrent Data Structures on JVM" by Nikita Koval et al.: Paper. 2023

Getting started with Lincheck

This quickstart will guide you through setting up Lincheck, writing your first Lincheck tests, and interpreting the test reports.

You will:

- Create a new IntelliJ IDEA project and install Lincheck.
- Write your first concurrent test and run it with Lincheck.
- Create a concurrent data structure and test it with Lincheck using two testing strategies.

Create a project

Open an existing Kotlin project in IntelliJ IDEA or create a new one.

Add dependencies

To use Lincheck in the project, add the corresponding dependencies to your build configuration:

Kotlin

```
// build.gradle.kts
repositories {
    mavenCentral()
}

dependencies {
    testImplementation("org.jetbrains.lincheck:lincheck:3.7")
    testImplementation(kotlin("test"))
}
```

Groovy

```
// build.gradle
repositories {
    mavenCentral()
}

dependencies {
    testImplementation "org.jetbrains.lincheck:lincheck:3.7"
    testImplementation "org.jetbrains.kotlin:kotlin-test"
}
```

Maven

```
<!-- pom.xml -->
<project>
    <dependencies>
        <dependency>
            <groupId>org.jetbrains.lincheck</groupId>
            <artifactId>lincheck</artifactId>
            <version>3.7</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    ...
</project>
```

Write your first test

For a basic concurrent test, create a test function that describes what operations should be executed in each thread and the expected assertions. Lincheck explores possible thread interleavings of the program using model checking and provides an error report in case of incorrect behavior.

1. In the `src/test` directory, create a `CounterTest.kt` file.
2. Import the `org.jetbrains.lincheck`, `kotlinx.concurrent`, and `kotlin.test` libraries:

```
import org.jetbrains.lincheck.Lincheck
import kotlin.test.*
import kotlin.concurrent.thread
```

3. Write a test that creates a variable and two threads manipulating that variable:

```
class CounterTest {
    @Test // Test function declaration
    fun test() = Lincheck.runConcurrentTest {
        var counter = 0

        // Increments the counter concurrently
        val t1 = thread { counter++ }
        val t2 = thread { counter++ }

        // Waits for the threads to finish
        t1.join()
        t2.join()

        // Checks that both increments have been applied
        assertEquals(2, counter)
    }
}
```

4. Run the test. Lincheck generates a report with a thread interleaving that led to incorrect behavior:

Install the Lincheck plugin to visualize the error trace.

```
| ------------------------------------------------------------------------ |
|                  Main Thread                   |   Thread 1    |   Thread 2    |
| ------------------------------------------------------------------------ |
| thread(block = Lambda#2): Thread#1             |               |               |
| thread(block = Lambda#3): Thread#2             |               |               |
| switch (reason: waiting for Thread 1 to finish) |               |               |
|                                                |               | run()         |
|                                                |               |   counter → 0 |
|                                                |               |   switch      |
|                                                | run()         |               |
|                                                |   counter → 0 |               |
|                                                |   counter = 1 |               |
|                                                |               |   counter = 1 |
| Thread#1.join()                                |               |               |
| Thread#2.join()                                |               |               |
| counter.element → 1                            |               |               |
| assertEquals(2, 1): threw AssertionFailedError  |               |               |
| ------------------------------------------------------------------------ |
```

Lincheck found a thread interleaving where one of the `inc()` operations overwrites the `counter` value.

Step-by-step report explanation

1. In Thread 2, the JVM reads the initial `counter` value.
2. The execution switches from Thread 2 to Thread 1.
3. In Thread 1, the JVM increments the counter. All steps of the `inc()` operation are performed without interruptions: reading the value from the variable, incrementing the value, and writing the value back to the variable.
4. The execution switches back to Thread 2.
5. In Thread 2, the JVM increments the value acquired at step 1 and writes the result to the `counter` variable.

Write a test for a data structure

In addition to basic concurrent tests, Lincheck supports a declarative approach to testing concurrent data structures.

To test a data structure in Lincheck, you only need to declare the concurrent methods of the structure and a test function. Lincheck generates random concurrent scenarios, executes them using the specified testing strategy, and provides error reports.

In this section, you will test a simple counter:

1. In the `src/test` directory, create a `CounterStructureTest.kt` file.
2. Import the `lincheck.datastructures` and `kotlin.test` libraries:

```
import org.jetbrains.lincheck.datastructures.*
import kotlin.test.*
```

3. Create a `Counter` structure:

```
class Counter {
    var value = 0

    fun inc(): Int = ++value
    fun dec(): Int = --value
}
```

4. Create a `CounterStructureTest` class. Set the initial state of the structure and mark the concurrent operations of the structure with the `@Operation` annotation:

```
class CounterStructureTest {
    private val c = Counter()

    @Operation
    fun inc() = c.inc()

    @Operation
    fun dec() = c.dec()
        .iterations(100) // Specify the number of generated scenarios
```

5. In the `CounterStructureTest` class, declare a test function using `ModelCheckingOptions()`:

```
@Test
    fun test() = ModelCheckingOptions().check(this::class)
```

> Learn how model checking works in the Testing Strategies article.

6. Run the test. Lincheck generates an error report with the concurrent scenario and the specific thread interleaving that led to incorrect behavior:

```
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| inc(): 1 | inc(): 1 |
| ------------------- |
```

```
| ------------------------ |
| Thread 1 |   Thread 2    |
| ------------------------ |
|          | inc(): 1      |
|          |   c.inc(): 1  |
|          |     value → 0 |
|          |     switch    |
| inc(): 1 |               |
|          |     value = 1 |
|          |     value → 1 |
|          |   result: 1   |
| ------------------------ |
```

What's next

Learn about the declarative approach to testing concurrent data structures in Lincheck.

Testing arbitrary code

Lincheck provides a `runConcurrentTest()` function to test arbitrary concurrent code.

The `runConcurrentTest()` function executes a block of concurrent code multiple times and uses model checking to explore its potential execution schedules.

To test concurrent code with Lincheck:

1. Create a test class:

```
class CounterTestWithInvocations {
}
```

2. Create a test function as a member function using `runConcurrentTest()`:

```
@Test
    fun test() = Lincheck.runConcurrentTest(100_000) {
        var counter = 0

        // Increments the counter concurrently
        val t1 = thread { counter++ }
        val t2 = thread { counter++ }

        // Waits for the threads to finish
        t1.join()
        t2.join()

        // Checks that both increments have been applied
        assertEquals(2, counter)
    }
```

> The function parameter is optional; it specifies the number of execution schedules to explore. The default value is `10_000`.

3. Run the test. If it fails, Lincheck generates a report with an execution schedule that leads to incorrect behavior.

```
| ------------------------------------------------------------------------ |
|                  Main Thread                   |   Thread 1    |   Thread 2    |
| ------------------------------------------------------------------------ |
| thread(block = Lambda#2): Thread#1             |               |               |
| thread(block = Lambda#3): Thread#2             |               |               |
| switch (reason: waiting for Thread 1 to finish) |               |               |
|                                                |               | run()         |
|                                                |               |   counter → 0 |
|                                                |               |   switch      |
|                                                | run()         |               |
|                                                |   counter → 0 |               |
|                                                |   counter = 1 |               |
|                                                |               |   counter = 1 |
| Thread#1.join()                                |               |               |
| Thread#2.join()                                |               |               |
| counter.element → 1                            |               |               |
| assertEquals(2, 1): threw AssertionFailedError |               |               |
| ------------------------------------------------------------------------ |
```

Example: test `ConcurrentHashMap` functions

Consider this test for `ConcurrentHashMap` functions:

```
package org.lincheck.docs

import org.jetbrains.lincheck.*
import kotlin.test.Test
import java.util.concurrent.*
import kotlin.concurrent.*

// This test demonstrates a deadlock caused by two threads
// performing nested `computeIfAbsent` calls in opposite order.
class ConcurrentHashMapDeadlockTest {
    @Test
    fun test() = Lincheck.runConcurrentTest {
        val map = ConcurrentHashMap<String, String>()

        // Updates `key2` while locking `key1`.
        val thread1 = thread {
            map.computeIfAbsent("key1") {
                map.computeIfAbsent("key2") { "value2" }
                "value1"
            }
        }

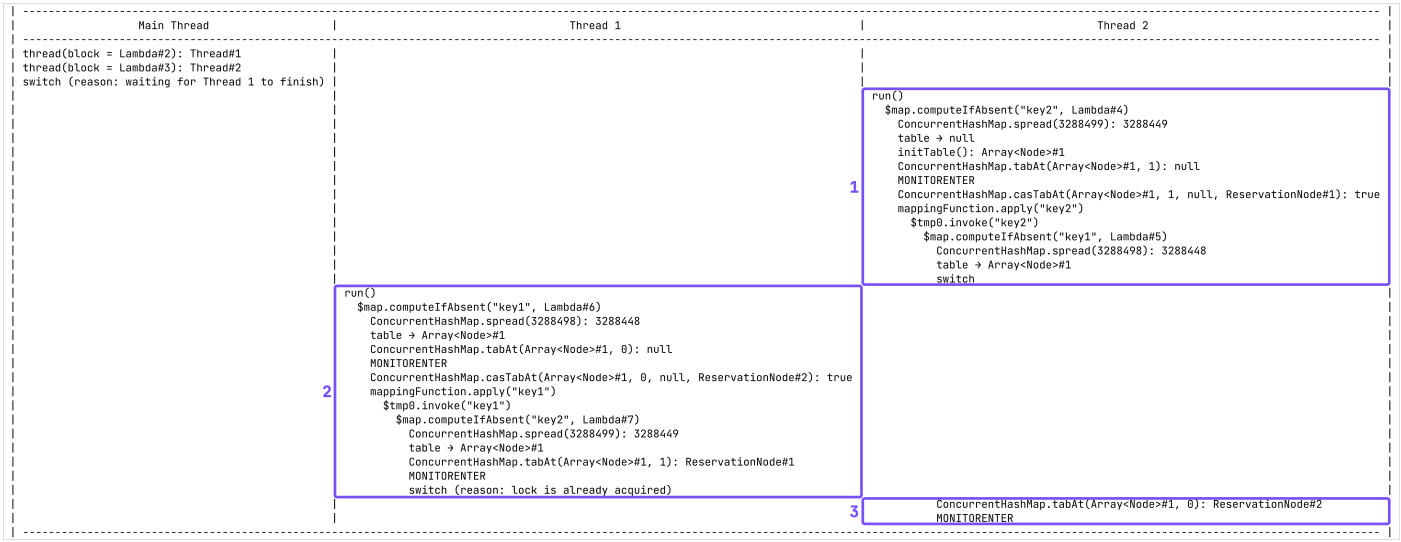
        // Updates `key1` while locking `key2`.
        val thread2 = thread {
            map.computeIfAbsent("key2") {
                map.computeIfAbsent("key1") { "value1" }
                "value2"
            }
        }

        // Wait until both threads complete.
        thread1.join()
        thread2.join()
    }
}
```

The test fails due to Lincheck finding an execution schedule that leads to a deadlock:

1. Thread 2 maps `key2` to the bucket at index 1, places a lock on this bucket, and starts executing `computeIfAbsent("key1")`. The execution switches from Thread 2 to Thread 1 **before** Thread 2 maps `key1` and locks the bucket with `key1`.
2. Thread 1 maps `key1` to the bucket at index 0, places a lock on this bucket, and starts executing `computeIfAbsent("key2")`. Thread 1 maps `key2` to the bucket at index 1 and tries to lock the bucket, but it is already locked by Thread 2. The execution switches from Thread 1 to Thread 2.
3. Thread 2 tries to lock the bucket with `key1`, but it is already locked by Thread 1.

Both threads are locked, so the execution has encountered a deadlock.



A screenshot of the Lincheck report for the failed test.

What's next

Learn how to test data structures using Lincheck.

See also

- Model checking in Lincheck
- Lincheck in Kotlin Multiplatform projects

How to test data structures

Lincheck provides a declarative interface for testing concurrent data structures. Instead of describing how to perform the test, you declare all the operations that need to be tested, and Lincheck generates concurrent execution scenarios, runs them, and analyzes the results.

Let's test this `Counter` data structure with Lincheck:

```
class Counter {
    var value = 0

    fun inc(): Int = ++value
    fun dec(): Int = --value
}
```

1. Create a test class:

   ```
   class CounterTest {
   }
   ```

2. Create a class property that holds an instance of your structure:

   ```
   private val c = Counter()
   ```

3. Declare the operations you want to test as member functions and annotate them with `@Operation`:

```
@Operation
    fun inc() = c.inc()

    @Operation
    fun dec() = c.dec()
```

This annotation tells Lincheck which methods to include when generating execution scenarios.

4. Declare a test function as a member function using `ModelCheckingOptions()` or `StressOptions()`. Annotate it with `@Test`:

```
@Test
    fun test() = ModelCheckingOptions().check(this::class)
```

> Learn about the differences between model checking and stress testing in the Testing Strategies article.

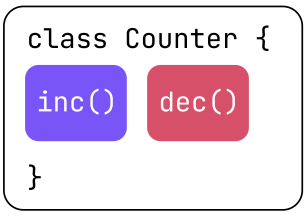
5. Run the test. If it fails, Lincheck generates an error report with the scenario and execution trace that led to incorrect behavior:

```
= Invalid execution results =
| ------------------- |
| Thread 1  | Thread 2 |
| ------------------- |
| dec(): -1 | inc(): 1 |
| ------------------- |
```

The testing process

When testing a data structure, Lincheck generates a list of execution scenarios, runs them, and analyzes the results.

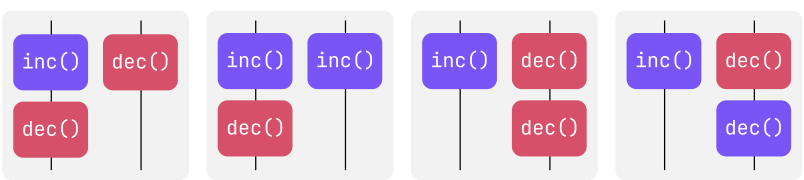
Consider this `Counter` data structure:



A diagram of a Counter data structure with two methods: inc() and dec()

To test it, Lincheck performs the following steps:

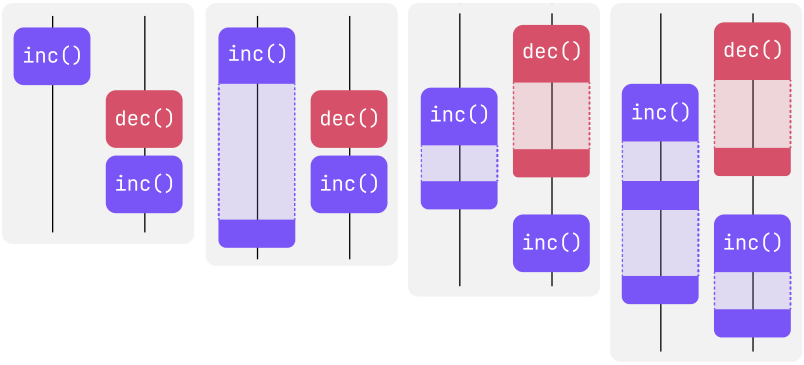
1. Generates a list of random execution scenarios by randomly placing declared operations across different threads:



A diagram of four execution scenarios. In each scenario, operations are placed in different orders in two threads.

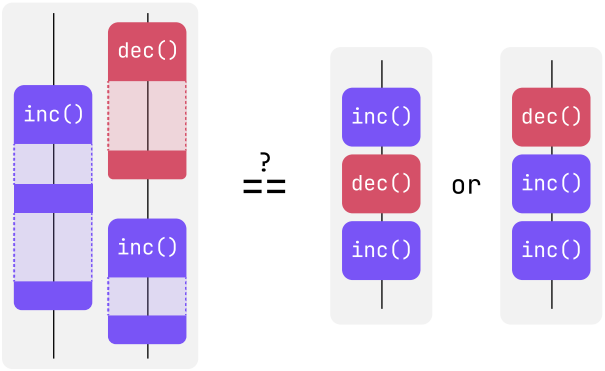
You can specify the number of threads and the number of operations per thread using the configuration options provided by Lincheck.

2. Executes the generated scenarios using the specified testing strategy: model checking or stress testing. Each of the generated scenarios is executed multiple times to examine different execution schedules:



A diagram of four execution schedules. All schedules correspond to a single execution scenario. In each schedule, the operations interrupt each other at different times.

3. Verifies the results of the execution against the correctness property. By default, it's linearizability.



A diagram of the verification process. The results of one execution schedule are compared to the results of the same operations executed sequentially.

At this step, Lincheck can also validate the structure if provided with a validation function.

Example: test an implementation of a Treiber stack structure

Consider this *incorrect* implementation of a Treiber Stack:

```
class TreiberStack<E> {
    private val top = AtomicReference<Node<E>?>(null)

    fun push(item: E) {
        val newHead = Node(item)
        var oldHead: Node<E>?

        do {
            oldHead = top.get()
            newHead.next = oldHead
        } while (!top.compareAndSet(oldHead, newHead))
    }

    fun pop(): E? {
        val oldHead = top.get()

        if (oldHead == null) {
            return null
        }

        val newHead = oldHead.next
        top.compareAndSet(oldHead, newHead)

        // Bug: by the time `pop()` finishes execution,
        // another thread might have already popped this item.
        return oldHead.item
    }

    private class Node<E>(
        val item: E,
        var next: Node<E>? = null
    )
}
```

You can test this structure with Lincheck to examine how the injected bug affects the behavior of the program:

1. Create a test structure:

```
class TreiberStackTest {
    private val stack = TreiberStack<Int>()

    @Operation
    fun push(value: Int) = stack.push(value)

    @Operation
    fun pop(): Int? = stack.pop()

    @Test
    fun modelCheckingTest() = ModelCheckingOptions()
        .check(this::class)
}
```

2. Run the test. Lincheck generates an error report and provides the execution scenario that causes incorrect behavior:

```
| ------------------------------ |
|   Thread 1    |    Thread 2    |
| ------------------------------ |
| push(1): void |                |
| ------------------------------ |
| pop(): 1      | push(-1): void |
| ------------------------------ |
| pop(): -1     |                |
| pop(): 1      |                |
| ------------------------------ |
```

This diagram shows how the operations are placed across different threads and the return values of the operations. Lincheck also provides the specific thread interleaving that leads to incorrect results:

```
| ------------------------------------------------------ |
|                  Thread 1                  | Thread 2  |
| ------------------------------------------------------ |
| push(1)                                    |           |
| ------------------------------------------------------ |
| pop(): 1                                   |           |
|   stack.pop(): 1                           |           |
|     top.get(): Node#1                      |           |
|     switch                                 |           |
|                                            | push(-1)  |
|     oldHead.getNext(): null                |           |
|     top.compareAndSet(Node#1, null): false |           |
|     oldHead.getItem(): 1                   |           |
|   result: 1                                |           |
| ------------------------------------------------------ |
| pop(): -1                                  |           |
| pop(): 1                                   |           |
| ------------------------------------------------------ |
```

Because the implementation does not account for another thread interrupting the `pop()` function, `pop()` returns `1` twice, which should not be possible.

3. Fix the data structure. A correct implementation updates the `oldHead` variable to the most recent value before returning the results:

```
fun pop(): E? {
        var oldHead: Node<E>?
        var newHead: Node<E>?

        do {
            oldHead = top.get()
            if (oldHead == null) return null
            newHead = oldHead.next
        } while (!top.compareAndSet(oldHead, newHead))

        return oldHead.item
    }
```

What's next

Learn about testing strategies available in Lincheck.

See also

- Generating operation arguments
- Configuring operation execution options
- Checking for non-blocking progress guarantees
- Defining sequential specification of the algorithm

Testing strategies

Lincheck provides two strategies for testing concurrent data structures: model checking and stress testing.

In this article, you will learn the difference between the strategies and what to keep in mind when choosing a testing strategy.

Model checking

With model checking, Lincheck simulates possible thread interleavings and reports those that cause incorrect behavior.

To use model checking for testing a data structure, declare a test function using `ModelCheckingOptions()`:

```
@Test
    fun test() = ModelCheckingOptions().check(this::class)
```

When using the model checking strategy, Lincheck inserts explicit thread-switch instructions at points of shared-memory access (`read` and `write`) or at synchronization points, such as lock acquisition and release, `park`/`unpark`, `wait`/`notify`, and others.

Controlling the thread switching enables Lincheck to:

- Deterministically explore different possible execution schedules of a program.
- Provide detailed execution traces.

Currently, model checking requires Lincheck to assume a sequentially consistent memory model of the execution. This means that Lincheck does not simulate and cannot catch bugs related to instruction reordering, memory cache behavior, and other similar effects under the relaxed Java memory model.

> For more information, see Model checking.

Stress testing

With stress testing, Lincheck executes each scenario multiple times to increase the chances of finding an error.

To use stress testing, declare a test function using `StressOptions()`:

```
@Test
    fun stressTest() = StressOptions().check(this::class)
```

Unlike with model checking, Lincheck does not control or track thread switches. This makes stress testing faster and does not require Lincheck to make any assumptions about the memory model. However, with stress testing, tests are not reproducible, and Lincheck cannot provide an execution trace.

Choose a strategy

When choosing a strategy, consider the following:

	Model checking	Stress testing
Speed	Slower.	Faster.
Reproducibility	The tests return exactly the same results if the input data has not changed.	The tests might return different results as the thread schedules might change from run to run.
Assumptions	• Assumes a sequentially consistent memory model. • Will miss bugs caused by incorrect behavior outside of that model.	• Does not make any assumptions about a memory model. • Has a chance to catch any incorrect behavior regardless of the underlying cause.
Verbosity	Reports both a concurrent scenario and an execution trace that led to incorrect behavior.	Reports only a concurrent scenario.
Standard library coverage	• Does not simulate the behavior of some standard library features, such as weak references. • Will miss bugs caused by such features.	Has a chance to catch a bug caused by the usage of any feature.

What's next

Learn how to configure a testing strategy by customizing scenario generation, enabling stalled execution detection, and providing thread-safety guarantees for libraries.

See also

- Generating operation arguments
- Configuring operation execution options
- Checking for non-blocking progress guarantees
- Defining sequential specification of the algorithm

Configuring a testing strategy

Lincheck supports various configuration options for testing strategies, including scenario generation, detection of stalled execution, verification, and others.

How to enable options

To enable an option for a testing strategy, set it in the strategy class:

```
@Test
    fun testWithIterations() = ModelCheckingOptions()
        .iterations(100) // Specify the number of generated scenarios
        .check(this::class)
```

Scenario minimization

By default, Lincheck tries to minimize the failed scenario by removing the operations that do not change the behavior of the test.

Set the `minimizeFailedScenario` option to `false` to see the full failed scenario.

```
| ------------------------- |
|    Thread 1    | Thread 2 |
| -------------------------|
| inc(): 1       |          |
| get(): 1       |          |
| get(): 1       |          |
| -------------------------|
| inc(): 4 [0,1] | inc(): 2 |
| get(): 4 [1,1] | inc(): 4 |
| get(): 4 [2,1] | get(): 4 |
| -------------------------|
| get(): 4       |          |
| get(): 4       |          |
| get(): 4       |          |
| -------------------------|
```

```
| ------------------ |
| Thread 1 | Thread 2 |
| ------------------ |
| inc(): 1 | inc(): 1 |
| ------------------ |
```

Scenario generation

Option	Default value	Description
`iterations`	`100`	The number of generated concurrent scenarios.

Option	Default value	Description
`invocationsPerIteration`	`10_000`	The number of invocations for each concurrent scenario.
`threads`	`2`	The number of threads in each scenario.
`actorsBefore`	`5`	The number of operations called before the parallel section of a scenario.
`actorsPerThread`	`5`	The number of operations in each thread in the parallel section of a scenario.
`actorsAfter`	`5`	The number of operations called after the parallel section of a scenario.
`customScenarios`	–	A list of custom concurrent scenarios. The custom scenarios are executed before the randomly generated ones.

Defining a custom scenario

Lincheck uses a domain-specific language for defining custom scenarios:

```
@Test
    fun test() = StressOptions()
        .addCustomScenario {
            initial {
                actor(Structure::noArgsOp)
            }
            parallel {
                thread {
                    actor(Structure::argsOp, 1)
                    actor(Structure::argsOp, 2)
                }
                thread {
                    actor(Structure::argsOp, 3)
                }
            }
            post {
                actor(Structure::noArgsOp)
            }
        }
        // Report the scenarios even if the test has not failed
        // The custom scenario should be first in the list of scenarios
        .logLevel(LoggingLevel.INFO)
        .check(this::class)
```

Each scenario consists of three optional sections:

- `initial` – operations executed before the parallel part.
- `parallel` – thread definitions. Threads are defined using the `thread` block. The parallel section might contain multiple `thread` blocks.
- `post` – operations executed after the parallel part.

Operations are defined using the `actor(function, arg1, arg2, ...)` function. Operations within a single block are executed sequentially.

Stalled execution detection

Option	Default value	Description
`timeoutMs`	`3000`	Invocation timeout in milliseconds after which Lincheck reports a stalled execution.
`loopBound`	`50`	The number of loop iterations after which Lincheck reports a stalled execution. Increase the value of `loopBound` if Lincheck falsely reports stalled execution for long loops. This option can only be applied to model checking.
`recursionBound`	`20`	The number of recursive calls after which Lincheck reports a stalled execution. The value of `loopIterationsBeforeThreadSwitch` should be less than `loopBound`. This option can only be applied to model checking.

Thread switching in loops

Option	Default value	Description
`loopIterationsBeforeThreadSwitch`	`10`	The number of loop iterations a thread can perform before trying to switch to another thread. The value of `loopIterationsBeforeThreadSwitch` should be less than `loopBound`. This option can only be applied to model checking.

Verification

Option	Default value	Description
`verifierClass`	`LinearizabilityVerifier`	The verifier class used during the verification process: • `LinearizabilityVerifier` • `SerializabilityVerifier` • `QuiescentConsistencyVerifier`
`sequentialSpecification`	Same as the tested data structure.	The sequential version of the tested data structure. This structure is used during the verification process.

Progress guarantees

Option	Default value	Description
`checkObstructionFreedom`	`false`	Set this option to `true` to verify the obstruction-freedom guarantee of the data structure operations. This option can only be applied to model checking.

Library analysis

Option	Default value	Description
`stdLibAnalysisEnabled`	`false`	By default, Lincheck does not verify the behavior of the operations from the standard library, treating them as thread-safe. Set this option to `true` to enable analysis of standard library functions/classes. This option can only be applied to model checking.
`addGuarantee`	–	Define guarantees for thread-safe or irrelevant to analysis methods using the `addGuarantee` option to exclude them from model checking. This option can only be applied to model checking.

Defining a guarantee

To define a guarantee, use a builder chain: select classes, then methods, then the guarantee type.

```
@Test
    fun modelCheckingWithGuaranteesTest() = ModelCheckingOptions()
        .addGuarantee(
            forClasses("java.util.concurrent.ConcurrentHashMap")
                .allMethods()
                .treatAsAtomic()
        )
        .check(this::class)
```

1. Select classes using one of the `forClasses` overloads:
   - `forClasses(vararg fullClassNames: String)` — matches classes if their full name is present in the `fullClassNames` string.
   - `forClasses(vararg classes: KClass<*>)` — matches classes by reference.
   - `forClasses(classPredicate: (fullClassName: String) -> Boolean)` — match classes using a predicate on full class names.
2. Select methods to apply the guarantee to:
   - `methods(methodNames: String)` – matches methods if their name is present in the `methodNames` string.
   - `methods(methodPredicate: (methodName: String) -> Boolean)` – matches methods using a predicate.
   - `allMethods()` – match all methods of the selected classes.
3. Choose the guarantee type:
   - `treatAsAtomic()` — treat each method as an atomic operation. Lincheck will not insert switch points inside the method call, but might add them before or after the call.

     Use `treatAsAtomic()` for methods that are known to be thread-safe.
   - `ignore()` — exclude the methods from analysis. Lincheck will not insert switch points inside, before, or after the method calls.

     > If a method uses synchronization primitives internally (for instance, `synchronized` blocks), ignoring the method may cause Lincheck to deadlock.

     Use `ignore()` for methods that are irrelevant to analysis, such as logging or debugging utilities.

What's next

Learn how to configure argument generation for operations used in Lincheck's execution scenarios.

See also

- Configuring operation execution options
- Checking for non-blocking progress guarantees
- Defining a sequential specification of an algorithm

Argument generation constraints

To test a concurrent data structure, Lincheck generates a set of concurrent scenarios by placing the operations randomly in different threads and invoking them with random arguments.

You can constrain the range of operation arguments to increase the chance of finding a concurrency bug. For example, concurrent operations in a hash map are more likely to access the same key if the range of possible key values is limited. This enables Lincheck to expose race conditions and other concurrency bugs more efficiently.

To limit the range of the generated argument values in Lincheck:

1. Use the `@Param` annotation to declare an argument generator:

```
@Param(name = "key", gen = IntGen::class, conf = "1:2")
class MultiMapTest {
}
```

   - `name` – the name of the argument generator.
   - `gen` – the type of the generator.
   - `conf` – the configuration string for the generator. Here, Lincheck generates integer values from 1 to 2.

   > Lincheck provides generators for multiple value types. Each type uses a different configuration string template.
   >
   > Read more in the Generator types section.

2. Annotate the operations parameters with `@Param` to apply the constraints:

```
@Operation
    fun add(@Param(name = "key") key: Int, value: Int) = map.add(key, value)

    @Operation
    fun get(@Param(name = "key") key: Int) = map.get(key)
```

With the constraints in place, Lincheck generates scenarios using only values within the specified range:

```
| ---------------------------------- |
|    Thread 1     |     Thread 2     |
| ---------------------------------- |
| add(2, 0): void | add(2, -1): void |
| ---------------------------------- |
| get(2): [-1]    |                  |
| ---------------------------------- |
```

Generator types

Lincheck provides the following argument generator types:

Generator	Configuration template	Description
		If the configuration string is empty, uses the full integer range from `Int.MIN_VALUE` to `Int.MAX_VALUE`. Example: `"1:3" -> [1, 2, 3]`

`StringGen`	`"maxWordLength:alphabet"`
	`"maxWordLength"`
	`""`
		Generates random string values up to `maxWordLength` from the provided `alphabet`. The default `alphabet` is `[a-zA-Z\d _]`.
		The default `maxWordLength` is `15`.
		
		Example:
		
		`"2:abc" → [ "a", "b", "c", "aa", "bb", "cc", "ab", "bc", "ac", "ba", "cb", "ca" ]`

`EnumGen`	`"Enum.Const1,Enum.Const2,..."`	Generates a random value from the specified list of enum values. Example: `"Enum.Const1,Enum.Const2" → [ Enum.Const1, Enum.Const2 ]`
`BooleanGen`	`""`	Generates `true` and `false` values. Does not require a specific configuration string. Example: `"" -> [true, false]`

`DoubleGen`	`"start:step:end"`
	`"start:end"`
	`""`
		Generates `Double` values from `start` to `end`, incrementing by `step`.
		
		Default `step` value is `(end - start)/100`.
		
		If the configuration string is empty, generates values from `Int.MIN_VALUE` to `Int.MAX_VALUE` with `step = 0.1`.
		
		Example: `"0.0:0.1:1.0" -> [0.0, 0.1, 0.2, ..., 0.9, 1.0]`

`FloatGen`	`"start:step:end"`
	`"start:end"`
	`""`
		Same as `DoubleGen` with values converted to `Float`.
		
		Example: `"0.0:0.1:1.0" -> [0.0, 0.1, 0.2, ..., 0.9, 1.0]`

Generator	Configuration template	Description
`LongGen`	`"min:max"`	Same as `IntGen` with values converted to `Long`. Example: `"1:3" -> [1, 2, 3]`
`ShortGen`	`"min:max"`	Generates `Short` values between `min` and `max`, inclusive. If the configuration string is empty, uses the full short integer range from `-32768` to `32767`. Example: `"1:3" -> [1, 2, 3]`
`ByteGen`	`"min:max"`	Generates `Byte` values between `min` and `max`, inclusive. If the configuration string is empty, uses the full byte range from `-128` to `127`. Example: `"1:3" -> [1, 2, 3]`
`ThreadIdGen`	`""`	Returns the ID number of the current thread. Does not require a specific configuration string. Example: `"" -> [1, 2]`

What's next

Learn how to restrict certain operations to a single thread in Lincheck.

See also

- Checking for non-blocking progress guarantees
- Defining sequential specification of the algorithm

Operation execution options

Lincheck provides a set of options for controlling how specific operations are executed, such as running operations on a single thread, executing an operation only once, marking the operation as blocking, and others.

In this article, you will learn about different execution options and how to set them.

Single-thread operation groups

Certain operations are required to never run concurrently, such as operations in single-producer single-consumer queues.

To create a group of operations that are never executed in parallel, use the `nonParallelGroup` option when declaring the operations:

```
@Operation(nonParallelGroup = "consumers")
    fun poll(): Int? = queue.poll()

    @Operation(nonParallelGroup = "consumers")
    fun peek(): Int? = queue.peek()

    @Operation(nonParallelGroup = "producer")
    fun offer(x: Int) = queue.offer(x)

    @Operation
    fun isEmpty(): Boolean = queue.isEmpty()
```

Lincheck ensures that the operations from the non-parallel group are never executed in parallel with each other. However, these operations can still run in parallel with operations outside of the non-parallel group:

```
| ------------------- |
| Thread 1  | Thread 2  |
| ------------------- |
| poll()    | offer(0)  |
| peek()    | offer(0)  |
| poll()    | isEmpty() |
| poll()    | isEmpty() |
| isEmpty() | isEmpty() |
| ------------------- |
```

Single-use operations

Use the `runOnce` option to execute the operation only once per test invocation:

```
@Operation(runOnce = true)
    fun singleOp() = struct.singleOp()

    @Operation
    fun regularOp() = struct.regularOp()
```

An example of a generated scenario:

```
| --------------------------- |
| Thread 1      | Thread 2    |
| --------------------------- |
| regularOp()   | singleOp()  |
| regularOp()   | regularOp() |
| --------------------------- |
```

Blocking operations

Use the `blocking` option if the operation is intended to block the execution. If the test checks for non-blocking guarantees, Lincheck does not fail a test when the execution stalls on an operation marked with the `blocking` option:

```
@Operation(blocking = true)
    fun put(key: Int, value: Int) = map.put(key, value)
```

Cancelable operations

Use the `cancellableOnSuspension` option if the operation can be canceled when it suspends:

```
@Operation(cancellableOnSuspension = true)
    suspend fun receive() = ch.receive()
```

Consider the following channel test:

```
@Param(name = "value", gen = IntGen::class, conf = "1:3")
class CancellableOnSuspensionTest {
    private val ch = Channel<Int>()

    @Operation
    suspend fun send(@Param(name = "value") value: Int) = ch.send(value)

    @Operation(cancellableOnSuspension = true)
    suspend fun receive() = ch.receive()

    @Test
    fun test() = ModelCheckingOptions()
        .iterations(50)
        .invocationsPerIteration(1000)
        // Report the scenarios even if the test has not failed
        .logLevel(LoggingLevel.INFO)
        .check(this::class)
}
```

`cancellableOnSuspension = false`

If `receive()` is suspended, it stays suspended for the rest of the scenario:

```
| ------------------------------ |
|      Thread 1    |      Thread 2 |
| ------------------------------ |
| send(3)          | send(3) + cancel |
| receive()        | send(2) + cancel |
| receive()        | receive()        |
| receive()        | receive()        |
| send(1) + cancel | receive()        |
| ------------------------------ |
```

`cancellableOnSuspension = true`

Lincheck explores the scenarios where `receive()` is suspended and then gets canceled, which better models real-life coroutine code:

```
| ------------------------------------- |
|      Thread 1      |      Thread 2      |
| ------------------------------------- |
| send(3)            | send(3) + cancel   |
| receive() + cancel | send(2) + cancel   |
| receive() + cancel | receive()          |
| receive() + cancel | receive() + cancel |
| send(1) + cancel   | receive() + cancel |
| ------------------------------------- |
```

Prompt cancellation

If `cancellableOnSuspension` is enabled and the operation should support prompt cancellation, you can also set `promptCancellation` to `true`:

```
@Operation(cancellableOnSuspension = true, promptCancellation = true)
    suspend fun receive() = ch.receive()
```

Consider the following channel test:

```
@Param(name = "value", gen = IntGen::class, conf = "1:3")
class PromptCancellationTest {
    private val ch = Channel<Int>()

    @Operation
    suspend fun send(@Param(name = "value") value: Int) = ch.send(value)

    @Operation(cancellableOnSuspension = true, promptCancellation = true)
    suspend fun receive() = ch.receive()

    @Test
    fun test() = ModelCheckingOptions()
        .iterations(50)
        .invocationsPerIteration(1000)
        // Report the scenarios even if the test has not failed
        .logLevel(LoggingLevel.INFO)
        .check(this::class)
}
```

`promptCancellation = false`

Lincheck can only try to cancel the operation after it has actually been suspended. If the operation was resumed before Lincheck attempts cancellation, the cancellation fails and the operation completes normally:

```
| ---------------------------------------
|
|      Thread 1      |      Thread 2
|
| ---------------------------------------
|
| send(2)            | send(2)
|
| send(2) + cancel   | receive() + cancel
|
| receive() + cancel | receive()
|
| receive()          | send(3) + cancel
|
| send(2)            | receive()
|
| ---------------------------------------
|
```

`promptCancellation = true`

Lincheck can cancel the operation even if it has already resumed, but didn't have a chance to run. This models the real-life behavior where a coroutine can be canceled between the moment it is resumed and the moment it actually processes the result:

```
| ------------------------------------------- |
|      Thread 1      |         Thread 2       |
| ------------------------------------------- |
| send(2)            | send(2)                |
| send(2) + cancel   | receive() + cancel     |
| receive() + cancel | receive() + prompt_cancel |
| receive()          | send(3) + cancel       |
| send(2)            | receive()              |
| ------------------------------------------- |
```

See also

- Checking for non-blocking progress guarantees
- Defining sequential specification of the algorithm

Progress guarantees

Many concurrent algorithms provide non-blocking progress guarantees, such as wait-freedom, lock-freedom, or obstruction-freedom.

Lincheck only supports verification of obstruction-freedom. However, because lock-free and wait-free algorithms are also obstruction-free, any violation of obstruction-freedom also indicates a violation of those stronger guarantees.

Use the `checkObstructionFreedom` option to verify the obstruction-freedom guarantee of the program:

```
@Test
    fun modelCheckingTest() = ModelCheckingOptions()
        .checkObstructionFreedom()
        .check(this::class)
```

The `checkObstructionFreedom` option is only available for the model checking strategy.

Lincheck verifies obstruction-freedom by checking whether a thread can make progress when all other threads are paused. If a thread's execution gets stuck in a loop, Lincheck reports an active lock.

If certain functions are intentionally blocking, you can mark them with `@Operation(blocking = true)` to prevent false positives.

Example: test `ConcurrentHashMap` for obstruction-freedom

In this example, you will test the `put()` function of a `ConcurrentHashMap` structure.

1. Create a `ConcurrentHashMapTest.kt` file.
2. Create a test class for `ConcurrentHashMap`, declare the `put()` function and a test function with the `checkObstructionFreedom()` option enabled:

   ```
   class ConcurrentHashMapTest {
       private val map = ConcurrentHashMap<Int, Int>()

       @Operation
       fun put(key: Int, value: Int) = map.put(key, value)

       @Test
       fun modelCheckingTest() = ModelCheckingOptions()
           .checkObstructionFreedom()
           .threads(2)
           .actorsPerThread(1)
           .check(this::class)
   }
   ```

   The `threads` and `actorsPerThread` options are used to reduce the number of potential execution scenarios. These options do not change the pass/fail state of the test, but they significantly reduce the testing time.
3. Run the test. It should fail with the following report:

```
= The algorithm should be non-blocking, but an active lock is detected =
| -------------------- |
| Thread 1  | Thread 2  |
| -------------------- |
| put(1, 0) | put(1, 1) |
| -------------------- |

The following interleaving leads to the error:
| ------------------------------------------------------------------------------------------------------
------------ |
|                                          Thread 1                                                    |
Thread 2      |
| ------------------------------------------------------------------------------------------------------
------------ |
| put(1, 0): <hung>                                                                                    |
|
|   map.put(1, 0)                                                                                      |
|
|     putVal(1, 0, false)                                                                              |
|
|       spread(1): 1                                                                                   |
|
|       table → null                                                                                   |
|
|       loop(1 iterations) at ConcurrentHashMap.putVal(ConcurrentHashMap.java:1016)                    |
|
|         <iteration 1>                                                                                |
|
|           initTable()                                                                                |
|
|             loop(1 iterations) at ConcurrentHashMap.initTable(ConcurrentHashMap.java:2293)           |
|
|             table → null                                                                             |
|
|             switch                                                                                   |
|
|                                                                                                      |
put(1, 1): <hung> |
| ------------------------------------------------------------------------------------------------------
------------ |
```

4. Add the `blocking = true` option to the `put()` function annotation:

```
@Operation(blocking = true)
    fun put(key: Int, value: Int) = map.put(key, value)
```

5. Rerun the test. It should pass successfully.

Example: test `ConcurrentSkipListMap` for obstruction-freedom

In this example, you will test the `put()` function of a non-blocking `ConcurrentSkipListMap` structure.

1. Create a `ConcurrentSkipListMapTest.kt` file.
2. Create a test class for `ConcurrentSkipListMap`, declare the `put()` function and a test function with the `checkObstructionFreedom()` option enabled:

```
class ConcurrentSkipListMapTest {
    private val map = ConcurrentSkipListMap<Int, Int>()

    @Operation
    fun put(key: Int, value: Int) = map.put(key, value)

    @Test
    fun modelCheckingTest() = ModelCheckingOptions()
        .checkObstructionFreedom()
        .check(this::class)
}
```

3. Run the test. It should pass successfully.

See also

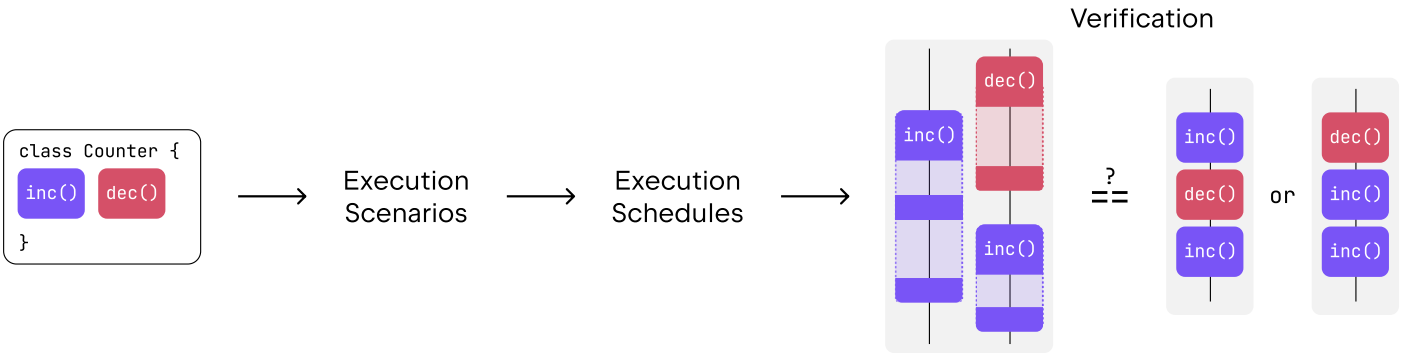
- Configuring argument generation constraints
- Configuring operation execution
- Validating execution results

Results verification and validation

After executing the scenarios generated for a concurrent data structure, Lincheck verifies the results against the specified verification model (for example, linearizability) and optionally checks the final state of the data structure against the user-provided validation function.

Verification

During the verification process, Lincheck tries to find a sequential execution of the operations in the concurrent scenario that achieves the same results as the concurrent execution:



A diagram of the verification process in Lincheck. Lincheck compares a concurrent execution to different sequential executions.

Depending on the verification model, there might be additional restrictions on sequential execution. If no sequential execution matching the verification property can produce the observed results, Lincheck reports an error.

Sequential specification

By default, during the verification process, Lincheck constructs a sequential execution using the operations of the *concurrent* data structure.

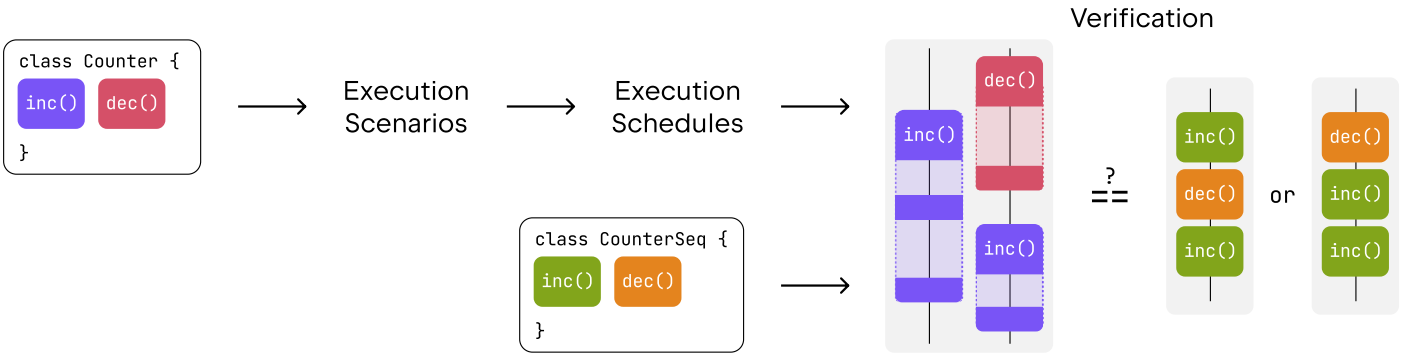
You can specify a sequential data structure with matching operations to:

- Make sure that the concurrent data structure provides the same results as the sequential one.

  Typically, single-threaded implementations are simpler than thread-safe implementations, and therefore can be much more easily verified for correctness (for example, `HashMap` and `ConcurrentHashMap`, `LinkedList` and `ConcurrentLinkedQueue`).

  By comparing the execution results of two versions of a structure, you can make sure that the more complex concurrent structure behaves similarly to a simpler structure in a single-threaded environment.

- Verify both sequential correctness and concurrency safety in a single test.



A diagram of the verification process in Lincheck. Lincheck compares a concurrent execution to different sequential executions. Sequential executions use the operations of the specified sequential version of the structure.

To specify a sequential version of a data structure:

1. Implement a data structure with sequential versions of all concurrent functions tested by Lincheck.
2. Specify the data structure using the `sequentialSpecification()` option:

```
@Test
    fun stressTest() = StressOptions()
        .sequentialSpecification(SequentialQueue::class.java)
        .check(this::class)
```

An example of a Lincheck test that uses a single-threaded `LinkedList` as the sequential specification of `ConcurrentLinkedQueue`:

```
class ConcurrentLinkedQueueTest {
    private val s = ConcurrentLinkedQueue<Int>()

    @Operation
    fun add(value: Int) = s.add(value)

    @Operation
    fun poll(): Int? = s.poll()

    @Test
    fun stressTest() = StressOptions()
        .sequentialSpecification(SequentialQueue::class.java)
        .check(this::class)
}

class SequentialQueue {
    private val s = LinkedList<Int>()

    fun add(x: Int) = s.add(x)
    fun poll(): Int? = s.poll()
}
```

Verification models

By default, Lincheck verifies the results of the concurrent execution against the linearizability model. To apply a different verification model, use the `verifier` option:

```
@Test
    fun customVerifierTest() = ModelCheckingOptions()
        .verifier(SerializabilityVerifier::class.java)
        .check(this::class.java)
```

Lincheck provides the following verifier classes:

- `LinearizabilityVerifier` – the default option. Concurrent execution is valid if there is a sequential execution that preserves the "happens-before" relations among the operations in the concurrent execution.
- `QuiescentConsistencyVerifier` – uses the *quiescent consistency* model which works similarly to the linearizability model, but the "happens-before" constraints are not applied to the operations annotated with `@QuiescentConsistent`:

  ```
  @Operation
  @QuiescentConsistent
  fun someOperation() = { ... }
  ```

  > `QuiescentConsistencyVerifier` does not track actual quiescent points. The verifier can potentially miss bugs that occur across the boundaries of quiescent points.

- `SerializabilityVerifier` – uses the *serializability* model where the concurrent execution is valid if there is some sequential execution (in any order) that leads to the same results as the concurrent execution, regardless of the "happens-before" constraints. It can be used for structures where the relative order of concurrent operations does not matter.

Compare serializability and linearizability

To understand the difference between the two models, see how a data structure can be serializable but not linearizable:

1. Consider the following data structure:

```
class ConcurrentQueue {
    private val elements: MutableList<Int> = ArrayList()

    fun put(x: Int) = synchronized(this) {
        elements += x
    }

    fun poll(): Int? = synchronized(this) {
        if (elements.isEmpty()) return null
        elements.shuffle()
        elements.removeAt(0)
    }
}
```

   This concurrent structure behaves incorrectly: it stores elements like in a typical queue, but returns them randomly.

2. Implement a sequential version of the queue that correctly stores and returns the elements:

```
class CorrectSequentialQueue {
    private val elements: MutableList<Int> = ArrayList()

    fun put(x: Int) {
        elements += x
    }

    fun poll(): Int? = if (elements.isEmpty()) null else elements.removeAt(0)
}
```

3. Create a test class and declare the `put()` and `poll()` operations:

```
@Param(name = "value", gen = IntGen::class, conf = "1:2")
class ConcurrentQueueTest {
    private val q = ConcurrentQueue()

    @Operation
    fun put(@Param(name = "value") x: Int) = q.put(x)

    @Operation
    fun poll(): Int? = q.poll()
}
```

4. Declare and run a serializability test:

```
@Test
    fun serializabilityTest() = ModelCheckingOptions()
        .actorsBefore(0)
        .actorsAfter(0)
        .actorsPerThread(2)
        .threads(2)
        // Use the `SerializabilityVerifier`
        .verifier(SerializabilityVerifier::class.java)
        // Specify the sequential version of the structure
        .sequentialSpecification(CorrectSequentialQueue::class.java)
        .check(this::class.java)
```

It should pass successfully.

5. Declare and run a linearizability test:

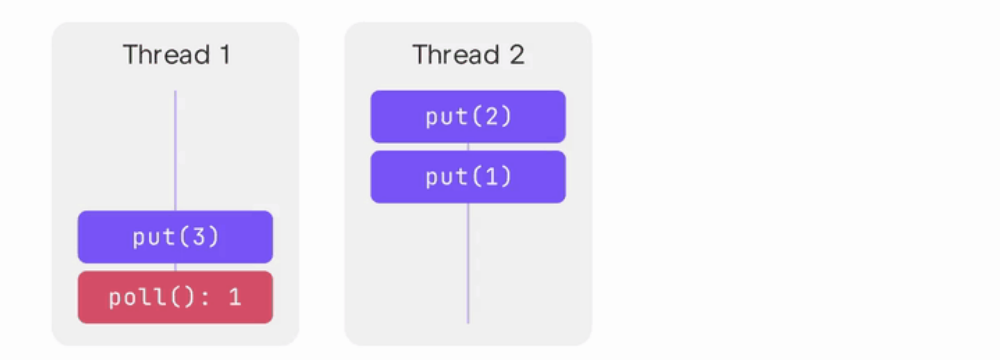
```
@Test
    fun linearizabilityTest() = ModelCheckingOptions()
        .actorsBefore(0)
        .actorsAfter(0)
        .actorsPerThread(2)
        .threads(2)
        // Show the full failed scenario
        .minimizeFailedScenario(false)
        // Specify the sequential version of the structure
        .sequentialSpecification(CorrectSequentialQueue::class.java)
        .check(this::class.java)
```

The test should fail with the following report:

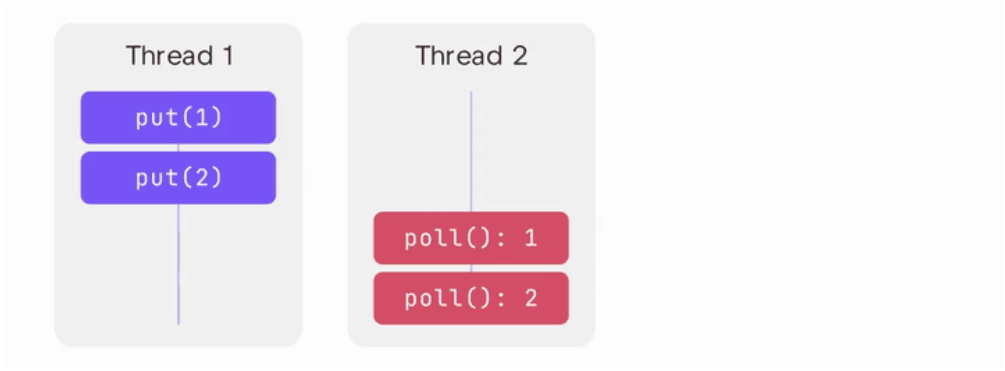
```
| -------------------- |
| Thread 1   | Thread 2 |
| -------------------- |
|            | put(2)   |
|            | put(1)   |
| put(3)     |          |
| poll(): 1  |          |
| -------------------- |
```

6. Analyze the results.

Because the serializability test passes, there exists *some* sequential order of `SequentialQueue` operations that produces `poll(): 1`, for example:



However, linearizability further restricts the operation order: if operation `A` finishes before operation `B` begins in the concurrent execution, then `A` must be executed before `B` in the sequential execution. An example of a linearizable execution would look like this:



Because Lincheck cannot reorder the `put()` operations during verification, it cannot find a sequential execution that complies with linearizability restrictions. This results in a failed test.

Validation

By default, Lincheck does not validate the state of the concurrent data structure after executing a generated scenario. To check the final state, use the `@Validate` annotation on your validation function in the test class:

```
@Validate
    fun validate() {
        // Check some property of the data structure
        // Throw an exception if the check is violated
        check(storage.size ⩾ 2) { "Size must be at least 2, but was ${storage.size}" }
    }
```

The validation function should:

- Accept no arguments.
- Throw an exception if the data structure is in an invalid state.

What's next

- Configuring argument generation constraints
- Configuring operation execution
- Checking for non-blocking progress guarantees

Model checking

To test the code using the model checking strategy, Lincheck inserts explicit thread-switch instructions at points of shared-memory access (`read` and `write`) or at synchronization points, such as lock acquisition and release, `park` / `unpark` , `wait` / `notify` , and others. This approach enables Lincheck to controllably explore the execution schedules of a program and find the ones that lead to incorrect results.

When testing concurrent code using model checking, Lincheck makes sure that the exploration of the execution schedules is:

- Deterministic. Each invocation of a model checking test returns the same result if the input data has not changed.
- Bounded. Each test explores only a limited number of execution schedules. The number of possible execution schedules grows exponentially with the size of the program, and always exploring all of them would significantly increase testing times. You can adjust the limit by changing the value of `invocationsPerIteration`.

Compared to stress testing, model checking enables Lincheck to collect execution traces and guarantees bug reproduction for failed tests. For a more detailed comparison, see the table in the Testing strategies article.

Deterministic exploration

Model checking in Lincheck requires that, given the same input data and execution schedule, the code under test produces the same results. Deterministic execution in model checking is what enables Lincheck to collect execution traces and guarantees bug reproduction for failed tests, which means that any non-deterministic code prevents model checking from working properly.

If an execution produces a non-deterministic result, Lincheck reports an error:

```
Non-determinism found. Probably caused by non-deterministic code (WeakHashMap, Object.hashCode, etc).
== Reporting the first execution without execution trace ==
= Invalid execution results =
| -------- |
| Thread 1 |
| -------- |
| inc(): 2 |
| -------- |

== Reporting the second execution ==
= Invalid execution results =
| -------- |
| Thread 1 |
| -------- |
| inc(): 3 |
| -------- |
```

Some sources of non-determinism are controlled by Lincheck, while others are either restricted for usage or might produce an unexpected test failure.

Controlled sources of non-determinism

When running a test with the model checking strategy, Lincheck controls the following sources of non-determinism:

- **Thread-switching**. Instead of relying on the JVM for thread-switching, Lincheck inserts explicit thread-switch instructions at points of shared-memory access (`read` and `write`) or at synchronization points, such as lock acquisition and release, `park` / `unpark` , `wait` / `notify` , and others.
- **Random number generators**. Lincheck fixes the random seed.
- **Identity hash codes**. Lincheck fixes the identity hash codes of objects.
- **Time API calls**. Lincheck intercepts the time API calls and returns deterministic results.
-

Top-level and `companion object` properties. Lincheck resets the values of top-level `var` properties and `companion object` properties (the Kotlin equivalents of global variables) between invocations in model checking tests:

```
@TestMethodOrder(MethodOrderer.OrderAnnotation::class)
class VariableResetTest {
    companion object {
        private var atomicInt = AtomicInteger(0)
    }

    @Test
    @Order(1)
    fun modelCheckingTest() = Lincheck.runConcurrentTest {
        val t1 = thread { atomicInt.getAndIncrement() }
        val t2 = thread { atomicInt.getAndIncrement() }

        t1.join()
        t2.join()

        check(atomicInt.get() == 2)
    }

    @Test
    @Order(2)
    fun resetAfterModelCheckingTest() {
        // Verify `atomicInt` has been reset to 0 after `modelCheckingTest()`
        check(atomicInt.get() == 0)
    }

    @Test
    @Order(3)
    fun regularIncTest() {
        atomicInt.getAndIncrement()
        check(atomicInt.get() == 1)
    }

    @Test
    @Order(4)
    fun valuePersistsAfterRegularIncTest() {
        // Verify `atomicInt` still holds 1 after `regularIncTest()`
        check(atomicInt.get() == 1)
    }
}
```

Uncontrolled sources of non-determinism

Lincheck controls some sources of non-determinism, but not all. Using non-deterministic code in a way that Lincheck cannot control either prevents you from using Lincheck with this particular part of code or requires workarounds.

Each uncontrolled source of non-determinism is explained in detail in a dedicated section:

- Thread-local variables
- Weak references
- I/O API calls

Bounded exploration

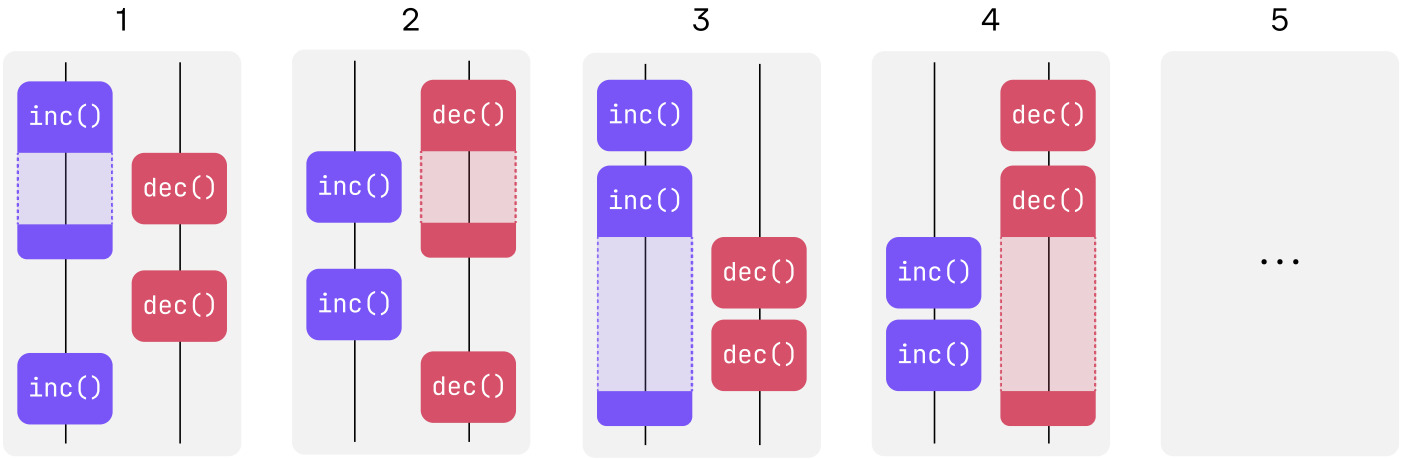
When testing concurrent code, Lincheck runs each execution scenario multiple times – a different execution schedule of a program is explored in each scenario invocation. Because the number of possible execution schedules grows exponentially with the size of the program, the number of invocations for a single execution scenario test is limited to reduce the testing times. If the number of the invocations required to explore all execution schedules exceeds the specified limit, Lincheck stops the exploration.

When Lincheck is unable to analyze *all* execution schedules, it tries to evenly analyze logically different ones:

- Lincheck first explores all schedules with a single preemptive thread switch, then all schedules with two, and so on.
- When choosing the next schedule to explore, Lincheck prioritizes schedules with thread switches in new locations.

Example: schedules with one thread switch

See how Lincheck models execution schedules with a single preemptive thread switch in a two-thread scenario:



A diagram of four execution schedules modeled by Lincheck for the same two-thread scenario with a single thread switch between threads.

Because Lincheck starts by modeling a schedule with a thread switch in the first thread, the next modeled schedule is more likely to have a thread switch in the second thread. This continues until Lincheck either reaches the limit of explored schedules or exhausts all possible schedules.

Known limitations and workarounds

The model checking strategy has the following known limitations.

Relaxed Java memory model

Model checking requires Lincheck to assume a sequentially consistent memory model of the execution.

The relaxed memory model used in Java can introduce bugs related to instruction reordering, memory cache behavior, and other similar effects. With model checking, Lincheck cannot simulate such effects and catch bugs related to them.

Vote for the related issue and track its progress on GitHub.

Most concurrency bugs can be found even under the assumption of a sequentially consistent memory model. However, Lincheck can miss some bugs caused by low-level effects. For example, a missing `@Volatile` modifier might produce a bug caused by store buffer or a compiler reordering, which cannot be caught by Lincheck's model checker:

```
class RelaxedMemoryModelTest {
    var x = 0 // Not @Volatile
    var y = 0 // Not @Volatile

    @Test
    fun modelCheckingTest() = Lincheck.runConcurrentTest {
        thread {
            x = 1
            y = 1
        }
        thread {
            if (y == 1 && x == 0) {
                // Code in this block might be executed on real hardware because of
                // store buffer and compiler reordering.
                // Lincheck cannot model this behavior with model checking.
                error("Unreachable under sequential consistency")
            }
        }
    }
}
```

Workaround

If you want to test concurrent code without the assumption of a sequentially consistent memory model, Lincheck provides a stress testing strategy for concurrent data structures.

Threads created outside the scenario

Lincheck can only track the threads created inside a concurrent scenario. It can miss bugs occurring in externally created threads, such as when using the default dispatcher with coroutines or the common thread pool with Java's `ForkJoinPool`.

> Vote for the related issue and track its progress on GitHub.

Workaround

Use a fixed thread pool:

- As a local coroutines dispatcher
- Instead of the common thread pool used by `ForkJoinPool`

This guarantees that Lincheck can track the lifecycle and activity of threads in a concurrent scenario:

As a local coroutine dispatcher

```
class FixedThreadPoolDispatcherTest {
    @Test
    fun test() = Lincheck.runConcurrentTest {
        val dispatcher = Executors.newFixedThreadPool(nThreads).asCoroutineDispatcher()
        dispatcher.use {
            runBlocking(dispatcher) {
                val counter = AtomicInteger(0)
                val coro = launch {
                    while (isActive) { counter.getAndIncrement() }
                }
                coro.cancel()
                coro.join()
            }
        }
    }
}
```

Instead of the common thread pool

```
class FixedThreadPoolExecutorServiceTest {
    @Test
    fun test() = Lincheck.runConcurrentTest {
        val executorService = Executors.newFixedThreadPool(nThreads)
        try {
            val counter = AtomicInteger(0)
            val task = Runnable { counter.getAndIncrement() }
            val future1 = executorService.submit(task)
            val future2 = executorService.submit(task)
            future1.get()
            future2.get()
        } finally {
            executorService.shutdown()
        }
    }
}
```

Thread-local variables

Lincheck does not reset thread-local variables during multiple invocations of the same scenario (unlike it does with top-level `var` properties and `companion object` properties). This leads to inconsistencies between the runs of the same test.

Vote for the related issue and track its progress on GitHub.

Example:

```
class ThreadLocalVariableTest {
    @Test
    fun modelCheckingTest() = Lincheck.runConcurrentTest {
        var counter = getLocalCounter()
        var t = thread { counter.getAndIncrement() }
        t.join()
        check(counter.get() == 1)
    }

    private fun getLocalCounter() = localCounter.get()
}

// Using ThreadLocal to create a variable leads to a failed test
private val localCounter: ThreadLocal<AtomicInteger> = ThreadLocal.withInitial {
    AtomicInteger(0)
}
```

This test fails with an error because the value of the counter accumulates across scenario invocations:

```
| ------------------------------------------------------------------------------------- |
|                   Main Thread                    |                Thread 1              |
| ------------------------------------------------------------------------------------- |
| getLocalCounter(): AtomicInteger#1               |                                     |
| thread(block = Lambda#1): Thread#1               |                                     |
| switch (reason: waiting for Thread 1 to finish)  |                                     |
|                                                  | run()                               |
|                                                  |   counter => AtomicInteger#1        |
|                                                  |   AtomicInteger#1.getAndIncrement(): 2 |
| Thread#1.join()                                  |                                     |
| counter.element => AtomicInteger#1               |                                     |
| AtomicInteger#1.get(): 3                         |                                     |
| ------------------------------------------------------------------------------------- |
```

Workaround

Create thread-local variables manually by storing values in a `ConcurrentHashMap` with thread IDs as keys:

```
class ThreadLocalVariableWorkaroundTest {
    var threadLocalCounters = ConcurrentHashMap<Long, AtomicInteger>()

    @Test
    fun modelCheckingTest() = Lincheck.runConcurrentTest {
        var counter = getLocalCounter()
        var t = thread { counter.getAndIncrement() }
        t.join()
        check(counter.get() == 1)
    }

    private fun getLocalCounter() = threadLocalCounters.computeIfAbsent(Thread.currentThread().id) {
        AtomicInteger(0)
    }
}
```

Because `threadLocalCounters` is a top-level `var` property, Lincheck resets it between invocations, avoiding the accumulation problem.

Weak references

Lincheck does not control when a garbage collector removes objects that are only referenced by weak references. Calling `get()` on such objects produces non-deterministic results. The test might pass successfully despite the use of weak references, but if Lincheck encounters an inconsistency between the runs of the same test, it raises a non-determinism error.

Vote for the related issue and track its progress on GitHub.

Time API calls

Lincheck simulates calls of `java.lang.System.nanoTime()` and `java.lang.System.currentTimeMillis()` by always returning a predefined constant to prevent inconsistencies between the runs of the same test.

This approach might not correctly simulate timeouts, elapsed-time comparisons, rate-limiting, or other logic that depends on elapsed time.

Vote for the related issue and track its progress on GitHub.

I/O API calls

Lincheck does not support calls to I/O APIs, including operations on files and sockets, to prevent inconsistencies between the runs of the same test.

Vote for the related issue and track its progress on GitHub.

Calling I/O APIs leads to `java.lang.IllegalStateException`:

```
class FilesCreateTempFileTest {
    @Operation
    fun operation(): List<String> = List(10) {
        val tempFile = Files.createTempFile("test-prefix", ".txt")
        require(Files.exists(tempFile)) { "File was not created: $tempFile" }
        tempFile.toString()
    }

    // The test fails with the following error message:
    // "java.lang.IllegalStateException: File operations are not supported in Lincheck"
    @Test
    fun modelChecking() = ModelCheckingOptions().check(this::class)
}
```

Virtual threads

Support for virtual threads has not been verified and is not guaranteed. Consider using platform threads for scenarios requiring model checking until support is verified.

Vote for the related issue and track its progress on GitHub.

See also

- Test any concurrent code with model checking
- Use model checking to test a concurrent data structure
- Configure a Lincheck testing strategy
- Compare model checking to stress testing

Lincheck in Kotlin Multiplatform projects

Lincheck can be used with Kotlin Multiplatform to test the code targeting the JVM. This article will guide you through setting up Lincheck tests in a Kotlin Multiplatform project.

You will:

- Set up your project to run Lincheck tests for targets using the JVM.
- Create classes for counter data structures shared between all targets.
- Create tests for the counters shared between all targets and tests specific to the JVM.
- Run the tests.

Set up a project

1. Create a new Kotlin Multiplatform project in IntelliJ IDEA. When creating a project, make sure that:
   - The **Include tests** checkbox is checked.
   - Your project has a JVM-based (a server, desktop, or Android) target. You can verify that your code compiles for the JVM by checking the `shared/build.gradle.kts` or `core/build.gradle.kts` file:

```
kotlin {
    // ...

    jvm()

    // ...

    // If you have an Android target, the use of JVM is
    // configured in the `androidLibrary` section
    androidLibrary {
        // ...

        compilerOptions {
            jvmTarget = JvmTarget.JVM_11
        }

    // ...

    }
}
```

Server/Desktop + Android and **Server/Desktop + Android + iOS** projects

Kotlin currently doesn't support sharing the code between JVM and Android targets. You will not be able to create shared tests for your platforms, but you can still test the code with Lincheck using platform-specific tests.

For more information about the general test setup for Lincheck, read the Getting started with Lincheck article.

2. Set up the tests folder and Lincheck dependency according to the target platforms in your project:

- **Desktop/Android + iOS**
- **Server + iOS, Server + Desktop, Server + Desktop + iOS**

Desktop/Android + iOS

The directory for the platform-specific tests is created by the Kotlin Multiplatform plugin automatically. For example, for a desktop target it is `shared/src/jvmTest/kotlin`.

Add a Lincheck dependency to the platform-specific test source set and a `kotlin.test` dependency to the `commonTest` source set:

Desktop

```
// shared/build.gradle.kts
    kotlin {
        sourceSets {
            commonTest.dependencies {
                implementation(libs.kotlin.test)
            }
            jvmTest {
                implementation("org.jetbrains.lincheck:lincheck:3.7")
            }
        }
    }
```

Android

```
// shared/build.gradle.kts
    kotlin {
        sourceSets {
            commonTest.dependencies {
                implementation(libs.kotlin.test)
            }
            getByName("androidHostTest").dependencies {
                implementation("org.jetbrains.lincheck:lincheck:3.7")
            }
        }
    }
```

Server + iOS, Server + Desktop, Server + Desktop + iOS

1. Create a `core/src/jvmTest/kotlin` directory. Gradle will run the tests in this directory for all platforms that use the JVM.

   Projects with a desktop target also have a `shared/src/jvmTest/kotlin` directory which you can use instead. However, it is recommended to place the code relevant to both the server and desktop/iOS targets in the `core` directory.

   If you have a desktop target and create a `core/src/jvmTest/kotlin` directory, delete the `shared/src/jvmTest/kotlin` directory.

2. Add a Lincheck dependency to the `jvmTest` source set and a `kotlin.test` dependency to the `commonTest` source set:

```
// core/build.gradle.kts
kotlin {
    // ...
    sourceSets {
        // ...
        commonTest.dependencies {
            implementation(libs.kotlin.test)
        }
        jvmTest.dependencies {
            implementation("org.jetbrains.lincheck:lincheck:3.7")
        }
    }
}
```

Create shared classes

Create classes implementing counter data structures that are shared between all targets:

1. Create an `UnsafeCounter.kt` file in the `core/src/commonMain` or `shared/src/commonMain` directory:

```
class UnsafeCounter {
    private var value: Int = 0

    fun inc() = value++

    fun get(): Int = value
}
```

2. Create a `SafeCounter.kt` file in the same directory:

```
import kotlin.concurrent.atomics.AtomicInt
import kotlin.concurrent.atomics.ExperimentalAtomicApi

@OptIn(ExperimentalAtomicApi::class)
class SafeCounter {
    private var value = AtomicInt(0)

    fun inc() = value.addAndFetch(1)

    fun get(): Int = value.load()
}
```

Write shared tests

Write tests that are run for all targets:

1. Create an `UnsafeCounterTest.kt` file in the `core/src/commonTest` or `shared/src/commonTest` directory:

```
import kotlin.test.Test
import kotlin.test.assertEquals

class UnsafeCounterTest {
    @Test
    fun testIncrement() {
        val counter = UnsafeCounter()
        assertEquals(0, counter.get(), "Initial value should be 0")
        counter.inc()
        assertEquals(1, counter.get(), "Value after one increment should be 1")
    }
}
```

2. Create a `SafeCounterTest.kt` file in the same directory:

```
import kotlin.test.Test
import kotlin.test.assertEquals

class SafeCounterTest {
    @Test
    fun testIncrement() {
        val counter = SafeCounter()
        assertEquals(0, counter.get(), "Initial value should be 0")
        counter.inc()
        assertEquals(1, counter.get(), "Value after one increment should be 1")
    }
}
```

Write JVM-specific tests

Write tests that are only run for platforms targeting the JVM:

1. Create an `UnsafeCounterConcurrentTest.kt` file in the test directory for JVM-specific tests:
   - Android + iOS – `shared/src/androidHostTest/kotlin`
   - Desktop + iOS – `shared/src/jvmTest/kotlin`
   - Any project with a server target – `core/src/jvmTest/kotlin`

```
import org.jetbrains.lincheck.datastructures.ModelCheckingOptions
import org.jetbrains.lincheck.datastructures.Operation
import kotlin.test.Test

class UnsafeCounterConcurrentTest {
    private val c = UnsafeCounter()

    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test
    fun modelCheckingTest() {
        ModelCheckingOptions().check(this::class)
    }
}
```

2. Create a `SafeCounterConcurrentTest.kt` file in the same directory:

```
import org.jetbrains.lincheck.datastructures.ModelCheckingOptions
import org.jetbrains.lincheck.datastructures.Operation
import kotlin.test.Test

class SafeCounterConcurrentTest {
    private val c = SafeCounter()

    @Operation
    fun inc() = c.inc()

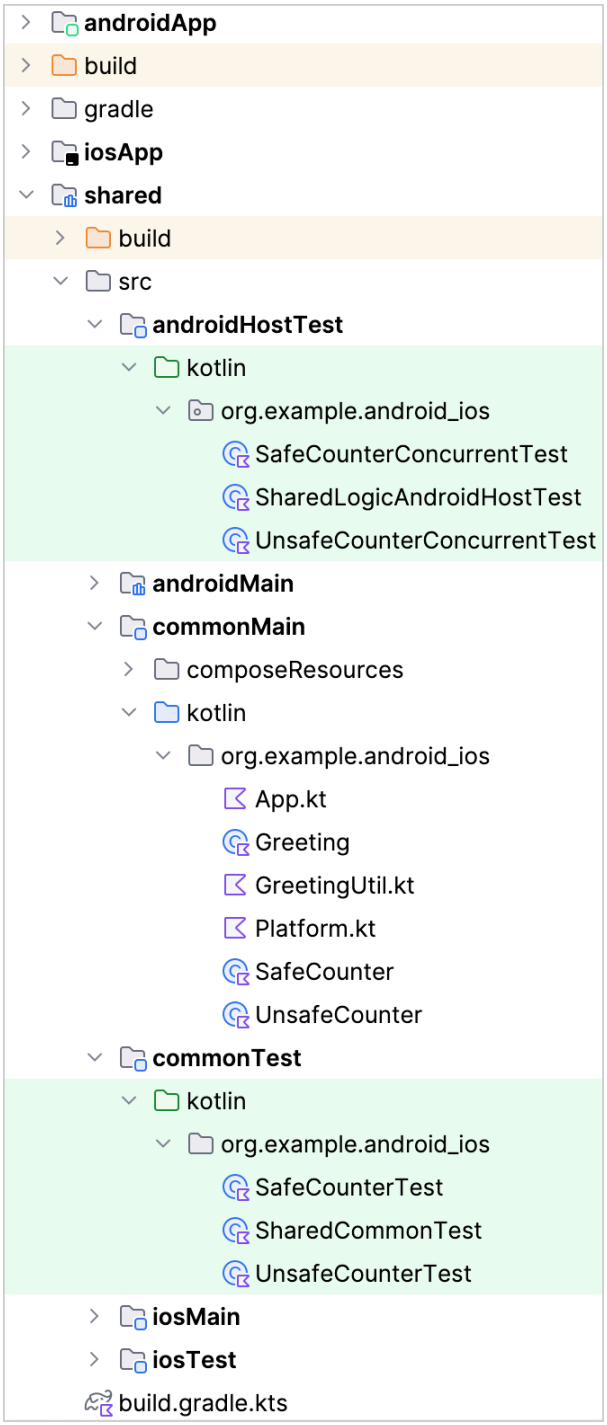
    @Operation
    fun get() = c.get()

    @Test
    fun modelCheckingTest() {
        ModelCheckingOptions().check(this::class)
    }
}
```

Run the tests

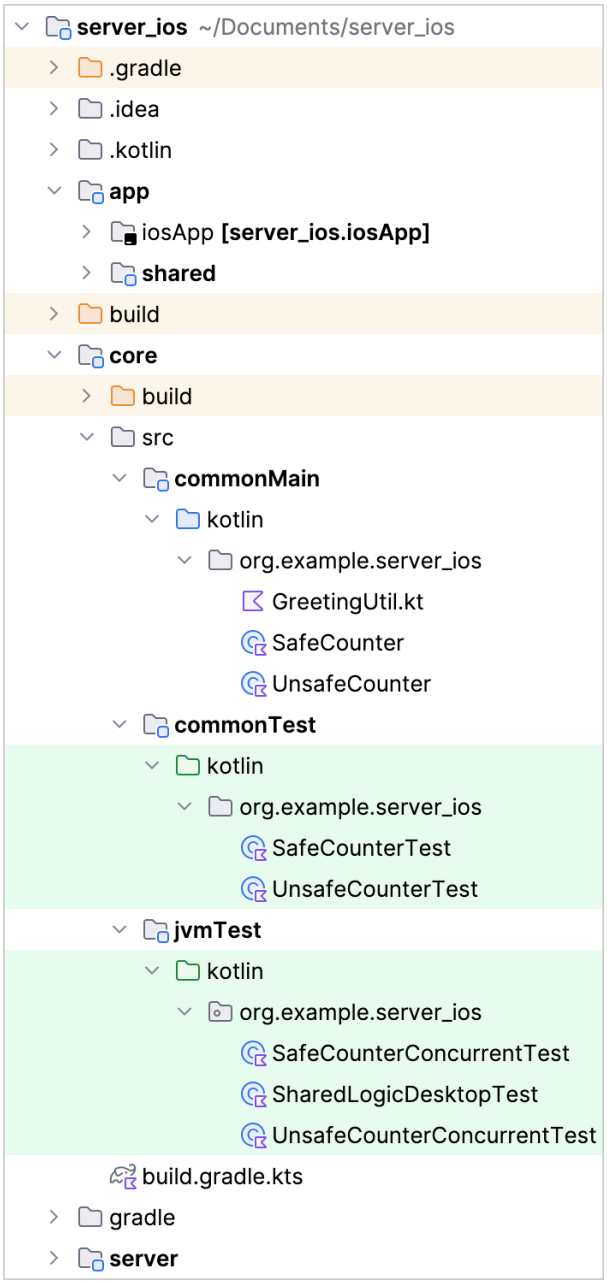
At this point, your project should have the shared implementation of the counter data structures, the shared tests, and the JVM-specific tests for counters. The directory structure of your project should look like this:

Android + iOS



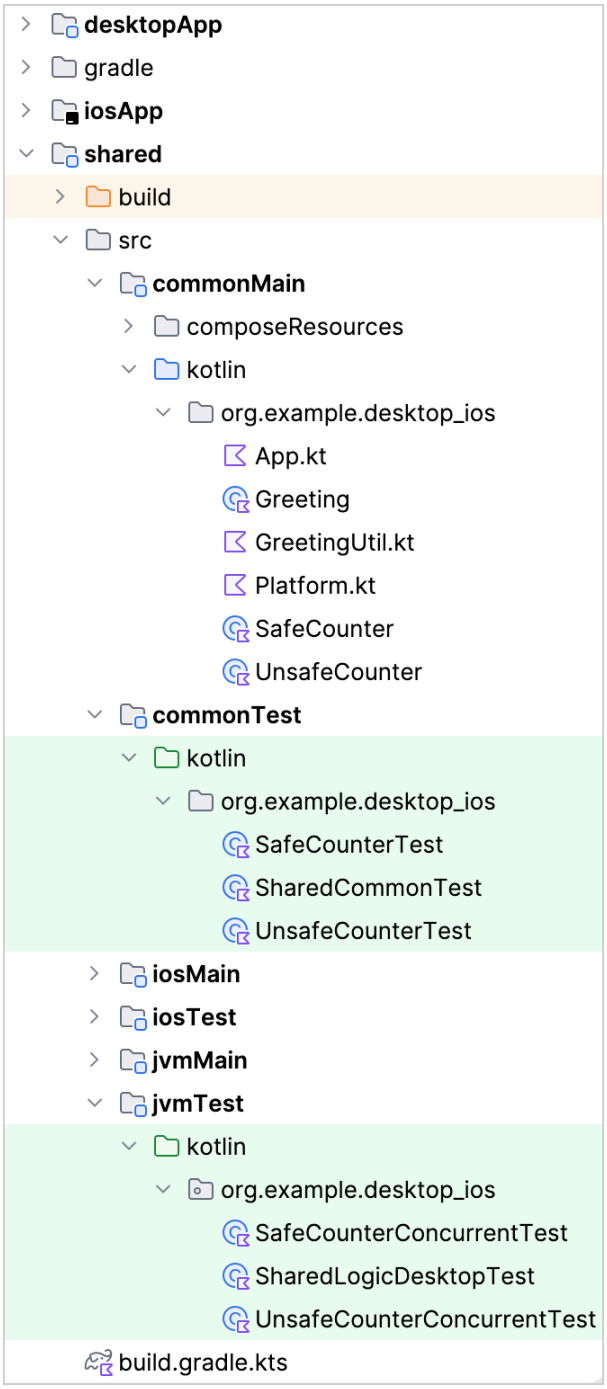
A screenshot of the project file structure for the Android and iOS platforms.

Server + iOS



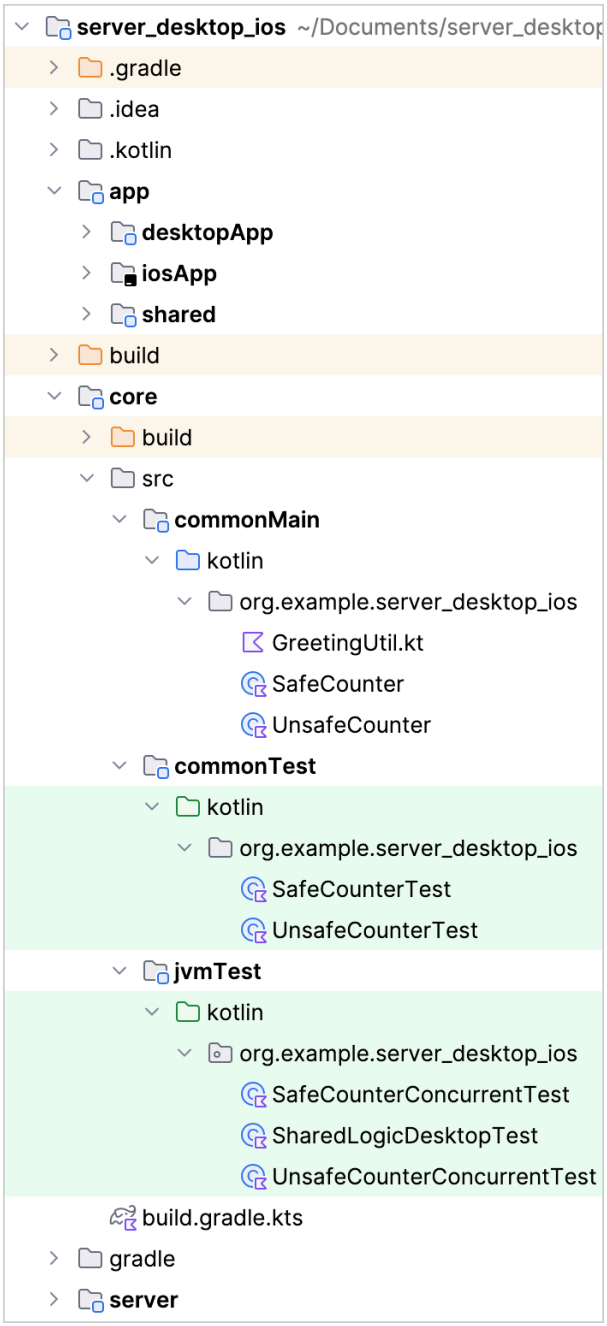
A screenshot of the project file structure for the server and iOS platforms.

Desktop + iOS



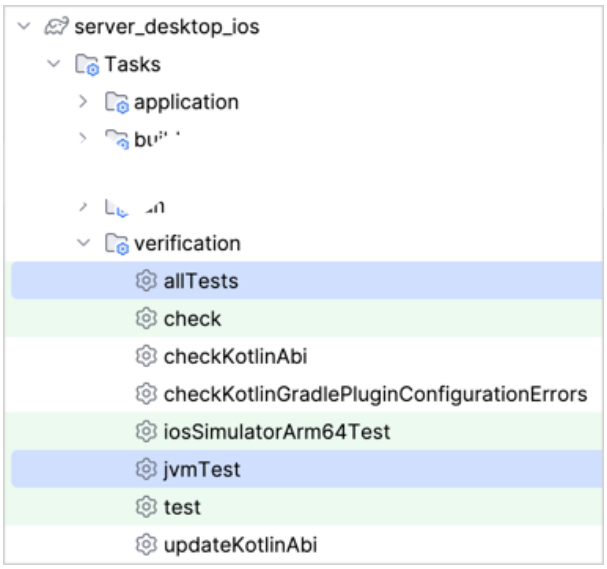
A screenshot of the project file structure for the desktop and iOS platforms.

Server + Desktop + iOS



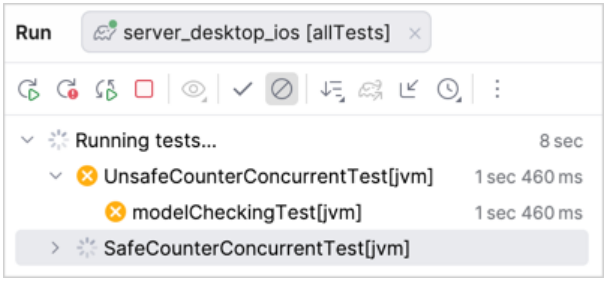
A screenshot of the project file structure for the server, desktop, and iOS platforms.

You can run individual tests from the context menu, using the shortcut, or by running a Gradle task. If you run the `allTests` Gradle task, every test in your project will be executed with the corresponding test runner:



A screenshot of the Gradle plugin in IntelliJ IDEA with the Tasks | Verification | allTests task highlighted.

For example, the JVM-specific tests are executed with the JVM test runner:



A screenshot of the test execution report. The Lincheck tests are marked with a jvm tag.

Support for platforms other than the JVM

Lincheck is unlikely to support other platforms in the future.

Lincheck relies heavily on JVM for bytecode manipulation, which makes model checking challenging to implement on other platforms. Moreover, using model checking for multiple platforms at the same time provides little value because Lincheck tests the code in a deterministic sandbox environment, and the results of the tests will be the same regardless of the platform.

See also

- Learn more about Kotlin Multiplatform
- Explore Lincheck documentation

API references

Welcome to the Kotlin API References page. Here you'll find links to the API documentation for the official Kotlin libraries and tools.

If you're looking for Kotlin Multiplatform libraries, browse them on klibs.io.

Standard library (stdlib)

The Kotlin Standard library provides core functionality for Kotlin programming, including essential APIs such as collections, text and string processing, ranges, sequences, and more. It extends platform-specific APIs and provides a Kotlin-first API for working with them.

kotlin

Test library (kotlin.test)

A multiplatform testing library that provides common testing annotations and utility functions. It supports integration with popular test frameworks on each platform and offers a unified testing experience across the Kotlin ecosystem.

kotlin

Coroutines (kotlinx.coroutines)

A powerful library for asynchronous programming using Kotlin coroutines. It offers tools to support structured concurrency, asynchronous streams, synchronization primitives like mutexes and semaphores, testing, and more.

kotlinx.coroutines

Serialization (kotlinx.serialization)

A multiplatform serialization library. It provides a type-safe, compile-time mechanism for converting Kotlin objects to and from a variety of formats such as JSON, CBOR, and Protocol Buffers.

kotlinx.serialization

Kotlin I/O library (kotlinx-io)

A multiplatform library for low-level I/O operations. It defines abstractions for reading from and writing to binary streams and buffers, designed to be efficient and portable across all Kotlin platforms.

kotlinx-io

Date and time (kotlinx-datetime)

A multiplatform library for calendar-based computations. It provides representations of date values and supports timezone-specific operations.

kotlinx-datetime

Immutable collections (kotlinx.collections.immutable)

A multiplatform library providing immutable and persistent collection interfaces and implementations. It offers efficient copy-on-write operations that share structure between versions, so updating a collection doesn't copy the whole thing.

kotlinx.collections.immutable

Kotlin Gradle plugins (kotlin-gradle-plugin)

The Kotlin Gradle plugins for compiling, testing, and packaging Kotlin code. These plugins simplify JVM and multiplatform builds, manage dependencies, and integrate with IDEs and CI systems.

kotlin-gradle-plugin

Ktor

A framework for building asynchronous clients and servers in connected systems using Kotlin. Ktor is designed for scalability and flexibility and is deeply integrated with coroutines for non-blocking I/O and structured concurrency.

ktor

JVM Metadata (kotlin-metadata-jvm)

A library for reading and writing Kotlin metadata stored in JVM class files. It is primarily used by tools such as annotation processors, static analyzers, and compiler plugins.

kotlinx-metadata

Compose Multiplatform Material3

A multiplatform library for building user interfaces with Material Design 3 components. The API reference includes a gallery of Material 3 components where you can preview the composables.

compose-material3

Gradle

Gradle is a build system that helps to automate and manage your building process. It downloads required dependencies, packages your code, and prepares it for compilation. Learn about Gradle basics and specifics on the Gradle website.

You can set up your own project with these instructions for different platforms or pass a small step-by-step tutorial that will show you how to create a simple backend "Hello World" application in Kotlin.

> You can find information about the compatibility of Kotlin, Gradle, and Android Gradle plugin versions here.

In this chapter, you can also learn about:

- Compiler options and how to pass them.
- Incremental compilation, caches support, build reports, and the Kotlin daemon.
- Support for Gradle plugin variants.

What's next?

Learn about:

- **Gradle Kotlin DSL**. The Gradle Kotlin DSL is a domain specific language that you can use to write build scripts quickly and efficiently.
- **Annotation processing**. Kotlin supports annotation processing via the Kotlin Symbol processing API.
- **Generating documentation**. To generate documentation for Kotlin projects, use Dokka; please refer to the Dokka README for configuration instructions. Dokka supports mixed-language projects and can generate output in multiple formats, including standard Javadoc.
- **OSGi**. For OSGi support, see the Kotlin OSGi page.

Get started with Gradle and Kotlin/JVM

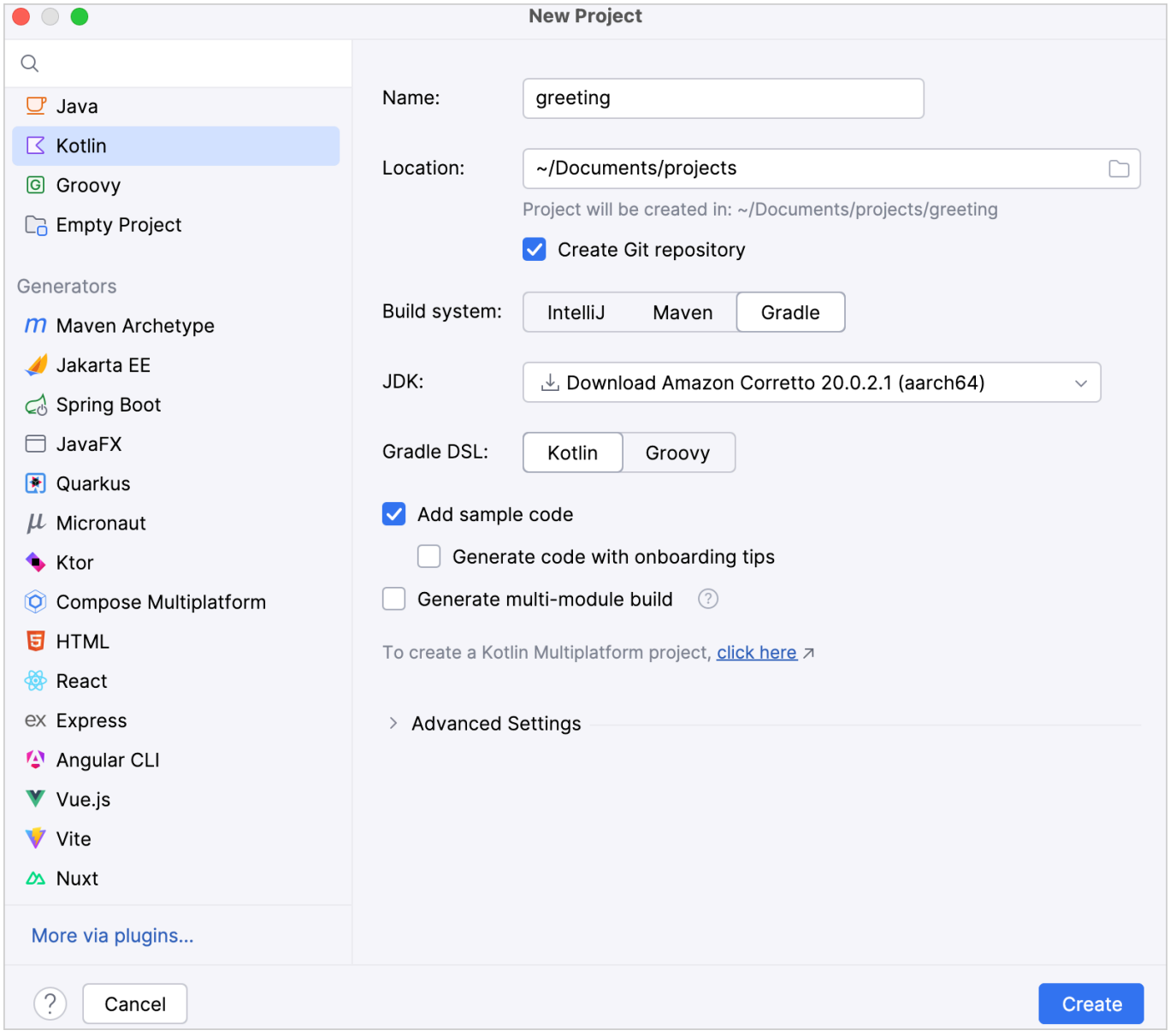
This tutorial demonstrates how to use IntelliJ IDEA and Gradle to create a JVM console application.

To get started, first download and install the latest version of IntelliJ IDEA.

Create a project

1. In IntelliJ IDEA, select **File** | **New** | **Project**.
2. In the panel on the left, select **Kotlin**.
3. Name the new project and change its location, if necessary.

> Select the **Create Git repository** checkbox to place the new project under version control. You will be able to do it later at any time.



Create a console application

4. Select the **Gradle** build system.

5. From the **JDK** list, select the JDK that you want to use in your project.
    - If the JDK is installed on your computer, but not defined in the IDE, select **Add JDK** and specify the path to the JDK home directory.
    - If you don't have the necessary JDK on your computer, select **Download JDK**.
6. Select the **Kotlin** DSL for Gradle.
7. Select the **Add sample code** checkbox to create a file with a sample `"Hello World!"` application.

    > You can also enable the **Generate code with onboarding tips** option to add some additional useful comments to your sample code.

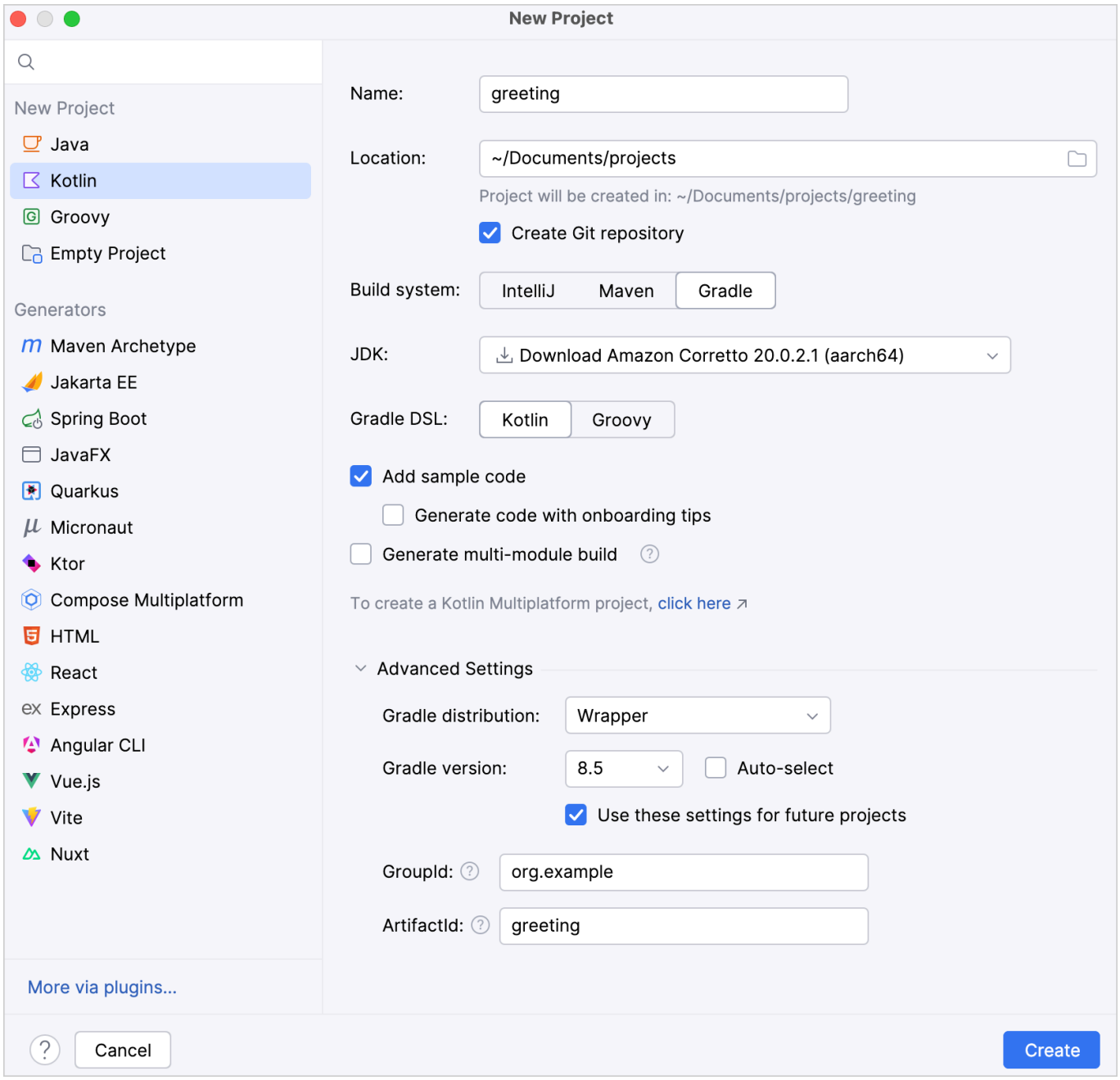
8. Click **Create**.

You have successfully created a project with Gradle!

Specify a Gradle version for your project

You can explicitly specify a Gradle version for your project under the **Advanced Settings** section, either by using the Gradle Wrapper or a local installation of Gradle:

- **Gradle Wrapper:**
    1. From the **Gradle distribution** list, select **Wrapper**.
    2. Disable the **Auto-select** checkbox.
    3. From the **Gradle version** list, select your Gradle version.
- **Local installation:**
    1. From the **Gradle distribution** list, select **Local installation**.
    2. For **Gradle location**, specify the path of your local Gradle version.



Advanced settings

Explore the build script

Open the `build.gradle.kts` file. This is the Gradle Kotlin build script, which contains Kotlin-related artifacts and other parts required for the application:

```
plugins {
    kotlin("jvm") version "2.4.20" // Kotlin version to use
}

group = "org.example" // A company name, for example, `org.jetbrains`
version = "1.0-SNAPSHOT" // Version to assign to the built artifact

repositories { // Sources of dependencies. See 1️⃣
    mavenCentral() // Maven Central Repository. See 2️⃣
}

dependencies { // All the libraries you want to use. See 3️⃣
    // Copy dependencies' names after you find them in a repository
    testImplementation(kotlin("test")) // The Kotlin test library
}

tasks.test { // See 4️⃣
    useJUnitPlatform() // JUnitPlatform for tests. See 5️⃣
}
```

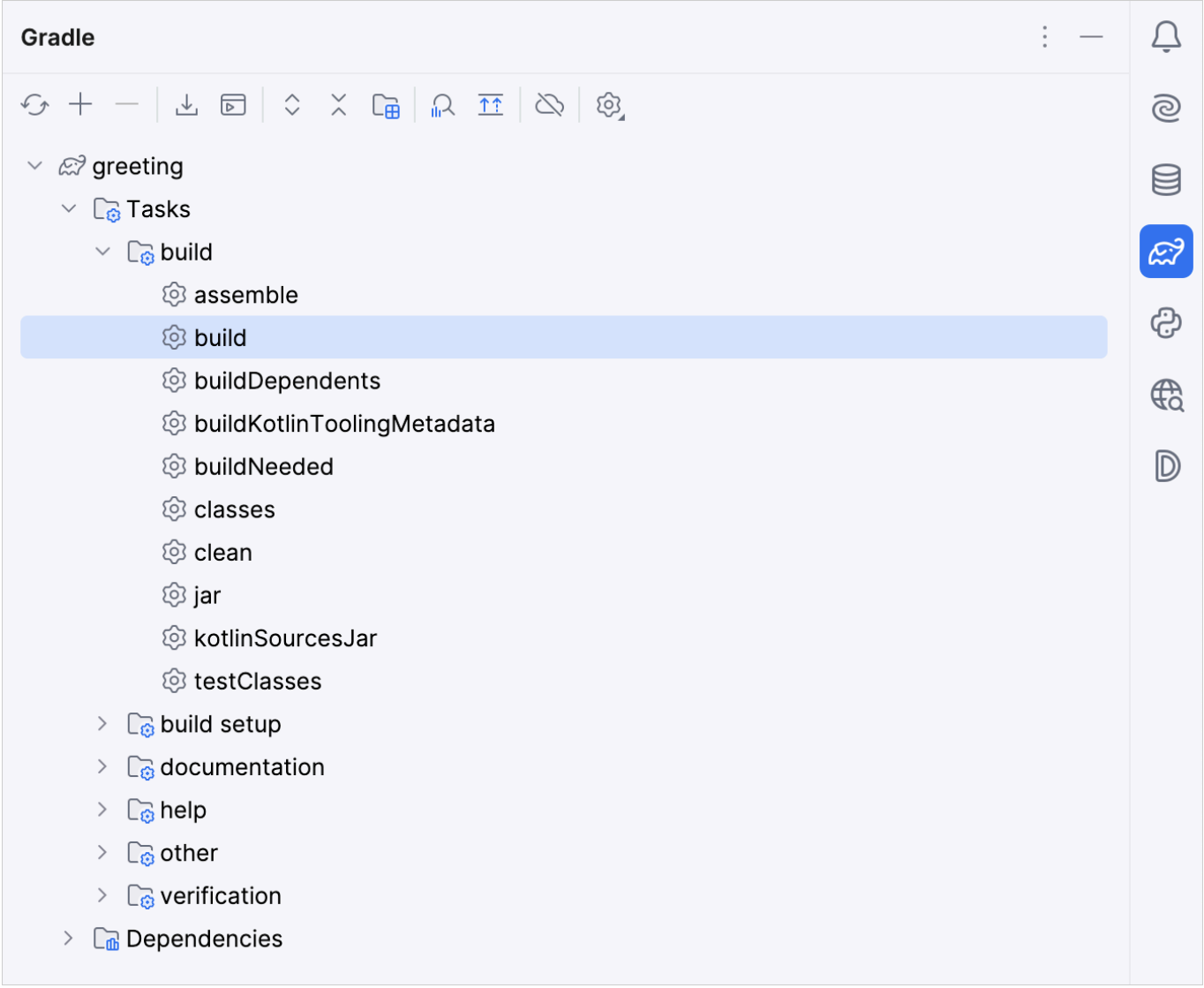
- 1️⃣ Lean more about sources of dependencies.
- 2️⃣ The Maven Central Repository. It can also be Google's Maven repository or your company's private repository.
- 3️⃣ Learn more about declaring dependencies.
- 4️⃣ Learn more about tasks.
- 5️⃣ JUnitPlatform for tests.

As you can see, there are a few Kotlin-specific artifacts added to the Gradle build file:

1. In the `plugins {}` block, there is the `kotlin("jvm")` artifact. This plugin defines the version of Kotlin to be used in the project.
2. In the `dependencies {}` block, there is `testImplementation(kotlin("test"))`. Learn more about setting dependencies on test libraries.

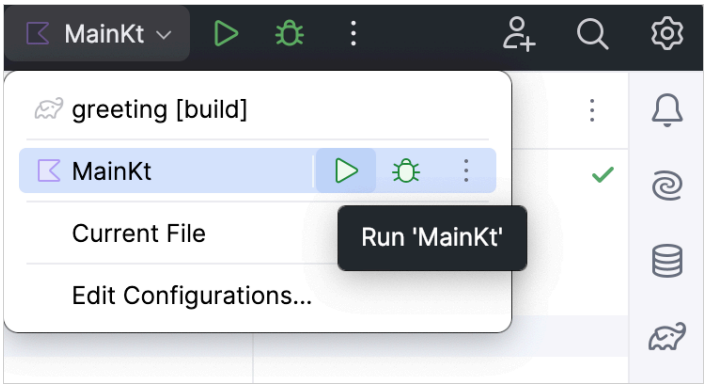
Run the application

1. Open the Gradle window by selecting **View** | **Tool Windows** | **Gradle**:



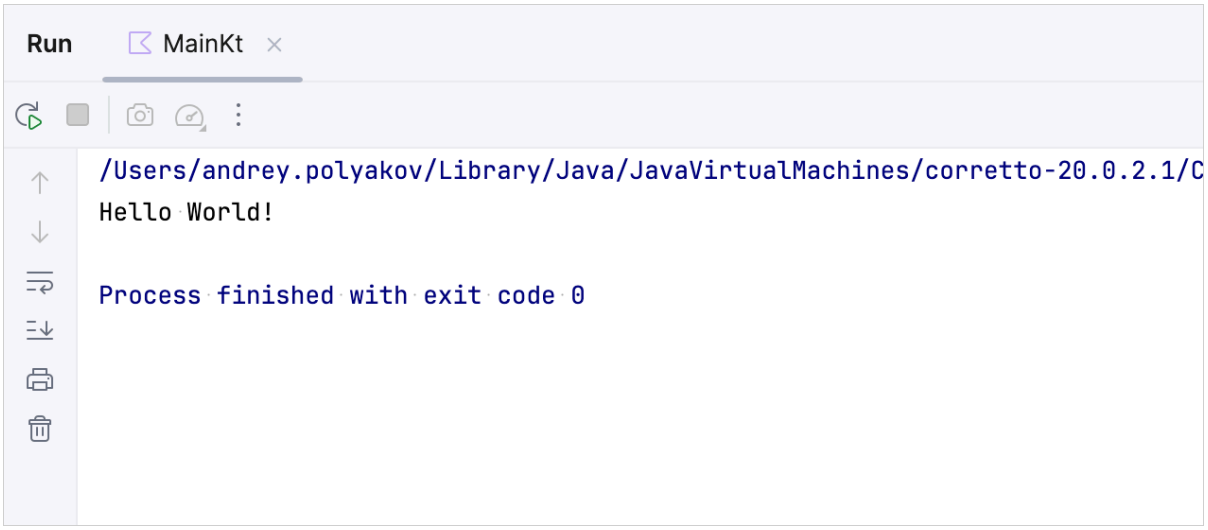
Main.kt with main fun

2. Execute the **build** Gradle task in `Tasks\build\`. In the **Build** window, `BUILD SUCCESSFUL` appears. It means that Gradle built the application successfully.
3. In `src/main/kotlin`, open the `Main.kt` file:
   - `src` directory contains Kotlin source files and resources.
   - `Main.kt` file contains sample code that will print `Hello World!`.
4. Run the application by clicking the green **Run** icon in the gutter and select **Run 'MainKt'**.



Running a console app

You can see the result in the **Run** tool window:



Kotlin run output

Congratulations! You have just run your first Kotlin application.

What's next?

Learn more about:

- Gradle build file properties.
- Targeting different platforms and setting library dependencies.
- Compiler options and how to pass them.
- Incremental compilation, caches support, build reports, and the Kotlin daemon.

Configure a Gradle project

To build a Kotlin project with Gradle, you need to add the Kotlin Gradle plugin to your build script file `build.gradle(.kts)` and configure the project's dependencies there.

To learn more about the contents of a build script, visit the Explore the build script section.

Apply the plugin

To apply the Kotlin Gradle plugin, use the `plugins{}` block from the Gradle plugins DSL:

Kotlin

```
plugins {
    // Replace `<...>` with the plugin name appropriate for your target environment
    kotlin("<...>") version "2.4.20"
    // For example, if your target environment is JVM:
    // kotlin("jvm") version "2.4.20"
}
```

Groovy

```
plugins {
    // Replace `<...>` with the plugin name appropriate for your target environment
    id 'org.jetbrains.kotlin.<...>' version '2.4.20'
    // For example, if your target environment is JVM:
    // id 'org.jetbrains.kotlin.jvm' version '2.4.20'
}
```

The Kotlin Gradle plugin (KGP) and Kotlin share the same version numbering.

When configuring your project, check the Kotlin Gradle plugin (KGP) compatibility with available Gradle versions. In the following table, there are the minimum and maximum **fully supported** versions of Gradle and Android Gradle plugin (AGP):

KGP version	Gradle min and max versions	AGP min and max versions
2.4.20	7.6.3–9.7.0	8.5.2–9.3.1
2.4.0-2.4.10	7.6.3–9.5.0	8.5.2–9.1.0
2.3.20–2.3.21	7.6.3–9.3.0	8.2.2–9.0.0
2.3.10	7.6.3–9.0.0	8.2.2–9.0.0
2.3.0	7.6.3–9.0.0	8.2.2–8.13.0
2.2.20–2.2.21	7.6.3–8.14	7.3.1–8.11.1
2.2.0–2.2.10	7.6.3–8.14	7.3.1–8.10.0
2.1.20–2.1.21	7.6.3–8.12.1	7.3.1–8.7.2
2.1.0–2.1.10	7.6.3–8.10*	7.3.1–8.7.2
2.0.20–2.0.21	6.8.3–8.8*	7.1.3–8.5

KGP version	Gradle min and max versions	AGP min and max versions
2.0.0	6.8.3–8.5	7.1.3–8.3.1
1.9.20–1.9.25	6.8.3–8.1.1	4.2.2–8.1.0

*Kotlin 2.0.20–2.0.21 and Kotlin 2.1.0–2.1.10 are fully compatible with Gradle up to 8.6. Gradle versions 8.7–8.10 are also supported, with only one exception: If you use the Kotlin Multiplatform Gradle plugin, you may see deprecation warnings in your multiplatform projects calling the `withJava()` function in the JVM target. For more information, see Java source sets created by default.

You can also use Gradle and AGP versions up to the latest releases, but if you do, keep in mind that you might encounter deprecation warnings or some new features might not work.

For example, the Kotlin Gradle plugin and the `kotlin-multiplatform` plugin 2.4.20 require the minimum Gradle version of 7.6.3 for your project to compile.

Similarly, the maximum fully supported version is 9.7.0. It doesn't have deprecated Gradle methods and properties, and supports all the current Gradle features.

Earlier KGP versions

KGP version	Gradle min and max versions	AGP min and max versions
1.9.0–1.9.10	6.8.3–7.6.0	4.2.2–7.4.0
1.8.20–1.8.22	6.8.3–7.6.0	4.1.3–7.4.0
1.8.0–1.8.11	6.8.3–7.3.3	4.1.3–7.2.1
1.7.20–1.7.22	6.7.1–7.1.1	3.6.4–7.0.4
1.7.0–1.7.10	6.7.1–7.0.2	3.4.3–7.0.2
1.6.20–1.6.21	6.1.1–7.0.2	3.4.3–7.0.2

Kotlin Gradle plugin data in a project

By default, the Kotlin Gradle plugin stores persistent project-specific data at the root of the project, in the `.kotlin` directory.

Do not commit the `.kotlin` directory to version control. For example, if you are using Git, add `.kotlin` to your project's `.gitignore` file.

There are properties you can add to the `gradle.properties` file of your project to configure this behavior:

Gradle property	Description
`kotlin.project.persistent.dir`	Configures the location where your project-level data is stored. Default: `<project-root-directory>/.kotlin`
`kotlin.project.persistent.dir.gradle.disableWrite`	Controls whether writing Kotlin data to the `.gradle` directory is disabled (for backward compatibility with older IDEA versions). Default: false

Targeting the JVM

To target the JVM, apply the Kotlin JVM plugin.

Kotlin

```
plugins {
    kotlin("jvm") version "2.4.20"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "2.4.20"
}
```

The `version` should be literal in this block, and it cannot be applied from another build script.

Kotlin and Java sources

Kotlin sources and Java sources can be stored in the same directory, or they can be placed in different directories.

The default convention is to use different directories:

```
project
    - src
        - main (root)
            - kotlin
            - java
```

Do not store Java `.java` files in the `src/*/kotlin` directory, as the `.java` files will not be compiled.

Instead, you can use `src/main/java`.

The corresponding `sourceSets` property should be updated if you are not using the default convention:

Kotlin

```
sourceSets.main {
    java.srcDirs("src/main/myJava", "src/main/myKotlin")
}
```

Groovy

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

Check for JVM target compatibility of related compile tasks

In the build module, you may have related compile tasks, for example:

- `compileKotlin` and `compileJava`
- `compileTestKotlin` and `compileTestJava`

`main` and `test` source set compile tasks are not related.

For related tasks like these, the Kotlin Gradle plugin checks for JVM target compatibility. Different values of the `jvmTarget` attribute in the `kotlin` extension or task and `targetCompatibility` in the `java` extension or task cause JVM target incompatibility. For example: the `compileKotlin` task has `jvmTarget=1.8`, and the `compileJava` task has (or inherits) `targetCompatibility=15`.

Configure the behavior of this check for the whole project by setting the `kotlin.jvm.target.validation.mode` property in the `gradle.properties` file to:

- `error` – the plugin fails the build; the default value for projects on Gradle 8.0+.
- `warning` – the plugin prints a warning message; the default value for projects on Gradle less than 8.0.
- `ignore` – the plugin skips the check and doesn't produce any messages.

You can also configure it at task level in your `build.gradle(.kts)` file:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompile>().configureEach {
    jvmTargetValidationMode.set(org.jetbrains.kotlin.gradle.dsl.jvm.JvmTargetValidationMode.WARNING)
}
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompile.class).configureEach {
    jvmTargetValidationMode = org.jetbrains.kotlin.gradle.dsl.jvm.JvmTargetValidationMode.WARNING
}
```

To avoid JVM target incompatibility, configure a toolchain or align JVM versions manually.

What can go wrong if targets are incompatible

There are two ways of manually setting JVM targets for Kotlin and Java source sets:

- The implicit way via setting up a Java toolchain.
- The explicit way via setting the `jvmTarget` attribute in the `kotlin` extension or task and `targetCompatibility` in the `java` extension or task.

JVM target incompatibility occurs if you:

- Explicitly set different values of `jvmTarget` and `targetCompatibility`.
- Have a default configuration, and your JDK is not equal to `1.8`.

Let's consider a default configuration of JVM targets when you have only the Kotlin JVM plugin in your build script and no additional settings for JVM targets:

Kotlin

```
plugins {
    kotlin("jvm") version "2.4.20"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "2.4.20"
}
```

When there is no explicit information about the `jvmTarget` value in the build script, its default value is `null`, and the compiler translates it to the default value `1.8`. The `targetCompatibility` equals the current Gradle's JDK version, which is equal to your JDK version (unless you use a Java toolchain approach). Assuming that your JDK version is `17`, your published library artifact will declare itself compatible with JDK 17+: `org.gradle.jvm.version=17`, which is wrong. In this case, you have to use Java 17 in your main project to add this library, even though the bytecode's version is `1.8`. Configure a toolchain to solve this issue.

Gradle Java toolchains support

A warning for Android users. To use Gradle toolchain support, use the Android Gradle plugin (AGP) version 8.1.0-alpha09 or higher.

Gradle Java toolchain support is available only from AGP 7.4.0. Nevertheless, because of this issue, AGP did not set `targetCompatibility` to be equal to the toolchain's JDK until the version 8.1.0-alpha09. If you use versions less than 8.1.0-alpha09, you need to configure `targetCompatibility` manually via `compileOptions`. Replace the placeholder `<MAJOR_JDK_VERSION>` with the JDK version you would like to use:

```
android {
    compileOptions {
        sourceCompatibility = <MAJOR_JDK_VERSION>
        targetCompatibility = <MAJOR_JDK_VERSION>
    }
}
```

Gradle 6.7 introduced Java toolchains support. Using this feature, you can:

- Use a JDK and a JRE that are different from the ones in Gradle to run compilations, tests, and executables.
- Compile and test code with a not-yet-released language version.

With toolchains support, Gradle can autodetect local JDKs and install missing JDKs that Gradle requires for the build. Now Gradle itself can run on any JDK and still reuse the remote build cache feature for tasks that depend on a major JDK version.

The Kotlin Gradle plugin supports Java toolchains for Kotlin/JVM compilation tasks. JS and Native tasks don't use toolchains. The Kotlin compiler always runs on the JDK the Gradle daemon is running on. A Java toolchain:

- Sets the `-jdk-home` option available for JVM targets.
- Sets the `compilerOptions.jvmTarget` to the toolchain's JDK version if the user doesn't set the `jvmTarget` option explicitly. If the user doesn't configure the toolchain, the `jvmTarget` field uses the default value. Learn more about JVM target compatibility.
- Sets the toolchain to be used by any Java compile, test, and javadoc tasks.
- Affects which JDK `kapt` workers are running on.

Use the following code to set a toolchain. Replace the placeholder `<MAJOR_JDK_VERSION>` with the JDK version you would like to use:

Kotlin

```
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
    }
    // Or shorter:
    jvmToolchain(<MAJOR_JDK_VERSION>)
    // For example:
    jvmToolchain(17)
}
```

Groovy

```
kotlin {
    jvmToolchain {
        languageVersion = JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)
    }
    // Or shorter:
    jvmToolchain(<MAJOR_JDK_VERSION>)
    // For example:
    jvmToolchain(17)
}
```

Note that setting a toolchain via the `kotlin` extension updates the toolchain for Java compile tasks as well.

You can set a toolchain via the `java` extension, and Kotlin compilation tasks will use it:

Kotlin

```
java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
    }
}
```

Groovy

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)
    }
}
```

If you use Gradle 8.0.2 or higher, you also need to add a toolchain resolver plugin. This type of plugin manages which repositories to download a toolchain from. As an example, add to your `settings.gradle(.kts)` the following plugin:

Kotlin

```
plugins {
    id("org.gradle.toolchains.foojay-resolver-convention") version("1.0.0")
}
```

Groovy

```
plugins {
    id 'org.gradle.toolchains.foojay-resolver-convention' version '1.0.0'
}
```

Check that the version of `foojay-resolver-convention` corresponds to your Gradle version on the Gradle site.

To understand which toolchain Gradle uses, run your Gradle build with the log level `--info` and find a string in the output starting with `[KOTLIN] Kotlin compilation 'jdkHome' argument:`. The part after the colon will be the JDK version from the toolchain.

To set any JDK (even local) for a specific task, use the Task DSL.

Learn more about Gradle JVM toolchain support in the Kotlin plugin.

Set JDK version with the Task DSL

The Task DSL allows setting any JDK version for any task implementing the `UsesKotlinJavaToolchain` interface. At the moment, these tasks are `KotlinCompile` and `KaptTask`. If you want Gradle to search for the major JDK version, replace the `<MAJOR_JDK_VERSION>` placeholder in your build script:

Kotlin

```
val service = project.extensions.getByType<JavaToolchainService>()
val customLauncher = service.launcherFor {
    languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
}
project.tasks.withType<UsesKotlinJavaToolchain>().configureEach {
    kotlinJavaToolchain.toolchain.use(customLauncher)
}
```

Groovy

```
JavaToolchainService service = project.getExtensions().getByType(JavaToolchainService.class)
Provider<JavaLauncher> customLauncher = service.launcherFor {
    it.languageVersion = JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)
}
tasks.withType(UsesKotlinJavaToolchain::class).configureEach { task ->
    task.kotlinJavaToolchain.toolchain.use(customLauncher)
}
```

Or you can specify the path to your local JDK and replace the placeholder `<LOCAL_JDK_VERSION>` with this JDK version:

```
tasks.withType<UsesKotlinJavaToolchain>().configureEach {
    kotlinJavaToolchain.jdk.use(
        "/path/to/local/jdk", // Put a path to your JDK
        JavaVersion.<LOCAL_JDK_VERSION> // For example, JavaVersion.17
    )
}
```

Associate compiler tasks

You can *associate* compilations by setting up such a relationship between them that one compilation uses the compiled outputs of the other. Associating compilations establishes `internal` visibility between them.

The Kotlin compiler associates some compilations by default, such as the `test` and `main` compilations of each target. If you need to express that one of your custom compilations is connected to another, create your own associated compilation.

To make the IDE support associated compilations for inferring visibility between source sets, add the following code to your `build.gradle(.kts)`:

Kotlin

```
val integrationTestCompilation = kotlin.target.compilations.create("integrationTest") {
    associateWith(kotlin.target.compilations.getByName("main"))
}
```

Groovy

```
integrationTestCompilation {
    kotlin.target.compilations.create("integrationTest") {
        associateWith(kotlin.target.compilations.getByName("main"))
    }
}
```

Here, the `integrationTest` compilation is associated with the `main` compilation that gives access to `internal` objects from functional tests.

Configure with Java Modules (JPMS) enabled

To make the Kotlin Gradle plugin work with Java Modules, add the following lines to your build script and replace `YOUR_MODULE_NAME` with a reference to your JPMS module, for example, `org.company.module`:

Kotlin

```
tasks.named("compileJava", JavaCompile::class.java) {
    // Provide compiled Kotlin classes to javac - needed for Java/Kotlin mixed sources to work
    val mainOutput: FileCollection = sourceSets["main"].output
    options.compilerArgumentProviders.add(CommandLineArgumentProvider {
        listOf("--patch-module", "YOUR_MODULE_NAME=${mainOutput.asPath}")
    })
}
```

Groovy

```
tasks.named("compileJava", JavaCompile.class) {
    // Provide compiled Kotlin classes to javac - needed for Java/Kotlin mixed sources to work
    FileCollection mainOutput = sourceSets["main"].output
    options.compilerArgumentProviders.add(new CommandLineArgumentProvider() {
        @Override
        Iterable<String> asArguments() {
            return ["--patch-module", "YOUR_MODULE_NAME=${mainOutput.asPath}"]
        }
    })
}
```

Put `module-info.java` into the `src/main/java` directory as usual.

For a module, a package name in Kotlin files should be equal to the package name from `module-info.java` to avoid a "package is empty or does not exist" build failure.

Learn more about:

- Building modules for the Java Module System
- Building applications using the Java Module System
- What "module" means in Kotlin

Other details

Disable use of artifact in compilation task

In some rare scenarios, you can experience a build failure caused by a circular dependency error. For example, when you have multiple compilations where one compilation can see all internal declarations of another, and the generated artifact relies on the output of both compilation tasks:

```
FAILURE: Build failed with an exception.

What went wrong:
Circular dependency between the following tasks:
:lib:compileKotlinJvm
--- :lib:jvmJar
     \--- :lib:compileKotlinJvm (*)
(*) - details omitted (listed previously)
```

To fix this circular dependency error, we've added a Gradle property: `archivesTaskOutputAsFriendModule`. This property controls the use of artifact inputs in the compilation task and determines if a task dependency is created as a result.

By default, this property is set to `true` to track the task dependency. If you encounter a circular dependency error, you can disable the use of the artifact in the compilation task to remove the task dependency and avoid the circular dependency error.

To disable the use of the artifact in the compilation task, add the following to your `gradle.properties` file:

```
kotlin.build.archivesTaskOutputAsFriendModule=false
```

Lazy Kotlin/JVM task creation

Starting from Kotlin 1.8.20, the Kotlin Gradle plugin registers all tasks and doesn't configure them on a dry run.

Non-default location of compile tasks' destinationDirectory

If you override the Kotlin/JVM `KotlinJvmCompile` / `KotlinCompile` task's `destinationDirectory` location, update your build script. You need to explicitly add `sourceSets.main.kotlin.classesDirectories` to `sourceSets.main.outputs` in your JAR file:

```
tasks.jar(type: Jar) {
    from sourceSets.main.outputs
    from sourceSets.main.kotlin.classesDirectories
}
```

Targeting multiple platforms

Projects targeting multiple platforms, called multiplatform projects, require the `kotlin-multiplatform` plugin.



The `kotlin-multiplatform` plugin works with Gradle 7.6.3 or later.

Kotlin

```
plugins {
    kotlin("multiplatform") version "2.4.20"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.4.20'
}
```

Learn more about Kotlin Multiplatform for different platforms and Kotlin Multiplatform for iOS and Android.

Targeting Android

It's recommended to use Android Studio for creating Android applications. Learn how to use the Android Gradle plugin.

Targeting the web

Kotlin, through Kotlin Multiplatform, offers two approaches for web development:

- JavaScript-based (using the Kotlin/JS compiler)
- WebAssembly-based (using the Kotlin/Wasm compiler)

Both approaches use the Kotlin Multiplatform plugin but support different use cases. The sections below explain how to configure each target in your Gradle build and when to use them.

Targeting JavaScript

Use Kotlin/JS if your goal is to:

- Share business logic with a JavaScript/TypeScript codebase
- Build non-shareable web apps with Kotlin

For more information, see Web development.

When targeting JavaScript, use the `kotlin-multiplatform` plugin:

Kotlin

```
plugins {
    kotlin("multiplatform") version "2.4.20"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.4.20'
}
```

Configure the JavaScript target by specifying whether it should run in the browser or the Node.js environment:

```
kotlin {
    js().browser {  // or js().nodejs
        /* ... */
    }
}
```

See further details about the Gradle configuration for JavaScript and learn more about setting up a Kotlin/JS project.

Targeting WebAssembly

Use Kotlin/Wasm if you want to share both logic and UI across multiple platforms. For more information, see Web development.

As with JavaScript, use the `kotlin-multiplatform` plugin when targeting WebAssembly (Wasm):

Kotlin

```
plugins {
    kotlin("multiplatform") version "2.4.20"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.4.20'
}
```

Depending on your requirements, you can target:

- `wasmJs`: for running in browsers or Node.js
- `wasmWasi`: for running in Wasm environments supporting WASI (WebAssembly System Interface), such as Wasmtime, WasmEdge, and so on.

Configure the `wasmJs` target for web browsers or Node.js:

```
kotlin {
    wasmJs {
        browser { // or nodejs
            /* ... */
        }
    }
}
```

For WASI environments, configure the `wasmWasi` target with Node.js or Wasmtime:

```
kotlin {
    wasmWasi {
        nodejs { // or wasmtime
            /* ... */
        }
    }
}
```

See further details about the Gradle configuration for Wasm.

Kotlin and Java sources for the web target

The KGP only works for Kotlin files, so it is recommended that you keep Kotlin and Java files separate (if the project contains Java files). If you don't store them separately, specify the source folder in the `sourceSets{}` block:

Kotlin

```
kotlin {
    sourceSets["main"].apply {
        kotlin.srcDir("src/main/myKotlin")
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        main.kotlin.srcDirs += 'src/main/myKotlin'
    }
}
```

Triggering configuration actions with the KotlinBasePlugin interface

To trigger some configuration action whenever any Kotlin Gradle plugin (JVM, JS, Multiplatform, Native, and others) is applied, use the `KotlinBasePlugin` interface that all Kotlin plugins inherit from:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.plugin.KotlinBasePlugin

// ...

project.plugins.withType<KotlinBasePlugin>() {
    // Configure your action here
}
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.plugin.KotlinBasePlugin

// ...

project.plugins.withType(KotlinBasePlugin.class) {
    // Configure your action here
}
```

Configure dependencies

To add a dependency on a library, set the dependency of the required type (for example, `implementation`) in the `dependencies{}` block of the source sets DSL.

Kotlin

```kotlin
kotlin {
    sourceSets {
        commonMain.dependencies {
            implementation("com.example:my-library:1.0")
        }
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'com.example:my-library:1.0'
            }
        }
    }
}
```

Experimental

Configure dependencies at the top level

You can configure common dependencies in multiplatform projects using a top-level `dependencies {}` block. Dependencies declared here behave as if they were added to the `commonMain` or `commonTest` source sets.

To use the top-level `dependencies {}` block, opt in by adding the `@OptIn(ExperimentalKotlinGradlePluginApi::class)` annotation before the block:

Kotlin

```
kotlin {
    @OptIn(ExperimentalKotlinGradlePluginApi::class)
    dependencies {
        implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0")
    }
}
```

Groovy

```
kotlin {
    dependencies {
        implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0'
    }
}
```

Add platform-specific dependencies inside the `sourceSets {}` block of the corresponding target.

You can share your feedback on this feature in YouTrack.

Dependency types

Choose the dependency type based on your requirements.

Type	Description	When to use
`api`	Used both during compilation and at runtime and is exported to library consumers.	If any type from a dependency is used in the public API of the current module, use an `api` dependency.
`implementation`	Used during compilation and at runtime for the current module, but is not exposed for compilation of other modules depending on the one with the `implementation` dependency.	Use for dependencies needed for the internal logic of a module. If a module is an endpoint application which is not published, use `implementation` dependencies instead of `api` dependencies.
`compileOnly`	Used for compilation of the current module and is not available at runtime nor during compilation of other modules.	Use for APIs which have a third-party implementation available at runtime.

Type	Description	When to use
`runtimeOnly`	Available at runtime but is not visible during compilation of any module.	

Dependency on the standard library

A dependency on the standard library (`stdlib`) is added automatically to each source set. The version of the standard library used is the same as the version of the Kotlin Gradle plugin.

For platform-specific source sets, the corresponding platform-specific variant of the library is used, while a common standard library is added to the rest. The Kotlin Gradle plugin selects the appropriate JVM standard library depending on the `compilerOptions.jvmTarget` compiler option of your Gradle build script.

If you declare a standard library dependency explicitly (for example, if you need a different version), the Kotlin Gradle plugin won't override it or add a second standard library.

If you don't need a standard library at all, you can add the following Gradle property to your `gradle.properties` file:

```
kotlin.stdlib.default.dependency=false
```

Versions alignment of transitive dependencies

From Kotlin standard library version 1.9.20, Gradle uses metadata included in the standard library to automatically align transitive `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` dependencies.

If you add a dependency for any Kotlin standard library version between 1.8.0 – 1.9.10, for example: `implementation("org.jetbrains.kotlin:kotlin-stdlib:1.8.0")` , then the Kotlin Gradle Plugin uses this Kotlin version for transitive `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` dependencies. This avoids class duplication from different standard library versions. Learn more about merging `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` into `kotlin-stdlib` . You can disable this behavior with the `kotlin.stdlib.jdk.variants.version.alignment` Gradle property in your `gradle.properties` file:

```
kotlin.stdlib.jdk.variants.version.alignment=false
```

Other ways to align versions

- If you have issues with version alignment, you can align all versions via the Kotlin BOM. Declare a platform dependency on `kotlin-bom` in your build script:

  Kotlin

  ```
  implementation(platform("org.jetbrains.kotlin:kotlin-bom:2.4.20"))
  ```

  Groovy

  ```
  implementation platform('org.jetbrains.kotlin:kotlin-bom:2.4.20')
  ```

-

If you don't add a dependency for a standard library version, but you have two different dependencies that transitively bring different old versions of the Kotlin standard library, then you can explicitly require `2.4.20` versions of these transitive libraries:

Kotlin

```
dependencies {
    constraints {
        add("implementation", "org.jetbrains.kotlin:kotlin-stdlib-jdk7") {
            version {
                require("2.4.20")
            }
        }
        add("implementation", "org.jetbrains.kotlin:kotlin-stdlib-jdk8") {
            version {
                require("2.4.20")
            }
        }
    }
}
```

Groovy

```
dependencies {
    constraints {
        add("implementation", "org.jetbrains.kotlin:kotlin-stdlib-jdk7") {
            version {
                require("2.4.20")
            }
        }
        add("implementation", "org.jetbrains.kotlin:kotlin-stdlib-jdk8") {
            version {
                require("2.4.20")
            }
        }
    }
}
```

- If you add a dependency for Kotlin standard library version `2.4.20`: `implementation("org.jetbrains.kotlin:kotlin-stdlib:2.4.20")`, and an old version (earlier than `1.8.0`) of the Kotlin Gradle plugin, update the Kotlin Gradle plugin to match the standard library version:

Kotlin

```
plugins {
    // replace `<...>` with the plugin name
    kotlin("<...>") version "2.4.20"
}
```

Groovy

```
plugins {
    // replace `<...>` with the plugin name
    id "org.jetbrains.kotlin.<...>" version "2.4.20"
}
```

- If you use versions prior to `1.8.0` of `kotlin-stdlib-jdk7` / `kotlin-stdlib-jdk8`, for example, `implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk7:SOME_OLD_KOTLIN_VERSION")`, and a dependency that transitively brings `kotlin-stdlib:1.8+`, replace your `kotlin-stdlib-jdk<7/8>:SOME_OLD_KOTLIN_VERSION` with `kotlin-stdlib-jdk*:2.4.20` or exclude the transitive `kotlin-stdlib:1.8+` from the library that brings it:

Kotlin

```
dependencies {
    implementation("com.example:lib:1.0") {
        exclude(group = "org.jetbrains.kotlin", module = "kotlin-stdlib")
    }
}
```

Groovy

```
dependencies {
    implementation("com.example:lib:1.0") {
        exclude group: "org.jetbrains.kotlin", module: "kotlin-stdlib"
    }
}
```

Set dependencies on test libraries

The `kotlin.test` API is available for testing Kotlin projects on all supported platforms. Add the `kotlin-test` dependency to the `commonTest` source set, so that the Gradle plugin can infer the corresponding test dependencies for each test source set.

Kotlin/Native targets do not require additional test dependencies, and the `kotlin.test` API implementations are built-in.

Kotlin

```
kotlin {
    sourceSets {
        commonTest.dependencies {
            implementation(kotlin("test")) // This brings all the platform dependencies automatically
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // This brings all the platform dependencies
automatically
            }
        }
    }
}
```

You can use shorthand for a dependency on a Kotlin module, for example, kotlin("test") for "org.jetbrains.kotlin:kotlin-test".

You can use the `kotlin-test` dependency in any shared or platform-specific source set as well.

JVM variants of kotlin-test

For Kotlin/JVM, Gradle uses JUnit 4 by default. Therefore, the `kotlin("test")` dependency resolves to the variant for JUnit 4, namely `kotlin-test-junit`.

You can choose JUnit 5 or TestNG by calling `useJUnitPlatform()` or `useTestNG()` in the test task of your build script. The following example is for a Kotlin Multiplatform project:

Kotlin

```
kotlin {
    jvm {
        testRuns["test"].executionTask.configure {
            useJUnitPlatform()
        }
    }
    sourceSets {
        commonTest.dependencies {
            implementation(kotlin("test"))
        }
    }
}
```

Groovy

```
kotlin {
    jvm {
        testRuns["test"].executionTask.configure {
            useJUnitPlatform()
        }
    }
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test")
            }
        }
    }
}
```

The following example is for a JVM project:

Kotlin

```
dependencies {
    testImplementation(kotlin("test"))
}

tasks {
    test {
        useTestNG()
    }
}
```

Groovy

```
dependencies {
    testImplementation 'org.jetbrains.kotlin:kotlin-test'
}

test {
    useTestNG()
}
```

Learn how to test code using JUnit on the JVM.

Automatic JVM variant resolution can sometimes cause problems for your configuration. In that case, you can specify the necessary framework explicitly and disable the automatic resolution by adding this line to the project `gradle.properties` file:

```
kotlin.test.infer.jvm.variant=false
```

If you have used a variant of `kotlin("test")` in your build script explicitly and your project build stopped working with a compatibility conflict, see this issue in the Compatibility guide.

Set a dependency on a kotlinx library

If you use a multiplatform library and need to depend on the shared code, set the dependency only once in the shared source set. Use the library's base artifact name, such as `kotlinx-coroutines-core` or `ktor-client-core`:

Kotlin

```
kotlin {
    sourceSets {
        commonMain.dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0")
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0'
            }
        }
    }
}
```

If you need a kotlinx library for a platform-specific dependency, you can still use the library's base artifact name in the corresponding platform source set:

Kotlin

```
kotlin {
    sourceSets {
        jvmMain.dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0")
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        jvmMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0'
            }
        }
    }
}
```

Declare repositories

You can declare a publicly-available repository to use its open source dependencies. In the `repositories{}` block, set the name of the repository:

Kotlin

```
repositories {
    mavenCentral()
}
```

Groovy

```
repositories {
    mavenCentral()
}
```

Popular repositories are Maven Central and Google's Maven repository.

If you also work with Maven projects, we recommend avoiding adding `mavenLocal()` as a repository because you may experience problems when switching between Gradle and Maven projects. If you must add the `mavenLocal()` repository, add it as the last repository in your `repositories{}` block. For more information, see The case for mavenLocal().

If you need to declare the same repositories in more than one subproject, declare the repositories centrally in the `dependencyResolutionManagement{}` block in your `settings.gradle(.kts)` file:

Kotlin

```
dependencyResolutionManagement {
    repositories {
        mavenCentral()
    }
}
```

Groovy

```
dependencyResolutionManagement {
    repositories {
        mavenCentral()
    }
}
```

Any declared repositories in subprojects override repositories declared centrally. For more information on how to control this behavior and what options are available, see Gradle's documentation.

Experimental

Register generated sources

Register generated sources to help IDEs, third-party plugins, and other tools distinguish between generated code and regular source files. This helps tools like IDEs highlight generated code differently in the UI and trigger generation tasks when importing the project. Use the `KotlinSourceSet` interface to register generated sources.

To register a directory that contains Kotlin files, use the `generatedKotlin` property with the `SourceDirectorySet` type in your `build.gradle.kts` file. For example:

```
val generatorTask = project.tasks.register("generator") {
    val outputDirectory = project.layout.projectDirectory.dir("src/main/kotlinGen")
    outputs.dir(outputDirectory)
    doLast {
        outputDirectory.file("generated.kt").asFile.writeText(
            // language=kotlin
            """
            fun printHello() {
                println("hello")
            }
            """.trimIndent()
        )
    }
}

kotlin.sourceSets.getByName("main").generatedKotlin.srcDir(generatorTask)
```

This example creates a new task `generator` with an output directory of `"src/main/kotlinGen"`. When the task runs, the `doLast {}` task action creates a `generated.kt` file in the output directory. Finally, the example registers the task's output as a generated source.

If you're developing a Gradle plugin, you can use the `allKotlinSources` property to access all sources registered in the `KotlinSourceSet.kotlin` and `KotlinSourceSet.generatedKotlin` properties.

What's next?

Learn more about:

- Compiler options and how to pass them.
- Incremental compilation, caches support, build reports, and the Kotlin daemon.
- Gradle basics and specifics.
- Support for Gradle plugin variants.

Gradle best practices

Gradle is a build system used by many Kotlin projects to automate and manage the build process.

Getting the best out of Gradle is essential to help you spend less time managing and waiting for builds, and more time coding. Here we provide a set of best practices split into two key areas: **organizing** and **optimizing** your projects.

Organize

This section focuses on structuring your Gradle projects to improve clarity, maintainability, and scalability.

Use Kotlin DSL

Use Kotlin DSL instead of the traditional Groovy DSL. You avoid learning another language and gain the benefits of strict typing. Strict typing lets IDEs provide better support for refactoring and auto-completion, making development more efficient.

Find more information in Gradle's Kotlin DSL primer.

Read Gradle's blog about Kotlin DSL becoming the default for Gradle builds.

Use a version catalog

Use a version catalog in a `libs.versions.toml` file to centralize dependency management. This enables you to define and reuse versions, libraries, and plugins consistently across projects.

```
[versions]
kotlinxCoroutines = "1.11.0"

[libraries]
kotlinxCoroutines = { module = "org.jetbrains.kotlinx:kotlinx-coroutines-core", version.ref =
"kotlinxCoroutines" }
```

With the following dependency added to your `build.gradle.kts` file:

```
dependencies {
    implementation(libs.kotlinxCoroutines)
}
```

Learn more in Gradle's documentation about Dependency management basics.

Use convention plugins

Use convention plugins to encapsulate and reuse common build logic across multiple build files. Moving a shared configuration into a plugin helps simplify and modularize your build scripts.

Although the initial setup may be time-consuming, it's easy to maintain and add new build logic once you complete it.

Learn more in Gradle's documentation about Convention plugins.

Optimize

This section provides strategies to enhance the performance and efficiency of your Gradle builds.

Use local build cache

Use a local build cache to save time by reusing outputs produced by other builds. The build cache can retrieve outputs from any earlier build that you have already created.

Learn more in Gradle's documentation about their Build cache.

Use configuration cache

> The configuration cache doesn't support all core Gradle plugins yet. For the latest information, see Gradle's table of supported plugins.

Use the configuration cache to significantly improve build performance by caching the result of the configuration phase and reusing it for subsequent builds. If Gradle detects no changes in the build configuration or related dependencies, it skips the configuration phase.

The configuration cache also enables parallel execution of independent tasks within a single project, which can further improve build performance. Additionally, it implicitly enables the `org.gradle.parallel` property, allowing tasks across different projects to execute in parallel.

Learn more about the configuration cache in Gradle's documentation.

Improve build times for multiple targets

When your multiplatform project includes multiple targets, tasks like `build` and `assemble` can compile the same code multiple times for each target, leading to longer compilation times.

If you're actively developing and testing a specific platform, run the corresponding `linkDebug*` task instead.

For more information, see Tips for improving compilation time.

Migrate from kapt to KSP

If you're using a library that relies on the kapt compiler plugin, check whether you can switch to using the Kotlin Symbol Processing (KSP) API instead. The KSP API improves build performance by reducing annotation processing time. KSP is faster and more efficient than kapt, as it processes source code directly without generating intermediary Java stubs.

For guidance on the migration steps, see the migration guide.

Use modularization

> Modularization only benefits projects of moderate to large size. It doesn't provide advantages for projects based on a microservices architecture.

Use a modularized project structure to increase build speed and enable easier parallel development. Structure your project into one root project and one or more subprojects. If changes only affect one of the subprojects, Gradle rebuilds only that specific subproject.

```
.
└── root-project/
    ├── settings.gradle.kts
    ├── app subproject/
    │   └── build.gradle.kts
    └── lib subproject/
        └── build.gradle.kts
```

Learn more in Gradle's documentation about Structuring projects with Gradle.

Set up CI/CD

Set up a CI/CD process to significantly reduce build time by using incremental builds and caching dependencies. Add persistent storage or use a remote build cache to see these benefits. This process doesn't have to be time-consuming, as some providers, like GitHub, offer this service almost out of the box.

Explore Gradle's community cookbook on Using Gradle with Continuous Integration systems.

Use remote build cache

Like the local build cache, the remote build cache helps you save time by reusing outputs from other builds. It can retrieve task outputs from any earlier build that anyone has already run, not just the last one.

The remote build cache uses a cache server to share task outputs across builds. For example, in a development environment with a CI/CD server, all builds on the server populate the remote cache. When you check out the main branch to start a new feature, you can immediately access incremental builds.

Keep in mind that a slow internet connection might make transferring cached results slower than running tasks locally.

Learn more in Gradle's documentation about their Build cache.

Compiler options in the Kotlin Gradle plugin

Each release of Kotlin includes compilers for the supported targets: JVM, JavaScript, and native binaries for supported platforms.

These compilers are used by:

- The IDE, when you click the **Compile** or **Run** button for your Kotlin project.
- Gradle, when you call `gradle build` in a console or in the IDE.
- Maven, when you call `mvn compile` or `mvn test-compile` in a console or in the IDE.

You can also run Kotlin compilers manually from the command line as described in the Working with command-line compiler tutorial.

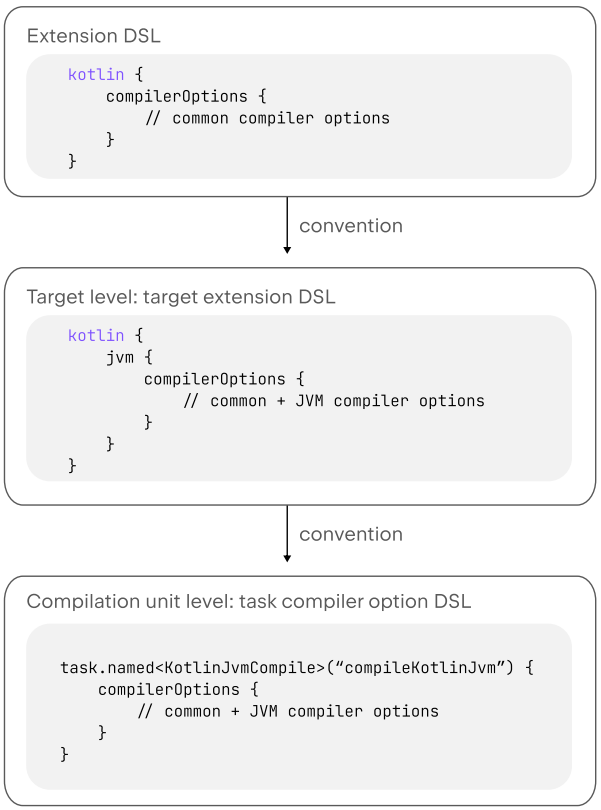
How to define options

Kotlin compilers have a number of options for tailoring the compiling process.

The Gradle DSL allows comprehensive configuration of compiler options. It is available for Kotlin Multiplatform and JVM/Android projects.

With the Gradle DSL, you can configure compiler options within the build script at three levels:

- Extension level, in the `kotlin {}` block for all targets and shared source sets.
- Target level, in the block for a specific target.
- Compilation unit level, usually in a specific compilation task.



Kotlin compiler options levels

The settings at a higher level are used as a convention (default) for a lower level:

- Compiler options set at the extension level are the default for target-level options, including shared source sets like `commonMain`, `nativeMain`, and `commonTest`.
- Compiler options set at the target level are the default for options at the compilation unit (task) level, like `compileKotlinJvm` and `compileTestKotlinJvm` tasks.

In turn, configurations made at a lower level override related settings at a higher level:

- Task-level compiler options override related configurations at the target or the extension level.
- Target-level compiler options override related configurations at the extension level.

To find out which level of compiler arguments is applied to the compilation, use the `DEBUG` level of Gradle logging. For JVM and JS/WASM tasks, search for the `"Kotlin compiler args:"` string within the logs; for Native tasks, search for the `"Arguments ="` string.

> If you're a third-party plugin author, it's best to apply your configuration on the project level to avoid overriding issues. You can use the new Kotlin plugin DSL extension types for this. It's recommended that you document this configuration on your side explicitly.

Extension level

You can configure common compiler options for all the targets and shared source sets in the `compilerOptions {}` block at the top level:

```
kotlin {
    compilerOptions {
        optIn.add("kotlin.RequiresOptIn")
    }
}
```

Target level

You can configure compiler options for the JVM/Android target in the `compilerOptions {}` block inside the `target {}` block:

```
kotlin {
    target {
        compilerOptions {
            optIn.add("kotlin.RequiresOptIn")
        }
    }
}
```

In Kotlin Multiplatform projects, you can configure compiler options inside the specific target. For example, `jvm { compilerOptions {}}`. For more information, see Multiplatform Gradle DSL reference.

Compilation unit level

You can configure compiler options for a specific compilation unit or task in a `compilerOptions {}` block inside the task configuration:

```
tasks.named<KotlinJvmCompile>("compileKotlin"){
    compilerOptions {
        optIn.add("kotlin.RequiresOptIn")
    }
}
```

You can also access and configure compiler options at a compilation unit level via `KotlinCompilation`:

```
kotlin {
    target {
        val main by compilations.getting {
            compileTaskProvider.configure {
                compilerOptions {
                    optIn.add("kotlin.RequiresOptIn")
                }
            }
        }
    }
}
```

If you want to configure a plugin of a target different from JVM/Android and Kotlin Multiplatform, use the `compilerOptions {}` property of the corresponding Kotlin compilation task. The following examples show how to set this configuration up in both Kotlin and Groovy DSLs:

Kotlin

```
tasks.named("compileKotlin", org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask::class.java) {
    compilerOptions {
        apiVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_0)
    }
}
```

Groovy

```
tasks.named('compileKotlin', org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class) {
    compilerOptions {
        apiVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_0)
    }
}
```

Migrate from `kotlinOptions {}` to `compilerOptions {}`

Before Kotlin 2.2.0, you could configure compiler options using the `kotlinOptions {}` block. Since the `kotlinOptions {}` block is deprecated from Kotlin 2.0.0, this section provides guidance and recommendations for migrating your build scripts to use the `compilerOptions {}` block instead:

- Centralize compiler options and use types
- Migrate away from `android.kotlinOptions`
- Migrate `freeCompilerArgs`

Centralize compiler options and use types

Whenever possible, configure compiler options at the extension level, and override them for specific tasks at the compilation unit level.

You can't use raw strings in the `compilerOptions {}` block, so convert them to typed values. For example, if you have:

Kotlin

```
plugins {
    kotlin("jvm") version "2.4.20"
}

tasks.withType<KotlinCompile>().configureEach {
    kotlinOptions {
        jvmTarget = "17"
        languageVersion = "2.4"
        apiVersion = "2.4"
    }
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '2.4.20'
}

tasks.withType(KotlinCompile).configureEach {
    kotlinOptions {
        jvmTarget = '17'
        languageVersion = '2.4'
        apiVersion = '2.4'
    }
}
```

After migration, it should be:

Kotlin

```
import org.jetbrains.kotlin.gradle.dsl.JvmTarget
import org.jetbrains.kotlin.gradle.dsl.KotlinVersion

plugins {
    kotlin("jvm") version "2.4.20"
}

kotlin {
    // Extension level
    compilerOptions {
        jvmTarget = JvmTarget.fromTarget("17")
        languageVersion = KotlinVersion.fromVersion("2.4")
        apiVersion = KotlinVersion.fromVersion("2.4")
    }
}

// Example of overriding at compilation unit level
tasks.named<KotlinJvmCompile>("compileKotlin"){
    compilerOptions {
        apiVersion = KotlinVersion.fromVersion("2.4")
    }
}
```

Groovy

```
import org.jetbrains.kotlin.gradle.dsl.JvmTarget
import org.jetbrains.kotlin.gradle.dsl.KotlinVersion

plugins {
    id 'org.jetbrains.kotlin.jvm' version '2.4.20'
}

kotlin {
  // Extension level
    compilerOptions {
        jvmTarget = JvmTarget.fromTarget("17")
        languageVersion = KotlinVersion.fromVersion("2.4")
        apiVersion = KotlinVersion.fromVersion("2.4")
    }
}

// Example of overriding at compilation unit level
tasks.named("compileKotlin", KotlinJvmCompile).configure {
    compilerOptions {
        apiVersion = KotlinVersion.fromVersion("2.4")
    }
}
```

Migrate away from `android.kotlinOptions`

If your build script previously used `android.kotlinOptions`, migrate to `kotlin.compilerOptions` instead. Either at the extension level or the target level.

For example, if you have an Android project:

Kotlin

```
plugins {
    id("com.android.application")
    kotlin("android")
}

android {
    kotlinOptions {
        jvmTarget = "17"
    }
}
```

Groovy

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
}

android {
    kotlinOptions {
        jvmTarget = '17'
    }
}
```

Update it to:

Kotlin

```
plugins {
    id("com.android.application")
    kotlin("android")
}

kotlin {
    compilerOptions {
        jvmTarget = JvmTarget.fromTarget("17")
    }
}
```

Groovy

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
}

kotlin {
    compilerOptions {
        jvmTarget = JvmTarget.fromTarget("17")
    }
}
```

And for example, if you have a Kotlin Multiplatform project with an Android target:

Kotlin

```
plugins {
    kotlin("multiplatform")
    id("com.android.application")
}

kotlin {
    androidTarget {
        compilations.all {
            kotlinOptions.jvmTarget = "17"
        }
    }
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform'
    id 'com.android.application'
}

kotlin {
    androidTarget {
        compilations.all {
            kotlinOptions {
                jvmTarget = '17'
            }
        }
    }
}
```

Update it to:

Kotlin

```
plugins {
    kotlin("multiplatform")
    id("com.android.application")
}

kotlin {
    androidTarget {
        compilerOptions {
            jvmTarget = JvmTarget.fromTarget("17")
        }
    }
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform'
    id 'com.android.application'
}

kotlin {
    androidTarget {
        compilerOptions {
            jvmTarget = JvmTarget.fromTarget("17")
        }
    }
}
```

Migrate `freeCompilerArgs`

- Replace all `+=` operations with `add()` or `addAll()` functions.
- If you use the `-opt-in` compiler option, check whether a specialized DSL already is available in the KGP API reference and use that instead.
- Migrate any use of the `-progressive` compiler option to use the dedicated DSL: `progressiveMode.set(true)`.
- Migrate any use of the `-Xjvm-default` compiler option to use the dedicated DSL: `jvmDefault.set()`. Use the following mapping for the options:

Before	After
`-Xjvm-default=all-compatibility`	`jvmDefault.set(JvmDefaultMode.ENABLE)`
`-Xjvm-default=all`	`jvmDefault.set(JvmDefaultMode.NO_COMPATIBILITY)`
`-Xjvm-default=disable`	`jvmDefault.set(JvmDefaultMode.DISABLE)`

For example, if you have:

Kotlin

```
kotlinOptions {
    freeCompilerArgs += "-opt-in=kotlin.RequiresOptIn"
    freeCompilerArgs += listOf("-Xcontext-receivers", "-Xinline-classes", "-progressive", "-Xjvm-default=all")
}
```

Groovy

```
kotlinOptions {
    freeCompilerArgs += "-opt-in=kotlin.RequiresOptIn"
    freeCompilerArgs += ["-Xcontext-receivers", "-Xinline-classes", "-progressive", "-Xjvm-
default=all"]
}
```

Migrate to:

Kotlin

```
kotlin {
    compilerOptions {
        optIn.add("kotlin.RequiresOptIn")
        freeCompilerArgs.addAll(listOf("-Xcontext-receivers", "-Xinline-classes"))
        progressiveMode.set(true)
        jvmDefault.set(JvmDefaultMode.NO_COMPATIBILITY)
    }
}
```

Groovy

```
kotlin {
    compilerOptions {
        optIn.add("kotlin.RequiresOptIn")
        freeCompilerArgs.addAll(["-Xcontext-receivers", "-Xinline-classes"])
        progressiveMode.set(true)
        jvmDefault.set(JvmDefaultMode.NO_COMPATIBILITY)
    }
}
```

Target the JVM

As explained before, you can define compiler options for your JVM/Android projects at the extension, target, and compilation unit levels (tasks).

Default JVM compilation tasks are called `compileKotlin` for production code and `compileTestKotlin` for test code. The tasks for custom source sets are named according to their `compile<Name>Kotlin` patterns.

You can see the list of Android compilation tasks by running the `gradlew tasks --all` command in the terminal and searching for `compile*Kotlin` task names in the `Other tasks` group.

Some important details to be aware of:

- `kotlin.compilerOptions` configures every Kotlin compilation task in the project.
- You can override the configuration applied by `kotlin.compilerOptions` DSL using the `tasks.named<KotlinJvmCompile>("compileKotlin") { }` (or `tasks.withType<KotlinJvmCompile>().configureEach { }`) approach.

Target JavaScript

JavaScript compilation tasks are called `compileKotlinJs` for production code, `compileTestKotlinJs` for test code, and `compile<Name>KotlinJs` for custom source sets.

To configure a single task, use its name:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

val compileKotlin: KotlinCompilationTask<*> by tasks

compileKotlin.compilerOptions.suppressWarnings.set(true)
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        suppressWarnings = true
    }
}
```

Note that with the Gradle Kotlin DSL, you should get the task from the project's `tasks` first.

Use the `Kotlin2JsCompile` and `KotlinCompileCommon` types for JS and common targets, respectively.

You can see the list of JavaScript compilation tasks by running the `gradlew tasks --all` command in the terminal and searching for `compile*KotlinJS` task names in the `Other tasks` group.

All Kotlin compilation tasks

It is also possible to configure all the Kotlin compilation tasks in the project:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named<KotlinCompilationTask<*>>("compileKotlin").configure {
    compilerOptions { /*...*/ }
}
```

Groovy

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions { /*...*/ }
}
```

All compiler options

Here is a complete list of options for the Gradle compiler:

Common attributes

Name	Description	Possible values	Default value
`optIn`	A property for configuring a list of opt-in compiler arguments	`listOf( /* opt-ins */ )`	`emptyList()`
`progressiveMode`	Enables the progressive compiler mode	`true` , `false`	`false`
`extraWarnings`	Enables additional declaration, expression, and type compiler checks that emit warnings if true	`true` , `false`	`false`

Attributes specific to JVM

Name	Description	Possible values	Default value
`javaParameters`	Generate metadata for Java 1.8 reflection on method parameters		false
`jvmTarget`	Target version of the generated JVM bytecode	"1.8", "9", "10", ..., "25", 26". Also, see Types for compiler options	"1.8"
`noJdk`	Don't automatically include the Java runtime into the classpath		false

Name	Description	Possible values	Default value
`jvmTargetValidationMode`	• Validation of the JVM target compatibility between Kotlin and Java • A property for tasks of the `KotlinCompile` type.	`WARNING`, `ERROR`, `IGNORE`	`ERROR`
`jvmDefault`	Control how functions declared in interfaces are compiled to default methods on the JVM	`ENABLE`, `NO_COMPATIBILITY`, `DISABLE`	`ENABLE`

Attributes common to JVM and JavaScript

Name	Description	Possible values	Default value
`allWarningsAsErrors`	Report an error if there are any warnings		false
`suppressWarnings`	Don't generate warnings		false
`verbose`	Enable verbose logging output. Works only when the Gradle debug log level enabled		false
`freeCompilerArgs`	A list of additional compiler arguments. You can use experimental `-X` arguments here too. See an example		[]
`apiVersion`	Control which Kotlin APIs your code can use. For more information, see `-api-version`.	"2.0", "2.1", "2.2", "2.3", "2.4", "2.5" (EXPERIMENTAL)	
`languageVersion`	Control which Kotlin language features and syntax are available during compilation. For more information, see `-language-version`.	"2.0", "2.1", "2.2", "2.3", "2.4", "2.5" (EXPERIMENTAL)	

We are going to deprecate the attribute `freeCompilerArgs` in future releases. If you miss some option in the Kotlin Gradle DSL, please, file an issue.

Example of additional arguments usage via freeCompilerArgs

Use the `freeCompilerArgs` attribute to supply additional (including experimental) compiler arguments. You can add a single argument to this attribute or a list of arguments:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

kotlin {
    compilerOptions {
        // Specifies the version of the Kotlin API and the JVM target
        apiVersion.set(KotlinVersion.KOTLIN_2_4)
        jvmTarget.set(JvmTarget.JVM_1_8)

        // Single experimental argument
        freeCompilerArgs.add("-Xexport-kdoc")

        // Single additional argument
        freeCompilerArgs.add("-Xno-param-assertions")

        // List of arguments
        freeCompilerArgs.addAll(
            listOf(
                "-Xno-receiver-assertions",
                "-Xno-call-assertions"
            )
        )
    }
}
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        // Specifies the version of the Kotlin API and the JVM target
        apiVersion = KotlinVersion.KOTLIN_2_4
        jvmTarget = JvmTarget.JVM_1_8

        // Single experimental argument
        freeCompilerArgs.add("-Xexport-kdoc")

        // Single additional argument, can be a key-value pair
        freeCompilerArgs.add("-Xno-param-assertions")

        // List of arguments
        freeCompilerArgs.addAll(["-Xno-receiver-assertions", "-Xno-call-assertions"])
    }
}
```

The `freeCompilerArgs` attribute is available at the extension, target, and compilation unit (task) levels.

Example of setting languageVersion

To set a language version, use the following syntax:

Kotlin

```
kotlin {
    compilerOptions {
        languageVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_4)
    }
}
```

Groovy

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_4
    }
```

Also, see Types for compiler options.

Attributes specific to JavaScript

Name	Description	Possible values	Default value
`friendModulesDisabled`	Disable internal declaration export		`false`
`main`	Specify whether the `main` function should be called upon execution	`JsMainFunctionExecutionMode.CALL`, `JsMainFunctionExecutionMode.NO_CALL`	`JsMainFunctionExecutionMode.CALL`
`moduleKind`	The kind of JS module generated by the compiler	`JsModuleKind.MODULE_AMD`, `JsModuleKind.MODULE_PLAIN`, `JsModuleKind.MODULE_ES`, `JsModuleKind.MODULE_COMMONJS`, `JsModuleKind.MODULE_UMD`	`null`
`sourceMap`	Generate source map		`false`
`sourceMapEmbedSources`	Embed source files into the source map	`JsSourceMapEmbedMode.SOURCE_MAP_SOURCE_CONTENT_INLINING`, `JsSourceMapEmbedMode.SOURCE_MAP_SOURCE_CONTENT_NEVER`, `JsSourceMapEmbedMode.SOURCE_MAP_SOURCE_CONTENT_ALWAYS`	`null`

Name	Description	Possible values	Default value
`sourceMapNamesPolicy`	Add variable and function names that you declared in Kotlin code into the source map. For more information on the behavior, see our compiler reference	`JsSourceMapNamesPolicy.SOURCE_MAP_NAMES_POLICY_FQ_NAMES`, `JsSourceMapNamesPolicy.SOURCE_MAP_NAMES_POLICY_SIMPLE_NAMES`, `JsSourceMapNamesPolicy.SOURCE_MAP_NAMES_POLICY_NO`	`null`
`sourceMapPrefix`	Add the specified prefix to paths in the source map		`null`
`target`	Generate JS files for specific ECMA version	`"es5"`, `"es2015"`	`"es5"`
`useEsClasses`	Let generated JavaScript code use ES2015 classes. Enabled by default in case of ES2015 target usage		`null`

Types for compiler options

Some of the `compilerOptions` use the new types instead of the `String` type:

Option	Type	Example
`jvmTarget`	`JvmTarget`	`compilerOptions.jvmTarget.set(JvmTarget.JVM_11)`
`apiVersion` and `languageVersion`	`KotlinVersion`	`compilerOptions.languageVersion.set(KotlinVersion.KOTLIN_2_4)`
`main`	`JsMainFunctionExecutionMode`	`compilerOptions.main.set(JsMainFunctionExecutionMode.NO_CALL)`
`moduleKind`	`JsModuleKind`	`compilerOptions.moduleKind.set(JsModuleKind.MODULE_ES)`
`sourceMapEmbedSources`	`JsSourceMapEmbedMode`	`compilerOptions.sourceMapEmbedSources.set(JsSourceMapEmbedMode.SOURCE_MAP_SOURCE_CONTENT_INLINING)`
`sourceMapNamesPolicy`	`JsSourceMapNamesPolicy`	`compilerOptions.sourceMapNamesPolicy.set(JsSourceMapNamesPolicy.SOURCE_MAP_NAMES_POLICY_FQ_NAMES)`

What's next?

Learn more about:

- Kotlin Multiplatform DSL reference.
- Incremental compilation, caches support, build reports, and the Kotlin daemon.
- Gradle basics and specifics.
- Support for Gradle plugin variants.

Compilation and caches in the Kotlin Gradle plugin

On this page, you can learn about the following topics:

- Incremental compilation
- Gradle build cache support
- Gradle configuration cache support
- The Kotlin daemon and how to use it with Gradle
- Rolling back to the previous compiler
- Defining Kotlin compiler execution strategy
- Kotlin compiler fallback strategy
- Trying the latest language version
- Build reports

Incremental compilation

The Kotlin Gradle plugin supports incremental compilation, which is enabled by default for Kotlin/JVM and Kotlin/JS projects. Incremental compilation tracks changes to files in the classpath between builds so that only the files affected by these changes are compiled. This approach works with Gradle's build cache and supports compilation avoidance.

For Kotlin/JVM, incremental compilation relies on classpath snapshots, which capture the API structure of modules to determine when recompilation is necessary. To optimize the overall pipeline, the Kotlin compiler uses two types of classpath snapshots:

- **Fine-grained snapshots:** include detailed information about class members, such as properties or functions. When member-level changes are detected, the Kotlin compiler recompiles only the classes that depend on the modified members. To maintain performance, the Kotlin Gradle plugin creates coarse-grained snapshots for `.jar` files in the Gradle cache.
- **Coarse-grained snapshots:** only contain the class ABI hash. When a part of ABI changes, the Kotlin compiler recompiles all classes that depend on the changed class. This is useful for classes that change infrequently, such as external libraries.

Kotlin/JS projects use a different incremental compilation approach based on history files.

There are several ways to disable incremental compilation:

- Set `kotlin.incremental=false` for Kotlin/JVM.
- Set `kotlin.incremental.js=false` for Kotlin/JS projects.

- Use `-Pkotlin.incremental=false` or `-Pkotlin.incremental.js=false` as a command line parameter.

  The parameter should be added to each subsequent build.

When you disable incremental compilation, incremental caches become invalid after the build. The first build is never incremental.

> Sometimes problems with incremental compilation become visible several rounds after the failure occurs. Use build reports to track the history of changes and compilations. This can help you to provide reproducible bug reports.

To learn more about how our current incremental compilation approach works and compares to the previous one, see our blog post.

Gradle build cache support

The Kotlin plugin uses the Gradle build cache, which stores the build outputs for reuse in future builds.

To disable caching for all Kotlin tasks, set the system property `kotlin.caching.enabled` to `false` (run the build with the argument `-Dkotlin.caching.enabled=false`).

Gradle configuration cache support

The Kotlin plugin uses the Gradle configuration cache, which speeds up the build process by reusing the results of the configuration phase for subsequent builds.

See the Gradle documentation to learn how to enable the configuration cache. After you enable this feature, the Kotlin Gradle plugin automatically starts using it.

The Kotlin daemon and how to use it with Gradle

The Kotlin daemon:

- Runs with the Gradle daemon to compile the project.
- Runs separately from the Gradle daemon when you compile the project with an IntelliJ IDEA built-in build system.

The Kotlin daemon starts at the Gradle execution stage when one of the Kotlin compile tasks starts to compile sources. The Kotlin daemon stops either with the Gradle daemon or after two idle hours with no Kotlin compilation.

The Kotlin daemon uses the same JDK that the Gradle daemon does.

Setting Kotlin daemon's JVM arguments

Each of the following ways to set arguments overrides the ones that came before it:

- Gradle daemon arguments inheritance
- `kotlin.daemon.jvm.options` system property
- `kotlin.daemon.jvmargs` property
- `kotlin` extension

- Specific task definition

Gradle daemon arguments inheritance

By default, the Kotlin daemon inherits a specific set of arguments from the Gradle daemon but overwrites them with any JVM arguments specified directly for the Kotlin daemon. For example, if you add the following JVM arguments in the `gradle.properties` file:

```
org.gradle.jvmargs=-Xmx1500m -Xms500m -XX:MaxMetaspaceSize=1g
```

These arguments are then added to the Kotlin daemon's JVM arguments:

```
-Xmx1500m -XX:ReservedCodeCacheSize=320m -XX:MaxMetaspaceSize=1g -XX:UseParallelGC -ea -
XX:+UseCodeCacheFlushing -XX:+HeapDumpOnOutOfMemoryError -Djava.awt.headless=true -
Djava.rmi.server.hostname=127.0.0.1 --add-exports=java.base/sun.nio.ch=ALL-UNNAMED
```

To learn more about the Kotlin daemon's default behavior with JVM arguments, see Kotlin daemon's behavior with JVM arguments.

kotlin.daemon.jvm.options system property

If the Gradle daemon's JVM arguments have the `kotlin.daemon.jvm.options` system property – use it in the `gradle.properties` file:

```
org.gradle.jvmargs=-Dkotlin.daemon.jvm.options=-Xmx1500m,Xms500m
```

When passing arguments, follow these rules:

- Use the minus sign `-` **only** before the arguments `Xmx`, `XX:MaxMetaspaceSize`, and `XX:ReservedCodeCacheSize`.
- Separate arguments with commas (`,`) *without* spaces. Arguments that come after a space will be used for the Gradle daemon, not for the Kotlin daemon.

Gradle ignores these properties if all the following conditions are satisfied:

- Gradle is using JDK 1.9 or higher.
- The version of Gradle is between 7.0 and 7.1.1 inclusively.
- Gradle is compiling Kotlin DSL scripts.
- The Kotlin daemon isn't running.

To overcome this, upgrade Gradle to the version 7.2 (or higher) or use the `kotlin.daemon.jvmargs` property – see the following section.

kotlin.daemon.jvmargs property

You can add the `kotlin.daemon.jvmargs` property in the `gradle.properties` file:

```
kotlin.daemon.jvmargs=-Xmx1500m -Xms500m
```

Note that if you don't specify the `ReservedCodeCacheSize` argument here or in Gradle's JVM arguments, the Kotlin Gradle plugin applies a default value of `320m`:

```
-Xmx1500m -XX:ReservedCodeCacheSize=320m -Xms500m
```

kotlin extension

You can specify arguments in the `kotlin` extension:

Kotlin

```
kotlin {
    kotlinDaemonJvmArgs = listOf("-Xmx486m", "-Xms256m", "-XX:+UseParallelGC")
}
```

Groovy

```
kotlin {
    kotlinDaemonJvmArgs = ["-Xmx486m", "-Xms256m", "-XX:+UseParallelGC"]
}
```

Specific task definition

You can specify arguments for a specific task:

Kotlin

```
tasks.withType<CompileUsingKotlinDaemon>().configureEach {
    kotlinDaemonJvmArguments.set(listOf("-Xmx486m", "-Xms256m", "-XX:+UseParallelGC"))
}
```

Groovy

```
tasks.withType(CompileUsingKotlinDaemon).configureEach { task →
    task.kotlinDaemonJvmArguments = ["-Xmx1g", "-Xms512m"]
}
```

In this case, a new Kotlin daemon instance can start on task execution. Learn more about Kotlin daemon's behavior with JVM arguments.

Kotlin daemon's behavior with JVM arguments

When configuring the Kotlin daemon's JVM arguments, note that:

- It is expected to have multiple instances of the Kotlin daemon running at the same time when different subprojects or tasks have different sets of JVM arguments.

- A new Kotlin daemon instance starts only when Gradle runs a related compilation task and existing Kotlin daemons do not have the same set of JVM arguments. Imagine that your project has a lot of subprojects. Most of them require some heap memory for a Kotlin daemon, but one module requires a lot (though it is rarely compiled). In this case, you should provide a different set of JVM arguments for such a module, so a Kotlin daemon with a larger heap size would start only for developers who touch this specific module.

  > If you are already running a Kotlin daemon that has enough heap size to handle the compilation request, even if other requested JVM arguments are different, this daemon will be reused instead of starting a new one.

If the following arguments aren't specified, the Kotlin daemon inherits them from the Gradle daemon:

- `-Xmx`
- `-XX:MaxMetaspaceSize`
- `-XX:ReservedCodeCacheSize`. If not specified or inherited, the default value is `320m`.

The Kotlin daemon has the following default JVM arguments:

- `-XX:UseParallelGC`. This argument is only applied if no other garbage collector is specified.
- `-ea`
- `-XX:+UseCodeCacheFlushing`
- `-Djava.awt.headless=true`
- `-D{java.servername.property}={localhostip}`
- `--add-exports=java.base/sun.nio.ch=ALL-UNNAMED`. This argument is only applied for JDK versions 16 or higher.

> The list of default JVM arguments for the Kotlin daemon may vary between versions. You can use a tool like VisualVM to check the actual settings of a running JVM process, like the Kotlin daemon.

Rolling back to the previous compiler

From Kotlin 2.0.0, the K2 compiler is used by default.

To use the previous compiler from Kotlin 2.0.0 onwards, either:

- In your `build.gradle.kts` file, set your language version to `1.9`.

  OR

- Use the following compiler option: `-language-version 1.9`.

To learn more about the benefits of the K2 compiler, see the K2 compiler migration guide.

Trying the latest language version

Starting with Kotlin 2.0.0, to try the latest language version, set the `kotlin.experimental.tryNext` property in your `gradle.properties` file. When you use this property, the Kotlin Gradle plugin increments the language version to one above the default value for your Kotlin version. For example, in Kotlin 2.0.0, the default language version is 2.0, so the property configures language version 2.1.

Alternatively, you can run the following command:

```
./gradlew assemble -Pkotlin.experimental.tryNext=true
```

In build reports, you can find the language version used to compile each task.

Build reports

Build reports contain the durations of different compilation phases and any reasons why compilation couldn't be incremental. Use build reports to investigate performance issues when the compilation time is too long or when it differs for the same project.

Kotlin build reports help you to investigate problems with build performance more efficiently than with Gradle build scans that have a single Gradle task as the unit of granularity.

There are two common cases that analyzing build reports for long-running compilations can help you resolve:

- The build wasn't incremental. Analyze the reasons and fix underlying problems.
- The build was incremental but took too much time. Try reorganizing source files — split big files, save separate classes in different files, refactor large classes, declare top-level functions in different files, and so on.

Build reports also show the Kotlin version used in the project. In addition, starting with Kotlin 1.9.0, you can see which compiler was used to compile the code in your Gradle build scans.

Learn how to read build reports and about how JetBrains uses build reports.

Enabling build reports

To enable build reports, declare where to save the build report output in `gradle.properties`:

```
kotlin.build.report.output=file
```

The following values and their combinations are available for the output:

Option	Description
`file`	Saves build reports in a human-readable format to a local file. By default, it's `${project_folder}/build/reports/kotlin-build/${project_name}-timestamp.txt`
`single_file`	Saves build reports in a format of an object to a specified local file.
`build_scan`	Saves build reports in the `custom values` section of the build scan. Note that the Gradle Enterprise plugin limits the number of custom values and their length. In big projects, some values could be lost.
`http`	Posts build reports using HTTP(S). The POST method sends metrics in JSON format. You can see the current version of the sent data in the Kotlin repository. You can find samples of HTTP endpoints in this blog post

Option	Description
`json`	Saves build reports in JSON format to a local file. By default, it's `${project_folder}/build/reports/kotlin-build/${project_name}-build-<date-time>-<index>.json`.

Here's a list of available options for `kotlin.build.report`:

```
# Required outputs. Any combination is allowed
kotlin.build.report.output=file,single_file,http,build_scan,json

# Mandatory if single_file output is used. Where to put reports
# Use instead of the deprecated `kotlin.internal.single.build.metrics.file` property
kotlin.build.report.single_file=my/directory/path/some_filename

# Optional. Output directory for file-based or JSON reports. Default: build/reports/kotlin-build/
kotlin.build.report.file.output_dir=kotlin-reports

# Optional. Label for marking your build report (for example, debug parameters)
kotlin.build.report.label=some_label
```

Options, applicable only to HTTP:

```
# Mandatory. Where to post HTTP(S)-based reports
kotlin.build.report.http.url=http://127.0.0.1:8080

# Optional. User and password if the HTTP endpoint requires authentication
kotlin.build.report.http.user=someUser
kotlin.build.report.http.password=somePassword

# Optional. Add a Git branch name of a build to a build report
kotlin.build.report.http.include_git_branch.name=true|false

# Optional. Add compiler arguments to a build report
# If a project contains many modules, its compiler arguments in the report can be very heavy and not that helpful
kotlin.build.report.include_compiler_arguments=true|false
```

Limit of custom values

To collect build scans' statistics, Kotlin build reports use Gradle's custom values. Both you and different Gradle plugins can write data to custom values. The number of custom values has a limit. See the current maximum custom value count in the Build scan plugin docs.

If you have a big project, a number of such custom values may be quite big. If this number exceeds the limit, you can see the following message in the logs:

```
Maximum number of custom values (1,000) exceeded
```

To reduce the number of custom values the Kotlin plugin produces, you can use the following property in `gradle.properties`:

```
kotlin.build.report.build_scan.custom_values_limit=500
```

Switching off collecting project and system properties

HTTP build statistic logs can contain some project and system properties. These properties can change builds' behavior, so it's useful to log them in build statistics. These properties can store sensitive data, for example, passwords or a project's full path.

You can disable collection of these statistics by adding the `kotlin.build.report.http.verbose_environment` property to your `gradle.properties`.

JetBrains doesn't collect these statistics. You choose a place where to store your reports.

What's next?

Learn more about:

- Gradle basics and specifics.
- Support for Gradle plugin variants.
- Compiler execution strategy

Experimental

Binary compatibility validation in the Kotlin Gradle plugin

Binary compatibility validation helps library authors ensure that users don't break their code when upgrading to newer versions. It's important not only for delivering a smooth upgrade experience, but also for building long-term trust with users and encouraging continued adoption of the library.

Binary compatibility means that the compiled bytecode of two versions of a library can run interchangeably without needing recompilation.

The Kotlin Gradle plugin includes support for binary compatibility validation. The plugin generates Application Binary Interface (ABI) dumps from the current code and compares them with previous dumps to highlight differences. You can review these changes to find any potentially binary-incompatible modifications and take action to address them.

How to enable

To enable binary compatibility validation, in your `build.gradle.kts` file, add an `abiValidation {}` block. If you have no custom configuration, you can use the `abiValidation()` function instead:

Kotlin

```
kotlin {
    @OptIn(org.jetbrains.kotlin.gradle.dsl.abi.ExperimentalAbiValidation::class)
    abiValidation()
}
```

Groovy

```
kotlin {
    abiValidation()
}
```

The KGP creates the necessary Gradle tasks. If your project has multiple modules where you want to check for binary compatibility, configure each module separately.

Check for binary compatibility issues

To check for potentially binary incompatible issues after making changes to your code, run the `checkKotlinAbi` Gradle task in IntelliJ IDEA or use the following command in your project directory:

```
./gradlew checkKotlinAbi
```

The task compares ABI dumps and prints any detected differences as errors. Check the output carefully to see if you need to make changes to your code to preserve binary compatibility.

By default, when binary compatibility validation is enabled in your project and you run the `check` task, Gradle also runs the `checkKotlinAbi` task.

Update reference ABI dump

To update the reference ABI dump that Gradle uses to check your latest changes, run the `updateKotlinAbi` task in IntelliJ IDEA or use the following command in your project directory:

```
./gradlew updateKotlinAbi
```

Only update the reference dump when you're confident your changes maintain binary compatibility with the previous version.

Configure filters

You can define filters to control which classes, properties, and functions the ABI dump includes. Use the `filters {}` block to add exclusion and inclusion rules with `excluded {}` and `included {}` blocks respectively.

Gradle includes a declaration in the ABI dump only if it doesn't match any exclusion rule. When inclusion rules are defined, the declaration must either match one of them or have at least one member that does.

A rule can be based on:

- The fully qualified name of a class, property, or function (`byNames`).
- The name of an annotation that has BINARY or RUNTIME retention (`annotatedWith`).

You can use wildcards `**`, `*`, and `?` in your rules for names:

- `**` matches zero or more characters, including periods.
- `*` matches zero or more characters excluding periods. Use this to specify a single class name.
- `?` matches exactly one character.

For example:

Kotlin

```
kotlin {
    @OptIn(org.jetbrains.kotlin.gradle.dsl.abi.ExperimentalAbiValidation::class)
    abiValidation {
        filters {
            excluded {
                byNames.add("**.InternalUtils")
                annotatedWith.add("com.example.annotations.InternalApi")
            }

            included {
                byNames.add("com.example.api.**")
                annotatedWith.add("com.example.annotations.PublicApi")
            }
        }
    }
}
```

Groovy

```
kotlin {
    abiValidation {
        filters {
            excluded {
                byNames.add("**.InternalUtils")
                annotatedWith.add("com.example.annotations.InternalApi")
            }

            included {
                byNames.add("com.example.api.**")
                annotatedWith.add("com.example.annotations.PublicApi")
            }
        }
    }
}
```

This example:

- Excludes:
  - The `InternalUtils` class.
  - Declarations annotated with `@InternalApi`.
- Includes:

- Everything in the `com.example.api` package.
- Declarations annotated with `@PublicApi`.

To learn more about filtering, see the Kotlin Gradle plugin API reference.

Prevent inferred changes for unsupported targets

In multiplatform projects, if your host system can't compile all targets, the Kotlin Gradle plugin tries to infer ABI changes from the available targets. This helps avoid false failures when you later switch to a host that supports more targets.

To disable this behavior, add the following to your `build.gradle.kts` file:

Kotlin

```
kotlin {
    @OptIn(org.jetbrains.kotlin.gradle.dsl.abi.ExperimentalAbiValidation::class)
    abiValidation {
        keepLocallyUnsupportedTargets.set(false)
    }
}
```

Groovy

```
kotlin {
    abiValidation {
        keepLocallyUnsupportedTargets = false
    }
}
```

If a target is unsupported and inference is disabled, the `checkKotlinAbi` task fails because it can't generate a complete ABI dump. This behavior may be useful if you'd prefer the task to fail rather than risk missing a binary-incompatible change.

Include publications from the `maven-publish` plugin

By default, binary compatibility validation uses Kotlin compilation outputs to generate ABI dumps. For this reason, the generated ABI dumps might not reflect the final published artifacts. For example, when you use the `maven-publish` plugin, post-processing steps such as relocation can modify artifacts after compilation.

To make sure the ABI dumps accurately reflect the artifacts published by the `maven-publish` plugin, add the following to your `build.gradle.kts` file:

Kotlin

```
kotlin {
    @OptIn(org.jetbrains.kotlin.gradle.dsl.abi.ExperimentalAbiValidation::class)
    abiValidation {
        binariesSource.set(MAVEN_PUBLICATIONS)
    }
}
```

Groovy

```
kotlin {
    abiValidation {
        binariesSource = MAVEN_PUBLICATIONS
    }
}
```

Because Kotlin/Android projects and multiplatform projects with an Android target don't publish JAR files, this feature doesn't apply to them.

Support for Gradle plugin variants

Gradle 7.0 introduced a new feature for Gradle plugin authors — plugins with variants. This feature makes it easier to add support for latest Gradle features while maintaining compatibility with older Gradle versions. Learn more about variant selection in Gradle.

With Gradle plugin variants, the Kotlin team can ship different Kotlin Gradle plugin (KGP) variants for different Gradle versions. The goal is to support the base Kotlin compilation in the `main` variant, which corresponds to the oldest supported versions of Gradle. Each variant will have implementations for Gradle features from a corresponding release. The latest variant will support the latest Gradle feature set. With this approach, it is possible to extend support for older Gradle versions with limited functionality.

Currently, there are the following variants of the Kotlin Gradle plugin:

Variant's name	Corresponding Gradle versions
`main`	7.6.0–7.6.3
`gradle80`	8.0–8.0.2
`gradle81`	8.1.1
`gradle82`	8.2.1–8.4
`gradle85`	8.5
`gradle86`	8.6-8.7

Variant's name	Corresponding Gradle versions
`gradle88`	8.8-8.10
`gradle811`	8.11-8.12
`gradle813`	8.13-8.14
`gradle96`	9.6-9.7

In future Kotlin releases, more variants will be added.

To check which variant your build uses, enable the `--info` log level and find a string in the output starting with `Using Kotlin Gradle plugin`, for example, `Using Kotlin Gradle plugin main variant`.

Troubleshooting

Here are workarounds for some known issues with variant selection in Gradle:

- ResolutionStrategy in pluginManagement is not working for plugins with multivariants
- Plugin variants are ignored when a plugin is added as the `buildSrc` common dependency

Gradle can't select a KGP variant in a custom configuration

This is an expected situation that Gradle can't select a KGP variant in a custom configuration. If you use a custom Gradle configuration:

Kotlin

```
configurations.register("customConfiguration") {
    // ...
}
```

Groovy

```
configurations.register("customConfiguration") {
    // ...
}
```

And want to add a dependency on the Kotlin Gradle plugin, for example:

Kotlin

```
dependencies {
```

```
    customConfiguration("org.jetbrains.kotlin:kotlin-gradle-plugin:2.4.20")
}
```

Groovy

```
dependencies {
    customConfiguration 'org.jetbrains.kotlin:kotlin-gradle-plugin:2.4.20'
}
```

You need to add the following attributes to your `customConfiguration`:

Kotlin

```
configurations {
    customConfiguration {
        attributes {
            attribute(
                Usage.USAGE_ATTRIBUTE,
                project.objects.named(Usage.class, Usage.JAVA_RUNTIME)
            )
            attribute(
                Category.CATEGORY_ATTRIBUTE,
                project.objects.named(Category.class, Category.LIBRARY)
            )
            // If you want to depend on a specific KGP variant:
            attribute(
                GradlePluginApiVersion.GRADLE_PLUGIN_API_VERSION_ATTRIBUTE,
                project.objects.named("7.0")
            )
        }
    }
}
```

Groovy

```
configurations {
    customConfiguration {
        attributes {
            attribute(
                Usage.USAGE_ATTRIBUTE,
                project.objects.named(Usage, Usage.JAVA_RUNTIME)
            )
            attribute(
                Category.CATEGORY_ATTRIBUTE,
                project.objects.named(Category, Category.LIBRARY)
            )
            // If you want to depend on a specific KGP variant:
            attribute(
                GradlePluginApiVersion.GRADLE_PLUGIN_API_VERSION_ATTRIBUTE,
                project.objects.named('7.0')
            )
        }
    }
}
```

Otherwise, you will receive an error similar to this:

```
> Could not resolve all files for configuration ':customConfiguration'.
      > Could not resolve org.jetbrains.kotlin:kotlin-gradle-plugin:1.7.0.
        Required by:
            project :
         > Cannot choose between the following variants of org.jetbrains.kotlin:kotlin-gradle-
plugin:1.7.0:
             - gradle70RuntimeElements
             - runtimeElements
           All of them match the consumer attributes:
             - Variant 'gradle70RuntimeElements' capability org.jetbrains.kotlin:kotlin-gradle-
plugin:1.7.0:
                 - Unmatched attributes:
```

What's next?

Learn more about Gradle basics and specifics.

Maven

Maven is a build system that helps manage Kotlin-only or mixed Kotlin-Java projects and automate your building process. It works with JVM-based projects and downloads required dependencies, compiles and packages your code. Learn more about its basics and specifics on the Maven website.

Here's a general workflow when working with a Kotlin Maven project:

1. Configure your Java or Kotlin project.
2. Declare repositories.
3. Set project dependencies.
4. Configure Kotlin compiler.
5. Package your application.

To get started, you can also follow our step-by-step tutorials:

- Configure a Java project to work with Kotlin
- Test your Java Maven project with Kotlin and JUnit

> You can check out our public sample project with both Maven and Gradle build files already set up for a mixed Kotlin/Java project.

What's next?

- **Improve your debugging experience** with the `power-assert` plugin.
- **Measure test coverage and generate reports** with the `kover-maven-plugin`.
- **Configure annotation processing** with the kapt plugin.
- **Generate documentation** with the Dokka documentation engine. It supports mixed-language projects and can generate output in multiple formats, including standard Javadoc.
- **Enable OSGi support** by adding the `kotlin-osgi-bundle`.

Configure a Maven project

When you introduce Kotlin to your existing Java Maven project or create a new Kotlin Maven project, you need to add the Kotlin Maven plugin that compiles Kotlin sources and modules.

Currently, only Maven v3 is supported.

Automatic configuration

You can simplify Maven configuration in both mixed Java-Kotlin projects and in pure Kotlin projects using the `<extensions>` option. This approach saves you time because you don't need to configure the Maven compiler plugin.

To apply the Kotlin Maven plugin with `<extensions>`, update your `pom.xml` build file as follows:

1. In the `<properties>` section, define the target versions of Kotlin and the JVM:

```
<properties>
    <maven.compiler.release>17</maven.compiler.release>
    <kotlin.version>2.4.20</kotlin.version>
</properties>
```

2. In the `<build><plugins>` section, add the Kotlin Maven plugin with the enabled `<extensions>` option:

```
<build>
    <plugins>
        <!-- Kotlin compiler plugin configuration -->
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <extensions>true</extensions> <!-- Enable the extension -->
        </plugin>
        <!-- No need to configure Maven compiler plugin with extensions -->
    </plugins>
</build>
```

The `<extensions>` option:

- Registers `src/main/kotlin` and `src/test/kotlin` directories as source roots if they already exist but are not specified in the plugin configuration.
- Adds the `kotlin-stdlib` dependency if it's not already defined in the project.
- Adds `compile`, `test-compile`, `kapt`, and `test-kapt` executions to your build, bound to their appropriate lifecycle phases. So you don't need to manually set up the `<executions>` section with `<id>` and `<goals>` for `kapt`, Kotlin's `compile`, and Java's `compile` executions to ensure they run in the correct order.
- Automatically aligns the JVM target version with the Java compiler version configured in the project.

If you have a mixed Java and Kotlin project, the configuration ensures that:

- Kotlin code is compiled first.
- Java code is compiled after Kotlin and can reference Kotlin classes.
- Default Maven behavior doesn't override the plugin order.

The extension configuration replaces the whole `<executions>` section. If you need to configure an execution, see an example in Compile Kotlin and Java sources.

> If several build plugins overwrite the default lifecycle, and you have also enabled the `<extensions>` option, the last plugin in the `<build>` section has priority for lifecycle settings. All earlier changes to lifecycle settings are ignored.

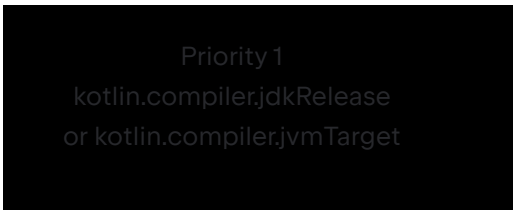
Currently, the default version of the Maven compiler plugin used with `<extensions>` is **3.10.1**. You can set a different version separately:

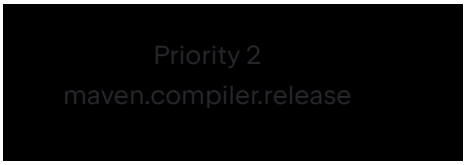
```
<build>
    <plugins>
        <!-- Kotlin compiler plugin configuration -->
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <extensions>true</extensions>
        </plugin>
        <!-- Maven compiler plugin configuration for Java classes -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.15.0</version>
        </plugin>
    </plugins>
</build>
```

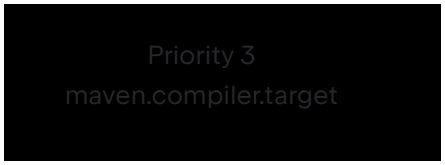
JVM target version

The `<extensions>` option ensures that the Kotlin and Maven compilers target the same bytecode version.

The Kotlin Maven plugin automatically resolves the JVM target version in the following order:







Kotlin compiler versions

The version set in `kotlin.compiler.jdkRelease` or `kotlin.compiler.jvmTarget` property takes priority if either is defined in the project.

Keep in mind that these Kotlin compiler options behave differently:

Kotlin compiler option	Controls bytecode version of the output	Limits API to specified JDK
`kotlin.compiler.jvmTarget`	Yes	No restrictions on JDK APIs in your code
`kotlin.compiler.jdkRelease`	Yes	Yes - only specific API version is allowed (equivalent to Java's `--release` compiler option)

Do not set different JDK options for `kotlin.compiler.jdkRelease` and `kotlin.compiler.jvmTarget` at the same time. Otherwise, you'll get an error.

Maven compiler versions

- If neither the `kotlin.compiler.jdkRelease` nor the `kotlin.compiler.jvmTarget` option is set, the plugin takes the `maven.compiler.release` version.

  The `maven.compiler.release` version can be defined either as a project property or within the `maven-compiler-plugin` configuration.

- If the Maven release version isn't set, the plugin takes the `maven.compiler.target` version.

  It can be defined either as a project property or within the `maven-compiler-plugin` configuration.

Keep in mind that `target` and `release` options of the Maven compiler behave differently:

Maven compiler option	Sets Kotlin's `jvmTarget`	Sets Kotlin's `jdkRelease`	Limits API to specified JDK
`maven.compiler.target`	Yes	No	No - the build's JDK classpath stays visible
`maven.compiler.release`	Yes	Yes	Yes - to the specific API version only

The `<extensions>` option only checks project-level properties and the global `maven-compiler-plugin` configuration. It doesn't check the configurations defined in the plugin's `<executions>` section.

Manual configuration

Without enabling `<extensions>` in the Kotlin Maven plugin, you need to configure the project manually to ensure that the source code compiles correctly.

You can set up your Maven project to compile a combination of Java and Kotlin sources or Kotlin-only sources.

Compile Kotlin and Java sources

To compile a project with both Kotlin and Java source files, ensure that the Kotlin compiler runs before the Java compiler.

The Java compiler can't see Kotlin declarations until they are compiled into `.class` files. If your Java code uses Kotlin classes, those classes must be compiled first to avoid `cannot find symbol` errors.

Maven determines plugin execution order based on two main factors:

- The order of plugin declarations in the `pom.xml` file.
- Built-in default executions, such as `default-compile` and `default-testCompile`, which always run before user-defined executions, regardless of their position in the `pom.xml` file.

To control the execution order:

- Declare `kotlin-maven-plugin` before `maven-compiler-plugin`.
- Disable the Java compiler plugin's default executions.
- Add custom executions to control compile phases explicitly.



You can use the special `none` phase in Maven to disable a default execution.

To apply the Kotlin Maven plugin, update your `pom.xml` build file as follows:

```
<build>
    <plugins>
        <!-- Kotlin compiler plugin configuration -->
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <executions>
                <execution>
                    <id>kotlin-compile</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                    <configuration>
                        <sourceDirs>
                            <sourceDir>${project.basedir}/src/main/kotlin</sourceDir>
                            <!-- Ensure Kotlin code can reference Java code -->
                            <sourceDir>${project.basedir}/src/main/java</sourceDir>
                        </sourceDirs>
                    </configuration>
                </execution>
                <execution>
                    <id>kotlin-test-compile</id>
                    <phase>test-compile</phase>
                    <goals>
                        <goal>test-compile</goal>
                    </goals>
                    <configuration>
                        <sourceDirs>
                            <sourceDir>${project.basedir}/src/test/kotlin</sourceDir>
                            <sourceDir>${project.basedir}/src/test/java</sourceDir>
                        </sourceDirs>
                    </configuration>
                </execution>
            </executions>
        </plugin>

        <!-- Maven compiler plugin configuration -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.15.0</version>
            <executions>
                <!-- Disable default executions -->
                <execution>
                    <id>default-compile</id>
                    <phase>none</phase>
                </execution>
                <execution>
                    <id>default-testCompile</id>
                    <phase>none</phase>
                </execution>

                <!-- Define custom executions -->
                <execution>
                    <id>java-compile</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
```

```
                    <id>java-test-compile</id>
                    <phase>test-compile</phase>
                    <goals>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

This configuration ensures that:

- Kotlin code is compiled first.
- Java code is compiled after Kotlin and can reference Kotlin classes.
- Default Maven behavior doesn't override the plugin order.

For more details on how Maven handles plugin executions, see Guide to default plugin execution IDs in the official Maven documentation.

Compile Kotlin-only sources

To compile a project with just Kotlin source files, declare source roots and configure the Kotlin Maven plugin:

1. Specify the source directories in the `<build>` section:

```
<build>
    <sourceDirectory>src/main/kotlin</sourceDirectory>
    <testSourceDirectory>src/test/kotlin</testSourceDirectory>
</build>
```

2. Ensure that the Kotlin Maven plugin is applied:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <executions>
                <execution>
                    <id>compile</id>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>test-compile</id>
                    <goals>
                        <goal>test-compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Set JDK version

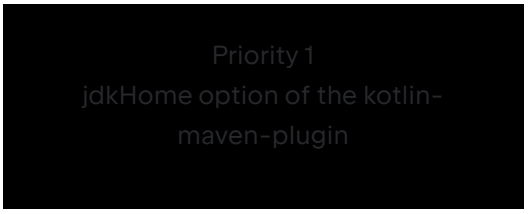
Kotlin supports Maven Toolchains that help you manage the JDK version in your build.

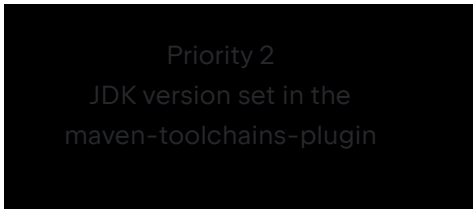
If you configure the `maven-toolchains-plugin` in your build, you can specify the JDK version used for Kotlin compilation, independent of the JVM version running Maven (set in the `JAVA_HOME` path). The Kotlin Maven plugin then automatically picks up the selected JDK toolchain.

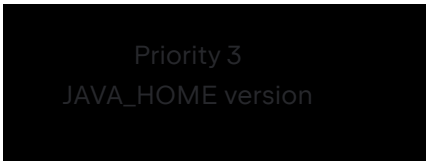
This allows you to configure a single toolchain that controls the JDK used across all plugins in the build, including Kotlin compilation. For example:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-toolchains-plugin</artifactId>
    <version>3.2.0</version>
    <executions>
        <execution>
            <goals>
                <goal>toolchain</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <toolchains>
            <jdk>
                <version>21</version>
            </jdk>
        </toolchains>
    </configuration>
</plugin>
```

Keep in mind the priority of different ways to set up the JDK version:







- The JDK version set in the `jdkHome` option of the `kotlin-maven-plugin` configuration always takes precedence over the toolchain version.
- The JDK version in the `maven-toolchains-plugin` overrides the JDK version set in the `JAVA_HOME` path.

You can also use a plugin-specific `<jdkToolchain>` option to directly set the JDK version in the toolchain of the `kotlin-maven-plugin`. Compared to using the `maven-toolchains-plugin`, this parameter only affects Kotlin compilation and has no impact on other plugins in the build.

> Currently, setting up the `maven-toolchains-plugin` to use a specific JDK version does not affect the `kapt` and `test-kapt` goals of the `kotlin-maven-plugin`. Instead, set the necessary version in the `JAVA_HOME` path.

Configure Java modules (JPMS)

The Kotlin Maven plugin supports the Java Platform Module System (JPMS), so you can compile Kotlin code alongside a `module-info.java` descriptor and consume the resulting module like other Java modules.

You don't need any extra JPMS-specific options in the build file. Just configure the Kotlin compiler before Maven.

When a `module-info.java` descriptor is present, the Kotlin compiler uses it as a source file and compiles against its module path instead of the classpath. The Kotlin compiler reads the descriptor to resolve the module graph, and the Maven compiler then compiles it into a `module-info.class` file.

To configure a Java module, create the `module-info.java` file in the `${project.basedir}/src/main/java` directory. In the module descriptor, declare all the dependencies that your module requires and the packages it exports. For example:

```
module org.example.myapp {
    requires transitive kotlin.stdlib;
    requires java.net.http;

    exports org.example.myapp;
}
```

Keep in mind that:

- Your Java module can only use what you declare. Since compilation uses the module path instead of the classpath, the descriptor should include all dependencies that your Kotlin code uses: the standard library, JDK modules (except for `java.base`), and other libraries. Otherwise, you may get `Unresolved reference` errors.
- For a module, the package name in Kotlin files must match the package name from `module-info.java` to avoid the `Package is empty or does not exist` build failures.
- The `pom.xml` build file should be configured so that Kotlin is compiled before Java. If you use automatic project configuration, the `<extensions>` option already ensures this.

What's next?

Set dependencies in your Kotlin Maven project

Set repositories and dependencies in a Maven project

For your Kotlin Maven project, you can configure where Maven looks for artifacts beyond the default Maven Central repository and define the libraries your project depends on.

Declare repositories

By default, the `mavenCentral` repository is available for all Maven projects. To access artifacts in other repositories, specify a custom ID for the repository name and its URL in the `<repositories>` section:

```
<repositories>
    <repository>
        <id>spring-repo</id>
        <url>https://repo.spring.io/release</url>
    </repository>
</repositories>
```

If you declare `mavenLocal()` as a repository in a Gradle project, you may experience problems when switching between Gradle and Maven projects. For more information, see Declare repositories.

In general, to add a dependency on a library, you should declare a new `<dependency>` entry in the `<dependencies>` section:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlinx</groupId>
        <artifactId>kotlinx-serialization-json</artifactId>
        <version>1.11.0</version>
    </dependency>
</dependencies>
```

Set dependencies

Dependency on the standard library

Kotlin has an extensive standard library that you can use in your applications. You can add the standard library dependency manually or enable the `<extensions>` option to set it up automatically if it's missing.

Automatic setup

You can avoid manual configuration using the `<extensions>` option provided by the Kotlin Maven plugin. It automatically adds the `kotlin-stdlib` dependency if it's not defined in the project. For example, when you create a new Kotlin Maven project or introduce Kotlin to your existing Java Maven project.

If you already declared a dependency `kotlin-stdlib`, for example, with a different version, the Kotlin Maven plugin with `<extensions>` will not overwrite it.

You can also opt out from the automatic addition of the standard library. For that, add the following to the `<properties>` section:

```
<project>
    <properties>
        <kotlin.smart.defaults.enabled>false</kotlin.smart.defaults.enabled>
    </properties>
</project>
```

This property disables not only the automatic addition of the standard library, but also the registration of source root paths. Other `<extensions>` features are not affected.

Manual configuration

To manually add Kotlin's standard library to your project, update the `dependencies` section in your `pom.xml` file with the following:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <!-- Uses kotlin.version specified in <properties/>: -->
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

If you're targeting JDK 7 or 8 with Kotlin versions older than:

- 1.8, use `kotlin-stdlib-jdk7` or `kotlin-stdlib-jdk8`, respectively.
- 1.2, use `kotlin-stdlib-jre7` or `kotlin-stdlib-jre8`, respectively.

Dependencies on test libraries

If your project uses Kotlin reflection or testing frameworks, add the relevant dependencies. Use `kotlin-reflect` for the reflection library, and `kotlin-test` and `kotlin-test-junit5` for testing libraries:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-reflect</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-test-junit5</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Dependency on a kotlinx library

For kotlinx libraries, you can either add the base artifact name or the name with a `-jvm` suffix. Refer to the library's README file on klibs.io.

For example, to add a dependency on `kotlinx.coroutines` library:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlinx</groupId>
        <artifactId>kotlinx-coroutines-core</artifactId>
        <version>1.11.0</version>
    </dependency>
</dependencies>
```

To add a dependency on the `kotlinx-datetime` library:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlinx</groupId>
        <artifactId>kotlinx-datetime-jvm</artifactId>
        <version>0.8.0</version>
    </dependency>
</dependencies>
```

Manage dependencies with a BOM

A Bill of Materials (BOM) is a special POM file that manages dependency versions in your project. This keeps related artifacts aligned and avoids version conflicts.

Kotlin publishes the `kotlin-bom` artifact, which specifies the versions of Kotlin libraries, such as `kotlin-stdlib`, `kotlin-reflect`, and `kotlin-test`, that correspond to the same Kotlin release. This is useful when other libraries in your project have transitive dependencies on Kotlin artifacts because it ensures that all dependencies resolve to the same Kotlin version.

To use the Kotlin BOM, import it in the `<dependencyManagement>` section of your `pom.xml` file as follows:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-bom</artifactId>
            <version>2.4.20</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

After importing the BOM, you can declare Kotlin dependencies in the `<dependencies>` section without specifying their versions; they come from the BOM file automatically.

- Importing a BOM doesn't add any dependencies to your project on its own. It only controls the versions of explicitly declared and transitive dependencies.
- If you still specify a `<version>` for a dependency, that value overrides the version from the BOM.

If your project publishes several libraries that are released together, you can provide your own BOM so that users can align the versions of those libraries in the same way. To learn how to author your own BOM, see the Maven documentation.

What's next?

Configure Kotlin compiler

Configure Kotlin compiler for your Maven project

The `kotlin-maven-plugin` allows you to configure the Kotlin compiler for your Maven projects. You can specify compiler options, choose an execution strategy, and enable incremental compilation.

Specify compiler options

You can specify additional options and arguments for the compiler as elements in the `<configuration>` section of the Kotlin Maven plugin node:

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <extensions>true</extensions> <!— If you want to enable automatic addition of executions to your
build —>
    <executions>...</executions>
    <configuration>
        <nowarn>true</nowarn> <!— Disable warnings —>
        <args>
            <arg>-Xjsr305=strict</arg> <!— Enable strict mode for JSR-305 annotations —>
            ...
        </args>
    </configuration>
</plugin>
```

Many of the options can also be configured through properties:

```
<project>
    <properties>
        <kotlin.compiler.languageVersion>2.4</kotlin.compiler.languageVersion>
    </properties>
</project>
```

The following attributes are supported:

Attributes specific to JVM

Name	Property name	Description	Possible values	Default value
`nowarn`		Generate no warnings	true, false	false
`languageVersion`	`kotlin.compiler.languageVersion`	Control which Kotlin language features and syntax are available during compilation. For more information, see `-language-version`.	"2.0", "2.1", "2.2", "2.3", "2.4", "2.5" (EXPERIMENTAL)	

Name	Property name	Description	Possible values	Default value
`apiVersion`	`kotlin.compiler.apiVersion`	Control which Kotlin APIs your code can use. For more information, see `-api-version`.	"2.0", "2.1", "2.2", "2.3", "2.4", "2.5" (EXPERIMENTAL)	
`sourceDirs`		The directories containing the source files to compile		The project source roots
`compilerPlugins`		Enabled compiler plugins		[]
`pluginOptions`		Options for compiler plugins		[]
`args`		Additional compiler arguments		[]
`jvmTarget`	`kotlin.compiler.jvmTarget`	Target JVM version of the generated bytecode. Controls only the bytecode version of the output and does not restrict which JDK APIs your code can use.	"1.8", "9", "10", ..., "26"	"1.8"
`jdkRelease`	`kotlin.compiler.jdkRelease`	Target JVM version. Controls the bytecode version and limits the available API to the specified JDK version, preventing accidental use of newer APIs. Equivalent to Java's `--release` compiler option.	"1.8", "9", "10", ..., "26"	
`jdkHome`	`kotlin.compiler.jdkHome`	Include a custom JDK from the specified location into the classpath instead of the default `JAVA_HOME`		
`jdkToolchain`	`kotlin.compiler.jdkToolchain`	Set the JDK version to use from the toolchain. Only affects Kotlin compilation		

Choose execution strategy

By default, Maven uses the Kotlin daemon compiler execution strategy. To switch to the "in process" strategy, set the following property in your `pom.xml` file:

```
<properties>
    <kotlin.compiler.daemon>false</kotlin.compiler.daemon>
</properties>
```

For more information about the different strategies, see Compiler execution strategy.

Enable incremental compilation

To make your builds faster, you can enable incremental compilation by adding the `kotlin.compiler.incremental` property:

```
<properties>
    <kotlin.compiler.incremental>true</kotlin.compiler.incremental>
</properties>
```

Alternatively, run your build with the `-Dkotlin.compiler.incremental=true` option.

What's next?

Package your project

Package your Maven application

To package your Kotlin application with Maven, you can create a standard JAR file or a self-contained (fat) JAR file that includes all its dependencies. Packaging your application allows you to distribute it and run it on any machine with a Java Runtime Environment (JRE).

Create JAR files

To create a small JAR file containing just the code from your module, include the following under `<build>` `<plugins>` in your Maven `pom.xml` file, where `main.class` is defined as a property and points to the main Kotlin or Java class:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>3.5.0</version>
    <configuration>
        <archive>
            <manifest>
                <addClasspath>true</addClasspath>
                <mainClass>${main.class}</mainClass>
            </manifest>
        </archive>
    </configuration>
</plugin>
```

Create self-contained JAR files

To create a self-contained JAR file containing the code from your module along with its dependencies, include the following under `<build><plugins>` in your Maven `pom.xml` file, where `main.class` is defined as a property and points to the main Kotlin or Java class:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>3.8.0</version>
    <executions>
        <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals> <goal>single</goal> </goals>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>${main.class}</mainClass>
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
            </configuration>
        </execution>
    </executions>
</plugin>
```

This self-contained JAR file can be passed directly to a JRE to run your application:

```
java -jar target/mymodule-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

Beta

Build tools API

Kotlin has the Build tools API (BTA) that simplifies how build systems integrate with the Kotlin compiler.

Adding full Kotlin support to a build system (like incremental compilation, Kotlin compiler plugins, daemons, and Kotlin Multiplatform) requires significant effort. The BTA aims to reduce this complexity by providing a unified API between build systems and the Kotlin compiler ecosystem.

The BTA defines a single entry point that build systems can implement. This removes the need to deeply integrate with internal compiler details.

The stability level differs per target: the BTA is Beta for Kotlin/JVM, and Alpha for Kotlin/JS and Kotlin/Wasm. For more details, see Build tools API (BTA). Using the BTA requires an opt-in with `@OptIn(ExperimentalBuildToolsApi::class)`.

If you're interested in the proposal or want to share feedback, see the KEEP. Follow the status of its implementation in YouTrack.

Integration with Gradle

The Kotlin Gradle plugin (KGP) uses the BTA by default for Kotlin/JVM compilation.

> We'd appreciate your feedback on your experience with the KGP in YouTrack.

Enable the BTA for Kotlin/JS, Kotlin/Wasm, and Kotlin metadata

Since Kotlin 2.4.20, the KGP can also run Kotlin/JS, Kotlin/Wasm, and Kotlin metadata compilations through the BTA. This makes the KGP interact with the compiler more consistently, and in some cases compilation becomes faster and more stable.

In Kotlin 2.4.20, these targets are available as an opt-in. To try them out, add the corresponding properties to your `gradle.properties` file:

```
kotlin.js.runViaBuildToolsApi = true
kotlin.wasm.runViaBuildToolsApi = true
kotlin.metadata.runViaBuildToolsApi = true
```

Configure different compiler versions

With the BTA, you can now use a different Kotlin compiler version than the version used by the KGP. This is useful when:

- You want to try new Kotlin features but haven't updated your build scripts yet.
- You need the latest plugin fixes but want to stay on an older compiler version for now.

Here's an example of how to configure this in your `build.gradle.kts` file:

```
import org.jetbrains.kotlin.buildtools.api.ExperimentalBuildToolsApi
import org.jetbrains.kotlin.gradle.ExperimentalKotlinGradlePluginApi

plugins {
    kotlin("jvm") version "2.4.20"
}

group = "org.jetbrains.example"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

kotlin {
    jvmToolchain(8)
    @OptIn(ExperimentalBuildToolsApi::class, ExperimentalKotlinGradlePluginApi::class)
    compilerVersion.set("2.3.21") // <-- different version than 2.4.20
}
```

Compatible Kotlin compiler and KGP versions

The BTA supports:

- The three previous major Kotlin compiler versions.
- One major version forward.

For example, in KGP 2.2.0, the supported Kotlin compiler versions are:

- 1.9.25
- 2.0.x
- 2.1.x
- 2.2.x
- 2.3.x

Limitations

Using different compiler versions together with compiler plugins may lead to Kotlin compiler exceptions. The Kotlin team plans to address this in future Kotlin releases.

Enable incremental compilation with "in process" strategy

The KGP supports three compiler execution strategies. Ordinarily, the "in-process" strategy (which runs the compiler in the Gradle daemon) doesn't support incremental compilation.

With the BTA, the "in-process" strategy now supports incremental compilation. To enable it, add the following property to your `gradle.properties` file:

```
kotlin.compiler.execution.strategy=in-process
```

Integration with Maven

The BTA enables the `kotlin-maven-plugin` to support the Kotlin daemon, which is the default compiler execution strategy. The `kotlin-maven-plugin` uses BTA by default, so there's no need to configure anything.

The BTA makes it possible to deliver more features like Incremental compilation stabilization in the future.

Kotlin daemon

The Kotlin daemon is a background process that build systems can use to improve build times by keeping the compiler and its environment ready to compile. This approach avoids starting a new Java Virtual Machine (JVM) instance and reinitializing the compiler for every compilation, leading to reduced build times for incremental builds or frequent small changes.

Some build systems have their own daemons that help reduce startup costs, such as the Gradle daemon and the Maven daemon. Using the Kotlin daemon instead reduces startup costs while also completely isolating the build system process from the compiler. This separation is useful in dynamic environments where system settings might change at runtime.

Although the Kotlin daemon has no direct user-facing interface, you can use it through build systems or the build tools API.

Kotlin daemon configuration

There are ways to configure some settings for the Kotlin daemon for Gradle or Maven.

Memory management

The Kotlin daemon is a separate process that has its own memory space, isolated from the client. By default, the Kotlin daemon tries to inherit the heap size (`-Xmx`) of the launching JVM process.

To configure specific memory limits, like `-Xmx` and `-XX:MaxMetaspaceSize` , use the following property:

Gradle

```
kotlin.daemon.jvmargs=-Xmx1500m
```

For more information, see `kotlin.daemon.jvmargs` property.

Maven

```
<kotlin.compiler.daemon.jvmArgs>-Xmx1500m</kotlin.compiler.daemon.jvmArgs>
```

Lifetime

There are two common lifetime strategies for the Kotlin daemon:

- **Attached daemon**: Shut down the daemon shortly after the client process shuts down or the daemon hasn't been used for a while. Use when the client is long-running.
- **Detached daemon**: Keep the daemon alive longer to await potential follow-up requests. Use when the client is short-lived.

To configure the lifetime strategy, you can use the following options:

Option	Description	Default value
`autoshutdownIdleSeconds`	How long the daemon should stay alive after the last compilation when a client is still connected.	2 hours
`autoshutdownUnusedSeconds`	How long a newly started daemon waits for a first client before shutting down if unused.	1 minute
`shutdownDelayMilliseconds`	How long the daemon waits to shut down after all clients disconnect.	1 second

To configure an attached daemon lifetime strategy, set `autoshutdownIdleSeconds` to a **high** value and `shutdownDelayMilliseconds` to a **low** value.

Add the following to your `gradle.properties` file:

Gradle

```
org.gradle.jvmargs=-
Dkotlin.daemon.jvm.options=autoshutdownIdleSeconds=7200,shutdownDelayMilliseconds=1000
```

Use the following command:

Maven

```
mvn package -Dkotlin.daemon.options=autoshutdownIdleSeconds=7200,shutdownDelayMilliseconds=1000
```

To configure a detached daemon lifetime strategy, set `shutdownDelayMilliseconds` to a **high** value.

Add the following to your `gradle.properties` file:

Gradle

```
org.gradle.jvmargs=-Dkotlin.daemon.jvm.options=shutdownDelayMilliseconds=7200
```

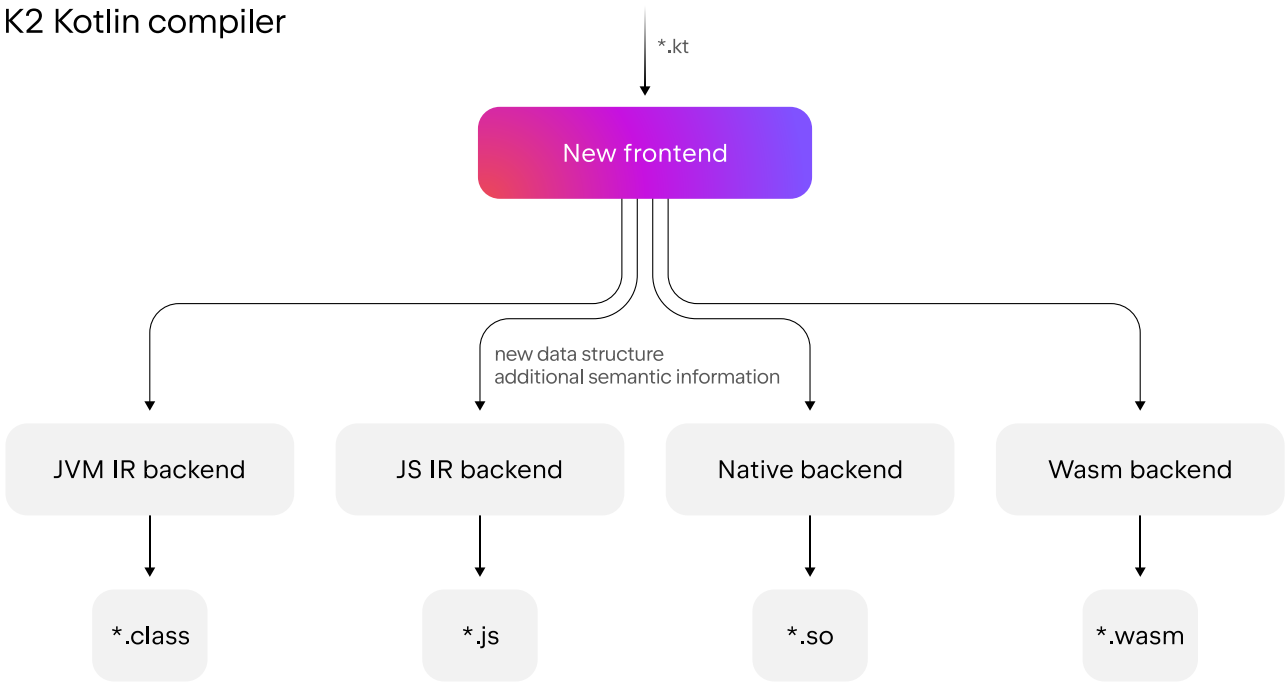
Add the following property to your `pom.xml` file:

Maven

```
<kotlin.compiler.daemon.shutdownDelayMs>7200</kotlin.compiler.daemon.shutdownDelayMs>
```

K2 compiler migration guide

As the Kotlin language and ecosystem have continued to evolve, so has the Kotlin compiler. The first step was the introduction of the new JVM and JS IR (Intermediate Representation) backends that share logic, simplifying code generation for targets on different platforms. Now, the next stage of its evolution brings a new frontend known as K2.



Kotlin K2 compiler architecture

With the arrival of the K2 compiler, the Kotlin frontend has been completely rewritten and features a new, more efficient architecture. The fundamental change the new compiler brings is the use of one unified data structure that contains more semantic information. This frontend is responsible for performing semantic analysis, call resolution, and type inference.

The new architecture and enriched data structure enables the K2 compiler to provide the following benefits:

- **Improved call resolution and type inference**. The compiler behaves more consistently and understands your code better.
- **Easier introduction of syntactic sugar for new language features**. In the future, you'll be able to use more concise, readable code when new features are introduced.
- **Faster compilation times**. Compilation times can be significantly faster.
- **Enhanced IDE performance**. IntelliJ IDEA and Android Studio use the K2 compiler to analyze your Kotlin code, increasing stability and providing performance improvements. For more information, see Support in IDEs.

This guide:

- Explains the benefits of the new K2 compiler.
- Highlights changes you might encounter during migration and how to adapt your code accordingly.
- Describes how you can roll back to the previous version.

The new K2 compiler is enabled by default starting with 2.0.0. For more information on the new features provided in Kotlin 2.0.0, as well as the new K2 compiler, see What's new in Kotlin 2.0.0.

Performance improvements

To evaluate the performance of the K2 compiler, we ran performance tests on two open-source projects: Anki-Android and Exposed. Here are the key performance improvements that we found:

- The K2 compiler brings up to 94% compilation speed gains. For example, in the Anki-Android project, clean build times were reduced from 57.7 seconds in Kotlin 1.9.23 to 29.7 seconds in Kotlin 2.0.0.
- The initialization phase is up to 488% faster with the K2 compiler. For example, in the Anki-Android project, the initialization phase for incremental builds was cut from 0.126 seconds in Kotlin 1.9.23 to just 0.022 seconds in Kotlin 2.0.0.
- The Kotlin K2 compiler is up to 376% quicker in the analysis phase compared to the previous compiler. For example, in the Anki-Android project, analysis times for incremental builds were slashed from 0.581 seconds in Kotlin 1.9.23 to only 0.122 seconds in Kotlin 2.0.0.

For more details on these improvements and to learn more about how we analyzed the performance of the K2 compiler, see our blog post.

Language feature improvements

The Kotlin K2 compiler improves language features related to smart-casting and Kotlin Multiplatform.

Smart casts

The Kotlin compiler can automatically cast an object to a type in specific cases, saving you the trouble of having to explicitly specify it yourself. This is called smart-casting. The Kotlin K2 compiler now performs smart casts in even more scenarios than before.

In Kotlin 2.0.0, we've made improvements related to smart casts in the following areas:

- Local variables and further scopes
- Type checks with the logical `or` operator
- Inline functions
- Properties with function types
- Exception handling
- Increment and decrement operators

Local variables and further scopes

Previously, if a variable was evaluated as not `null` within an `if` condition, the variable would be smart-cast. Information about this variable would then be shared further within the scope of the `if` block.

However, if you declared the variable **outside** the `if` condition, no information about the variable would be available within the `if` condition, so it couldn't be smart-cast. This behavior was also seen with `when` expressions and `while` loops.

From Kotlin 2.0.0, if you declare a variable before using it in your `if`, `when`, or `while` condition, then any information collected by the compiler about the variable will be accessible in the corresponding block for smart-casting.

This can be useful when you want to do things like extract boolean conditions into variables. Then, you can give the variable a meaningful name, which will improve your code readability and make it possible to reuse the variable later in your code. For example:

```
class Cat {
    fun purr() {
        println("Purr purr")
    }
}

fun petAnimal(animal: Any) {
    val isCat = animal is Cat
    if (isCat) {
        // In Kotlin 2.0.0, the compiler can access
        // information about isCat, so it knows that
        // animal was smart-cast to the type Cat.
        // Therefore, the purr() function can be called.
        // In Kotlin 1.9.20, the compiler doesn't know
        // about the smart cast, so calling the purr()
        // function triggers an error.
        animal.purr()
    }
}

fun main(){
    val kitty = Cat()
    petAnimal(kitty)
    // Purr purr
}
```

Type checks with the logical or operator

In Kotlin 2.0.0, if you combine type checks for objects with an `or` operator (`||`), a smart cast is made to their closest common supertype. Before this change, a smart cast was always made to the `Any` type.

In this case, you still had to manually check the object type afterward before you could access any of its properties or call its functions. For example:

```
interface Status {
    fun signal() {}
}

interface Ok : Status
interface Postponed : Status
interface Declined : Status

fun signalCheck(signalStatus: Any) {
    if (signalStatus is Postponed || signalStatus is Declined) {
        // signalStatus is smart-cast to a common supertype Status
        signalStatus.signal()
        // Prior to Kotlin 2.0.0, signalStatus is smart cast
        // to type Any, so calling the signal() function triggered an
        // Unresolved reference error. The signal() function can only
        // be called successfully after another type check:

        // check(signalStatus is Status)
        // signalStatus.signal()
    }
}
```

The common supertype is an **approximation** of a union type. Union types are not currently supported in Kotlin.

Inline functions

In Kotlin 2.0.0, the K2 compiler treats inline functions differently, allowing it to determine in combination with other compiler analyses whether it's safe to smart-cast.

Specifically, inline functions are now treated as having an implicit `callsInPlace` contract. This means that any lambda functions passed to an inline function are called in place. Since lambda functions are called in place, the compiler knows that a lambda function can't leak references to any variables contained within its function body.

The compiler uses this knowledge along with other compiler analyses to decide whether it's safe to smart-cast any of the captured variables. For example:

```
interface Processor {
    fun process()
}

inline fun inlineAction(f: () -> Unit) = f()

fun nextProcessor(): Processor? = null

fun runProcessor(): Processor? {
    var processor: Processor? = null
    inlineAction {
        // In Kotlin 2.0.0, the compiler knows that processor
        // is a local variable and inlineAction() is an inline function, so
        // references to processor can't be leaked. Therefore, it's safe
        // to smart-cast processor.

        // If processor isn't null, processor is smart-cast
        if (processor ≠ null) {
            // The compiler knows that processor isn't null, so no safe call
            // is needed
            processor.process()

            // In Kotlin 1.9.20, you have to perform a safe call:
            // processor?.process()
        }

        processor = nextProcessor()
    }

    return processor
}
```

Properties with function types

In previous versions of Kotlin, there was a bug that meant that class properties with a function type weren't smart-cast. We fixed this behavior in Kotlin 2.0.0 and the K2 compiler. For example:

```
class Holder(val provider: (() → Unit)?) {
    fun process() {
        // In Kotlin 2.0.0, if provider isn't null,
        // it is smart-cast
        if (provider ≠ null) {
            // The compiler knows that provider isn't null
            provider()

            // In 1.9.20, the compiler doesn't know that provider isn't
            // null, so it triggers an error:
            // Reference has a nullable type '(() → Unit)?', use explicit '?.invoke()' to make a
function-like call instead
        }
    }
}
```

This change also applies if you overload your `invoke` operator. For example:

```
interface Provider {
    operator fun invoke()
}

interface Processor : () → String

class Holder(val provider: Provider?, val processor: Processor?) {
    fun process() {
        if (provider ≠ null) {
            provider()
            // In 1.9.20, the compiler triggers an error:
            // Reference has a nullable type 'Provider?', use explicit '?.invoke()' to make a
function-like call instead
        }
    }
}
```

Exception handling

In Kotlin 2.0.0, we've made improvements to exception handling so that smart cast information can be passed on to `catch` and `finally` blocks. This change makes your code safer as the compiler keeps track of whether your object has a nullable type. For example:

```
fun testString() {
    var stringInput: String? = null
    // stringInput is smart-cast to String type
    stringInput = ""
    try {
        // The compiler knows that stringInput isn't null
        println(stringInput.length)
        // 0

        // The compiler rejects previous smart cast information for
        // stringInput. Now stringInput has the String? type.
        stringInput = null

        // Trigger an exception
        if (2 > 1) throw Exception()
        stringInput = ""
    } catch (exception: Exception) {
        // In Kotlin 2.0.0, the compiler knows stringInput
        // can be null, so stringInput stays nullable.
        println(stringInput?.length)
        // null

        // In Kotlin 1.9.20, the compiler says that a safe call isn't
        // needed, but this is incorrect.
    }
}
fun main() {
    testString()
}
```

Increment and decrement operators

Prior to Kotlin 2.0.0, the compiler didn't understand that the type of an object can change after using an increment or decrement operator. As the compiler couldn't accurately track the object type, your code could lead to unresolved reference errors. In Kotlin 2.0.0, this has been fixed:

```
interface Rho {
    operator fun inc(): Sigma = TODO()
}

interface Sigma : Rho {
    fun sigma() = Unit
}

interface Tau {
    fun tau() = Unit
}

fun main(input: Rho) {
    var unknownObject: Rho = input

    // Check if unknownObject inherits from the Tau interface
    // Note, it's possible that unknownObject inherits from both
    // Rho and Tau interfaces.
    if (unknownObject is Tau) {

        // Use the overloaded inc() operator from interface Rho.
        // In Kotlin 2.0.0, the type of unknownObject is smart-cast to
        // Sigma.
        ++unknownObject

        // In Kotlin 2.0.0, the compiler knows unknownObject has type
        // Sigma, so the sigma() function can be called successfully.
        unknownObject.sigma()

        // In Kotlin 1.9.20, the compiler doesn't perform a smart cast
        // when inc() is called so the compiler still thinks that
        // unknownObject has type Tau. Calling the sigma() function
        // throws a compile-time error.

        // In Kotlin 2.0.0, the compiler knows unknownObject has type
        // Sigma, so calling the tau() function throws a compile-time
        // error.
        unknownObject.tau()
        // Unresolved reference 'tau'

        // In Kotlin 1.9.20, since the compiler mistakenly thinks that
        // unknownObject has type Tau, the tau() function can be called,
        // but it throws a ClassCastException.
    }
}
```

Kotlin Multiplatform

There are improvements in the K2 compiler related to Kotlin Multiplatform in the following areas:

- Separation of common and platform sources during compilation
- Different visibility levels of expected and actual declarations

Separation of common and platform sources during compilation

Previously, the design of the Kotlin compiler prevented it from keeping common and platform source sets separate at compile time. As a consequence, common code could access platform code, which resulted in different behavior between platforms. In addition, some compiler settings and dependencies from common code used to leak into platform code.

In Kotlin 2.0.0, our implementation of the new Kotlin K2 compiler included a redesign of the compilation scheme to ensure strict separation between common and platform source sets. This change is most noticeable when you use expected and actual functions. Previously, it was possible for a function call in your common code to resolve to a function in platform code. For example:

Common code

```
fun foo(x: Any) = println("common
foo")

fun exampleFunction() {
    foo(42)
}
```

Platform code

```
// JVM
fun foo(x: Int) = println("platform foo")

// JavaScript
// There is no foo() function overload on the JavaScript
platform
```

In this example, the common code has different behavior depending on which platform it is run on:

- On the JVM platform, calling the `foo()` function in the common code results in the `foo()` function from the platform code being called as `platform foo`.
- On the JavaScript platform, calling the `foo()` function in the common code results in the `foo()` function from the common code being called as `common foo`, as there is no such function available in the platform code.

In Kotlin 2.0.0, common code doesn't have access to platform code, so both platforms successfully resolve the `foo()` function to the `foo()` function in the common code: `common foo`.

In addition to the improved consistency of behavior across platforms, we also worked hard to fix cases where there was conflicting behavior between IntelliJ IDEA or Android Studio and the compiler. For instance, when you used expected and actual classes, the following would happen:

Common code

```
expect class Identity {
    fun confirmIdentity(): String
}

fun common() {
    // Before 2.0.0, it triggers an IDE-only error
    Identity().confirmIdentity()
    // RESOLUTION_TO_CLASSIFIER : Expected class Identity
has no default constructor.
}
```

Platform code

```
actual class Identity {
    actual fun confirmIdentity() =
"expect class fun: jvm"
}
```

In this example, the expected class `Identity` has no default constructor, so it can't be called successfully in common code. Previously, an error was only reported by the IDE, but the code still compiled successfully on the JVM. However, now the compiler correctly reports an error:

```
Expected class 'expect class Identity : Any' does not have default constructor
```

When resolution behavior doesn't change

We're still in the process of migrating to the new compilation scheme, so the resolution behavior is still the same when you call functions that aren't within the same source set. You'll notice this difference mainly when you use overloads from a multiplatform library in your common code.

Suppose you have a library, which has two `whichFun()` functions with different signatures:

```
// Example library

// MODULE: common
fun whichFun(x: Any) = println("common function")

// MODULE: JVM
fun whichFun(x: Int) = println("platform function")
```

If you call the `whichFun()` function in your common code, the function that has the most relevant argument type in the library will be resolved:

```
// A project that uses the example library for the JVM target

// MODULE: common
fun main(){
    whichFun(2)
    // platform function
}
```

In comparison, if you declare the overloads for `whichFun()` within the same source set, the function from the common code will be resolved because your code doesn't have access to the platform-specific version:

```
// Example library isn't used

// MODULE: common
fun whichFun(x: Any) = println("common function")

fun main(){
    whichFun(2)
    // common function
}

// MODULE: JVM
fun whichFun(x: Int) = println("platform function")
```

Similar to multiplatform libraries, since the `commonTest` module is in a separate source set, it also still has access to platform-specific code. Therefore, the resolution of calls to functions in the `commonTest` module exhibits the same behavior as in the old compilation scheme.

In the future, these remaining cases will be more consistent with the new compilation scheme.

Different visibility levels of expected and actual declarations

Before Kotlin 2.0.0, if you used expected and actual declarations in your Kotlin Multiplatform project, they had to have the same visibility level. Kotlin 2.0.0 now also supports different visibility levels but **only** if the actual declaration is *more* permissive than the expected declaration. For example:

```
expect internal class Attribute // Visibility is internal
actual class Attribute          // Visibility is public by default,
                                // which is more permissive
```

Similarly, if you are using a type alias in your actual declaration, the visibility of the **underlying type** should be the same or more permissive than the expected declaration. For example:

```
expect internal class Attribute                 // Visibility is internal
internal actual typealias Attribute = Expanded

class Expanded                                  // Visibility is public by default,
                                                // which is more permissive
```

How to enable the Kotlin K2 compiler

Starting with Kotlin 2.0.0, the Kotlin K2 compiler is enabled by default.

To upgrade the Kotlin version, change it to 2.0.0 or a later release in your Gradle and Maven build scripts.

Use Kotlin build reports with Gradle

Kotlin build reports provide information about the time spent in different compilation phases for Kotlin compiler tasks, as well as which compiler and Kotlin version were used, and whether the compilation was incremental. These build reports are useful for assessing your build performance. They offer more insight into the Kotlin compilation pipeline than Gradle build scans do because they give you an overview of the performance of all Gradle tasks.

How to enable build reports

To enable build reports, declare where you'd like to save the build report output in your `gradle.properties` file:

```
kotlin.build.report.output=file
```

The following values and their combinations are available for the output:

Option	Description
`file`	Saves build reports in a human-readable format to a local file. By default, it's `${project_folder}/build/reports/kotlin-build/${project_name}-timestamp.txt`
`single_file`	Saves build reports in a format of an object to a specified local file.
`build_scan`	Saves build reports in the `custom values` section of the build scan. Note that the Gradle Enterprise plugin limits the number of custom values and their length. In big projects, some values could be lost.
`http`	Posts build reports using HTTP(S). The POST method sends metrics in JSON format. You can see the current version of the sent data in the Kotlin repository. You can find samples of HTTP endpoints in this blog post

Option	Description
`json`	Saves build reports in JSON format to a local file. Set the location for your build reports in `kotlin.build.report.json.directory`. By default, it's name is `${project_name}-build-<date-time>-<index>.json`.

For more information on what is possible with build reports, see Build reports.

Support in IDEs

Both IntelliJ IDEA and Android Studio fully support the K2 compiler and use it by default to improve code analysis, code completion, and highlighting. You don't need to configure anything. Update to the latest version to see the benefits.

Try the Kotlin K2 compiler in the Kotlin Playground

The Kotlin Playground supports Kotlin 2.0.0 and later releases. Check it out!

How to roll back to the previous compiler

To use the previous compiler in Kotlin 2.0.0–2.3.21, either:

- In your `build.gradle.kts` file, set your language version to `1.9`.

  OR

- Use the following compiler option: `-language-version 1.9`.

From Kotlin 2.4.0 onward, you can't roll back to the previous compiler.

Changes

With the introduction of the new frontend, the Kotlin compiler has undergone several changes. Let's start by highlighting the most significant modifications affecting your code, explaining what has changed and detailing best practices going forward. If you'd like to learn more, we've organized these changes into subject areas to facilitate your further reading.

This section highlights the following modifications:

- Immediate initialization of open properties with backing fields
- Deprecated synthetic setters on a projected receiver
- Forbidden use of inaccessible generic types
- Consistent resolution order of Kotlin properties and Java fields with the same name
- Improved null safety for Java primitive arrays
- Stricter rules for abstract members in expected classes

Immediate initialization of open properties with backing fields

What's changed?

In Kotlin 2.0, all `open` properties with backing fields must be immediately initialized; otherwise, you'll get a compilation error. Previously, only `open var` properties needed to be initialized right away, but now this extends to `open val` properties with backing fields too:

```
open class Base {
    open val a: Int
    open var b: Int

    init {
        // Error starting with Kotlin 2.0 that earlier compiled successfully
        this.a = 1 //Error: open val must have initializer
        // Always an error
        this.b = 1 // Error: open var must have initializer
    }
}

class Derived : Base() {
    override val a: Int = 2
    override var b = 2
}
```

This change makes the compiler's behavior more predictable. Consider an example where an `open val` property is overridden by a `var` property with a custom setter.

If a custom setter is used, deferred initialization can lead to confusion because it's unclear whether you want to initialize the backing field or to invoke the setter. In the past, if you wanted to invoke the setter, the old compiler couldn't guarantee that the setter would then initialize the backing field.

What's the best practice now?

We encourage you to always initialize open properties with backing fields, as we believe this practice is both more efficient and less error-prone.

However, if you don't want to immediately initialize a property, you can:

- Make the property `final`.
- Use a private backing property that allows for deferred initialization.

For more information, see the corresponding issue in YouTrack.

Deprecated synthetics setter on a projected receiver

What's changed?

If you use the synthetic setter of a Java class to assign a type that conflicts with the class's projected type, an error is triggered.

Suppose you have a Java class named `Container` that contains the `getFoo()` and `setFoo()` methods:

```
public class Container<E> {
    public E getFoo() {
        return null;
    }
    public void setFoo(E foo) {}
}
```

If you have the following Kotlin code, where instances of the `Container` class have projected types, using the `setFoo()` method will always generate an error. However, only from Kotlin 2.0.0 will the synthetic `foo` property trigger an error:

```
fun exampleFunction(starProjected: Container<*>, inProjected: Container<in Number>, sampleString:
String) {
    starProjected.setFoo(sampleString)
    // Error since Kotlin 1.0

    // Synthetic setter `foo` is resolved to the `setFoo()` method
    starProjected.foo = sampleString
    // Error since Kotlin 2.0.0

    inProjected.setFoo(sampleString)
    // Error since Kotlin 1.0

    // Synthetic setter `foo` is resolved to the `setFoo()` method
    inProjected.foo = sampleString
    // Error since Kotlin 2.0.0
}
```

What's the best practice now?

If you see that this change introduces errors in your code, you might wish to reconsider how you structure your type declarations. It could be that you don't need to use type projections, or perhaps you need to remove any assignments from your code.

For more information, see the corresponding issue in YouTrack.

Forbidden use of inaccessible generic types

What's changed?

Due to the new architecture of our K2 compiler, we've changed how we handle inaccessible generic types. Generally, you should never rely on inaccessible generic types in your code because this indicates a misconfiguration in your project's build configuration, preventing the compiler from accessing the necessary information to compile. In Kotlin 2.0.0, you can't declare or call a function literal with an inaccessible generic type, nor use a generic type with inaccessible generic type arguments. This restriction helps you avoid compiler errors later in your code.

For example, let's say that you declared a generic class in one module:

```
// Module one
class Node<V>(val value: V)
```

If you have another module (module two) with a dependency configured on module one, your code can access the `Node<V>` class and use it as a type in function types:

```
// Module two
fun execute(func: (Node<Int>) -> Unit) {}
// Function compiles successfully
```

However, if your project is misconfigured such that you have a third module (module three) that depends only on module two, the Kotlin compiler won't be able to access the `Node<V>` class in **module one** when compiling the third module. Now, any lambdas or anonymous functions in module three that use the `Node<V>` type trigger errors in Kotlin 2.0.0, thus preventing avoidable compiler errors, crashes, and run-time exceptions later in your code:

```
// Module three
fun test() {
    // Triggers an error in Kotlin 2.0.0, as the type of the implicit
    // lambda parameter (it) resolves to Node, which is inaccessible
    execute {}

    // Triggers an error in Kotlin 2.0.0, as the type of the unused
    // lambda parameter (_) resolves to Node, which is inaccessible
    execute { _ -> }

    // Triggers an error in Kotlin 2.0.0, as the type of the unused
    // anonymous function parameter (_) resolves to Node, which is inaccessible
    execute(fun (_) {})
}
```

In addition to function literals triggering errors when they contain value parameters of inaccessible generic types, errors also occur when a type has an inaccessible generic type argument.

For example, you have the same generic class declaration in module one. In module two, you declare another generic class: `Container<C>`. In addition, you declare functions in module two that use `Container<C>` with generic class `Node<V>` as a type argument:

Module one

```
// Module one
class Node<V>(val value: V)
```

Module two

```
// Module two
class Container<C>(vararg val content: C)

// Functions with generic class type that
// also have a generic class type argument
fun produce(): Container<Node<Int>> = Container(Node(42))
fun consume(arg: Container<Node<Int>>) {}
```

If you try to call these functions in module three, an error is triggered in Kotlin 2.0.0 because the generic class `Node<V>` is inaccessible from module three:

```
// Module three
fun test() {
    // Triggers an error in Kotlin 2.0.0, as generic class Node<V> is
    // inaccessible
    consume(produce())
}
```

In future releases, we will continue to deprecate the use of inaccessible types in general. We have already started in Kotlin 2.0.0 by adding warnings for some scenarios with inaccessible types, including non-generic ones.

For example, let's use the same module setup as the previous examples, but turn the generic class `Node<V>` into a non-generic class `IntNode`, with all functions declared in module two:

Module one

```
// Module one
class IntNode(val value: Int)
```

Module two

```
// Module two
// A function that contains a lambda
// parameter with `IntNode` type
fun execute(func: (IntNode) -> Unit) {}

class Container<C>(vararg val content: C)

// Functions with generic class type
// that has `IntNode` as a type argument
fun produce(): Container<IntNode> = Container(IntNode(42))
fun consume(arg: Container<IntNode>) {}
```

If you call these functions in module three, some warnings are triggered:

```
// Module three
fun test() {
    // Triggers warnings in Kotlin 2.0.0, as class IntNode is
    // inaccessible.

    execute {}
    // Class 'IntNode' of the parameter 'it' is inaccessible.

    execute { _ -> }
    execute(fun (_) {})
    // Class 'IntNode' of the parameter '_' is inaccessible.

    // Will trigger a warning in future Kotlin releases, as IntNode is
    // inaccessible.
    consume(produce())
}
```

What's the best practice now?

If you encounter new warnings regarding inaccessible generic types, it's highly likely that there's an issue with your build system configuration. We recommend checking your build scripts and configuration.

As a last resort, you can configure a direct dependency for module three on module one. Alternatively, you can modify your code to make the types accessible within the same module.

For more information, see the corresponding issue in YouTrack.

Consistent resolution order of Kotlin properties and Java fields with the same name

What's changed?

Before Kotlin 2.0.0, if you worked with Java and Kotlin classes that inherited from each other and contained Kotlin properties and Java fields with the same name, the resolution behavior of the duplicated name was inconsistent. There was also conflicting behavior between IntelliJ IDEA and the compiler. When developing the new resolution behavior for Kotlin 2.0.0, we aimed to cause the least impact to users.

For example, suppose there is a Java class `Base`:

```java
public class Base {
    public String a = "a";

    public String b = "b";
}
```

Let's say there is also a Kotlin class `Derived` that inherits from the aforementioned `Base` class:

```kotlin
class Derived : Base() {
    val a = "aa"

    // Declares custom get() function
    val b get() = "bb"
}

fun main() {
    // Resolves Derived.a
    println(a)
    // aa

    // Resolves Base.b
    println(b)
    // b
}
```

Prior to Kotlin 2.0.0, `a` resolves to the Kotlin property within the `Derived` Kotlin class, whereas `b` resolves to the Java field in the `Base` Java class.

In Kotlin 2.0.0, the resolution behavior in the example is consistent, ensuring that the Kotlin property supersedes the Java field of the same name. Now, `b` resolves to: `Derived.b`.

Prior to Kotlin 2.0.0, if you used IntelliJ IDEA to go to the declaration or usage of `a`, it would incorrectly navigate to the Java field when it should have navigated to the Kotlin property.

From Kotlin 2.0.0, IntelliJ IDEA correctly navigates to the same location as the compiler.

The general rule is that the subclass takes precedence. The previous example demonstrates this, as the Kotlin property `a` from the `Derived` class is resolved because `Derived` is a subclass of the `Base` Java class.

In the event that the inheritance is reversed and a Java class inherits from a Kotlin class, the Java field in the subclass takes precedence over the Kotlin property with the same name.

Consider this example:

Kotlin

```kotlin
open class Base {
    val a = "aa"
}
```

Java

```java
public class Derived extends Base {
    public String a = "a";
}
```

Now in the following code:

```
fun main() {
    // Resolves Derived.a
    println(a)
    // a
}
```

What's the best practice now?

If this change affects your code, consider whether you really need to use duplicate names. If you want to have Java or Kotlin classes that each contain a field or property with the same name and that each inherit from one another, keep in mind that the field or property in the subclass will take precedence.

For more information, see the corresponding issue in YouTrack.

Improved null safety for Java primitive arrays

What's changed?

Starting with Kotlin 2.0.0, the compiler correctly infers the nullability of Java primitive arrays imported to Kotlin. Now, it retains native nullability from the `TYPE_USE` annotations used with Java primitive arrays and emits errors when their values are not used according to annotations.

Usually, when Java types with `@Nullable` and `@NotNull` annotations are called from Kotlin, they receive the appropriate native nullability:

```
interface DataService {
    @NotNull ResultContainer<@Nullable String> fetchData();
}
```

```
val dataService: DataService = ...
dataService.fetchData() // → ResultContainer<String?>
```

Previously, however, when Java primitive arrays were imported to Kotlin, all `TYPE_USE` annotations were lost, resulting in platform nullability and possibly unsafe code:

```
interface DataProvider {
    int @Nullable [] fetchData();
}
```

```
val dataService: DataProvider = ...
dataService.fetchData() // → IntArray .. IntArray?
// No error, even though `dataService.fetchData()` might be `null` according to annotations
// This might result in a NullPointerException
dataService.fetchData()[0]
```

Note that this issue never affected nullability annotations on the declaration itself, only the `TYPE_USE` ones.

What's the best practice now?

In Kotlin 2.0.0, null safety for Java primitive arrays is now standard in Kotlin, so check your code for new warnings and errors if you use them:

- Any code that uses a `@Nullable` Java primitive array without an explicit nullability check or attempts to pass `null` to a Java method expecting a non-nullable primitive array will now fail to compile.
- Using a `@NotNull` primitive array with a nullability check now emits "Unnecessary safe call" or "Comparison with null always false" warnings.

For more information, see the corresponding issue in YouTrack.

Stricter rules for abstract members in expected classes

Expected and actual classes are in Beta. They are almost stable, but you may need to perform migration steps in the future. We'll do our best to minimize any further changes for you to make.

What's changed?

Due to the separation of common and platform sources during compilation with the K2 compiler, we've implemented stricter rules for abstract members in expected classes.

With the previous compiler, it was possible for an expected non-abstract class to inherit an abstract function without overriding the function. Since the compiler could access both common and platform code at the same time, the compiler could see whether the abstract function had a corresponding override and definition in the actual class.

Now that common and platform sources are compiled separately, the inherited function must be explicitly overridden in the expected class so that the compiler knows the function is not abstract. Otherwise, the compiler reports an `ABSTRACT_MEMBER_NOT_IMPLEMENTED` error.

For example, let's say you have a common source set where you declare an abstract class called `FileSystem` that has an abstract function `listFiles()`. You define the `listFiles()` function in the platform source set as part of an actual declaration.

In your common code, if you have an expected non-abstract class called `PlatformFileSystem` that inherits from the `FileSystem` class, the `PlatformFileSystem` class inherits the abstract function `listFiles()`. However, you can't have an abstract function in a non-abstract class in Kotlin. To make the `listFiles()` function non-abstract, you must declare it as an override without the `abstract` keyword:

Common code

```
abstract class FileSystem {
    abstract fun listFiles()
}
expect open class PlatformFileSystem() :
FileSystem {
    // In Kotlin 2.0.0, an explicit override is
needed
    expect override fun listFiles()
    // Before Kotlin 2.0.0, an override wasn't
needed
}
```

Platform code

```
actual open class PlatformFileSystem :
FileSystem {
    actual override fun listFiles() {}
}
```

What's the best practice now?

If you inherit abstract functions in an expected non-abstract class, add a non-abstract override.

For more information, see the corresponding issue in YouTrack.

Per subject area

These subject areas list changes that are unlikely to affect your code but provide links to the relevant YouTrack issues for further reading. Changes listed with an asterisk (*) next to the Issue ID are explained at the beginning of the section.

Type inference

Issue ID	Title
KT-64189	Incorrect type in compiled function signature of property reference if the type is Normal explicitly
KT-47986	Forbid implicit inferring a type variable into an upper bound in the builder inference context
KT-59275	K2: Require explicit type arguments for generic annotation calls in array literals
KT-53752	Missed subtyping check for an intersection type
KT-59138	Change Java type parameter based types default representation in Kotlin
KT-57178	Change inferred type of prefix increment to return type of getter instead of return type of inc() operator
KT-57609	K2: Stop relying on the presence of @UnsafeVariance using for contravariant parameters
KT-57620	K2: Forbid resolution to subsumed members for raw types
KT-64641	K2: Properly inferred type of callable reference to a callable with extension-function parameter
KT-57011	Make real type of a destructuring variable consistent with explicit type when specified
KT-38895	K2: Fix inconsistent behavior with integer literals overflow
KT-54862	Anonymous type can be exposed from anonymous function from type argument
KT-22379	Condition of while-loop with break can produce unsound smartcast
KT-62507	K2: Prohibit smart cast in common code for expect/actual top-level property
KT-65750	Increment and plus operators that change return type must affect smart casts
KT-65349	[LC] K2: specifying variable types explicitly breaks bound smart casts in some cases that worked in K1

Generics

Issue ID	Title
KT-54309*	Deprecate use of a synthetic setter on a projected receiver
KT-57600	Forbid overriding of Java method with raw-typed parameter with generic typed parameter
KT-54663	Forbid passing possibly nullable type parameter to `in` projected DNN parameter
KT-54066	Deprecate upper bound violation in typealias constructors
KT-49404	Fix type unsoundness for contravariant captured type based on Java class
KT-61718	Forbid unsound code with self upper bounds and captured types
KT-61749	Forbid unsound bound violation in generic inner class of generic outer class
KT-62923	K2: Introduce PROJECTION_IN_IMMEDIATE_ARGUMENT_TO_SUPERTYPE for projections of outer super types of inner class
KT-63243	Report MANY_IMPL_MEMBER_NOT_IMPLEMENTED when inheriting from collection of primitives with an extra specialized implementation from another supertype
KT-60305	K2: Prohibit constructor call and inheritance on type alias that has variance modifiers in expanded type
KT-64965	Fix type hole caused by improper handling of captured types with self-upper bounds
KT-64966	Forbid generic delegating constructor calls with wrong type for generic parameter
KT-65712	Report missing upper bound violation when upper bound is captured type

Resolution

Issue ID	Title
KT-55017*	Choose Kotlin property from derived class during overload resolution with Java field from base class
KT-58260	Make invoke convention works consistently with expected desugaring
KT-62866	K2: Change qualifier resolution behavior when companion object is preferred against static scope
KT-57750	Report ambiguity error when resolving types and having the same-named classes star imported
KT-63558	K2: migrate resolution around COMPATIBILITY_WARNING
KT-51194	False negative CONFLICTING_INHERITED_MEMBERS when dependency class contained in two different versions of the same dependency
KT-37592	Property invoke of a functional type with receiver is preferred over extension function invoke
KT-51666	Qualified this: introduce/prioritize this qualified with type case
KT-54166	Confirm unspecified behavior in case of FQ name conflicts in classpath
KT-64431	K2: forbid using typealiases as qualifier in imports
KT-56520	K1/K2: incorrect work of resolve tower for type references with ambiguity at lower level

Visibility

Issue ID	Title
KT-64474*	Declare usages of inaccessible types as unspecified behavior
KT-55179	False negative PRIVATE_CLASS_MEMBER_FROM_INLINE on calling private class companion object member from internal inline function

Issue ID	Title
KT-58042	Make synthetic property invisible if equivalent getter is invisible even when overridden declaration is visible
KT-64255	Forbid accessing internal setter from a derived class in another module
KT-33917	Prohibit to expose anonymous types from private inline functions
KT-54997	Forbid implicit non-public-API accesses from public-API inline function
KT-56310	Smart casts should not affect visibility of protected members
KT-65494	Forbid access to overlooked private operator functions from public inline function
KT-65004	K1: Setter of var, which overrides protected val, is generates as public
KT-64972	Forbid overriding by private members in link-time for Kotlin/Native

Annotations

Issue ID	Title
KT-58723	Forbid annotating statements with an annotation if it has no EXPRESSION target
KT-49930	Ignore parentheses expression during `REPEATED_ANNOTATION` checking
KT-57422	K2: Prohibit use-site 'get' targeted annotations on property getters
KT-46483	Prohibit annotation on type parameter in where clause
KT-64299	Companion scope is ignored for resolution of annotations on companion object
KT-64654	K2: Introduced ambiguity between user and compiler-required annotations
KT-64527	Annotations on enum values shouldn't be copied to enum value classes

KT-63389	K2: `WRONG_ANNOTATION_TARGET` is reported on incompatible annotations of a type wrapped into `()?`
KT-63388	K2: `WRONG_ANNOTATION_TARGET` is reported on catch parameter type's annotations

Null safety

Issue ID	Title
KT-54521*	Deprecate unsafe usages of array types annotated as Nullable in Java
KT-41034	K2: Change evaluation semantics for combination of safe calls and convention operators
KT-50850	Order of supertypes defines nullability parameters of inherited functions
KT-53982	Keep nullability when approximating local types in public signatures
KT-62998	Forbid assignment of a nullable to a not-null Java field as a selector of unsafe assignment
KT-63209	Report missing errors for error-level nullable arguments of warning-level Java types

Java interoperability

Issue ID	Title
KT-53061	Forbid Java and Kotlin classes with the same FQ name in sources
KT-49882	Classes inherited from Java collections have inconsistent behavior depending on order of supertypes
KT-66324	K2: unspecified behavior in case of Java class inheritance from a Kotlin private class
KT-66220	Passing java vararg method to inline function leads to array of arrays in runtime instead of just an array
KT-66204	Allow to override internal members in K-J-K hierarchy

Properties

Issue ID	Title
KT-57555*	[LC] Forbid deferred initialization of open properties with backing field

Issue ID	Title
KT-58589	Deprecate missed MUST_BE_INITIALIZED when no primary constructor is presented or when class is local
KT-64295	Forbid recursive resolve in case of potential invoke calls on properties
KT-57290	Deprecate smart cast on base class property from invisible derived class if base class is from another module
KT-62661	K2: Missed OPT_IN_USAGE_ERROR for data class properties

Control flow

Issue ID	Title
KT-56408	Inconsistent rules of CFA in class initialization block between K1 and K2
KT-57871	K1/K2 inconsistency on if-conditional without else-branch in parenthesis
KT-42995	False negative "VAL_REASSIGNMENT" in try/catch block with initialization in scope function
KT-65724	Propagate data flow information from try block to catch and finally blocks

Enum classes

Issue ID	Title
KT-57608	Prohibit access to the companion object of enum class during initialization of enum entry
KT-34372	Report missed error for virtual inline method in enum classes
KT-52802	Report ambiguity resolving between property/field and enum entry
KT-47310	Change qualifier resolution behavior when companion property is preferred against enum entry

Functional (SAM) interfaces

Issue ID	Title
KT-52628	Deprecate SAM constructor usages which require OptIn without annotation

Issue ID	Title
KT-57014	Prohibit returning values with incorrect nullability from lambda for SAM constructor of JDK function interfaces
KT-64342	SAM conversion of parameter types of callable references leads to CCE

Companion object

Issue ID	Title
KT-54316	Out-of-call reference to companion object's member has invalid signature
KT-47313	Change (V)::foo reference resolution when V has a companion

Miscellaneous

Issue ID	Title
KT-59739*	K2/MPP reports [ABSTRACT_MEMBER_NOT_IMPLEMENTED] for inheritor in common code when the implementation is located in the actual counterpart
KT-49015	Qualified this: change behavior in case of potential label conflicts
KT-56545	Fix incorrect functions mangling in JVM backend in case of accidental clashing overload in a Java subclass
KT-62019	[LC issue] Prohibit suspend-marked anonymous function declarations in statement positions
KT-55111	OptIn: forbid constructor calls with default arguments (parameters with default values) under marker
KT-61182	Unit conversion is accidentally allowed to be used for expressions on variables + invoke resolution
KT-55199	Forbid promoting callable references with adaptations to KFunction
KT-65776	[LC] K2 breaks `false && ...` and `false || ...`
KT-65682	[LC] Deprecate `header`/`impl` keywords

Issue ID	Title
KT-45375	Generate all Kotlin lambdas via invokedynamic + LambdaMetafactory by default

Compatibility with Kotlin releases

The following Kotlin releases have support for the new K2 compiler:

Kotlin release	Stability level
2.0.0–2.4.20	Stable
1.9.20–1.9.25	Beta
1.9.0–1.9.10	JVM is Beta
1.7.0–1.8.22	Alpha

Compatibility with Kotlin libraries

If you're working with Kotlin/JVM, the K2 compiler works with libraries compiled with any version of Kotlin.

If you're working with Kotlin Multiplatform, the K2 compiler is guaranteed to work with libraries compiled with Kotlin version 1.9.20 and onwards.

Compiler plugins support

Currently, the Kotlin K2 compiler supports the following Kotlin compiler plugins:

- `all-open`
- AtomicFU
- `jvm-abi-gen`
- `js-plain-objects`
- kapt
- Lombok
- `no-arg`
- Parcelize
- Power-assert
- SAM with receiver
- Serialization

In addition, the Kotlin K2 compiler supports:

- The Jetpack Compose 1.5.0 compiler plugin and later versions.
- Kotlin Symbol Processing (KSP) since KSP2.

If you use any additional compiler plugins, check their documentation to see if they are compatible with K2.

Upgrade your custom compiler plugins

Custom compiler plugins use the plugin API, which is Experimental. As a result, the API may change at any time, so we can't guarantee backward compatibility.

The upgrade process has two paths depending on the type of custom plugin you have.

Backend-only compiler plugins

If your plugin implements only `IrGenerationExtension` extension points, the process is the same as for any other new compiler release. Check if there are any changes to the API that you use and make changes if necessary.

Backend and frontend compiler plugins

If your plugin uses frontend-related extension points, you need to rewrite the plugin using the new K2 compiler API. For an introduction to the new API, see FIR Plugin API.

If you have questions about upgrading your custom compiler plugin, join our #compiler Slack channel, and we will do our best to help you.

Share your feedback on the new K2 compiler

We would appreciate any feedback you may have!

- Report any problems you face migrating to the new K2 compiler in our issue tracker.
- Enable the Send usage statistics option to allow JetBrains to collect anonymous data about K2 usage.

Kotlin command-line compiler

Every Kotlin release ships with a standalone version of the compiler. You can download the latest version manually or via a package manager.

Installing the command-line compiler is not an essential step to use Kotlin. The common approach is to write Kotlin applications using IDEs or code editors with official Kotlin support, such as IntelliJ IDEA or Android Studio. They provide full Kotlin support right out of the box.

Learn how to get started with Kotlin in an IDE.

Install the compiler

Manual install

To install the Kotlin compiler manually:

1. Download the latest version (`kotlin-compiler-2.4.20.zip`) from GitHub Releases.
2. Unzip the standalone compiler into a directory and optionally add the `kotlinc/bin` directory to the system path. The `bin` directory contains the scripts needed to compile and run Kotlin on Windows, macOS, and Linux.

If you want to use the Kotlin command-line compiler on Windows, we recommend installing it manually.

SDKMAN!

An easier way to install Kotlin on UNIX-based systems, such as macOS, Linux, Cygwin, FreeBSD, and Solaris, is SDKMAN!. It also works in Bash and ZSH shells. Learn how to install SDKMAN!.

To install the Kotlin compiler via SDKMAN!, run the following command in the terminal:

```
sdk install kotlin
```

Homebrew

Alternatively, on macOS you can install the compiler via Homebrew:

```
brew update
brew install kotlin
```

Snap package

If you use Snap on Ubuntu 16.04 or later, you can install the compiler from the command line:

```
sudo snap install --classic kotlin
```

Create and run an application

1. Create a simple console JVM application in Kotlin that displays `"Hello, World!"`. In a code editor, create a new file called `hello.kt` with the following code:

   ```
   fun main() {
       println("Hello, World!")
   }
   ```

2. Compile the application using the Kotlin compiler:

   ```
   kotlinc hello.kt -include-runtime -d hello.jar
   ```

   - The `-d` option indicates the output path for generated class files, which may be either a directory or a **.jar** file.

- The `-include-runtime` option makes the resulting **.jar** file self-contained and runnable by including the Kotlin runtime library in it.

To see all available options, run:

```
kotlinc -help
```

3. Run the application:

```
java -jar hello.jar
```

To compile a Kotlin/Native application, use the Kotlin/Native compiler.

Compile a library

If you're developing a library to be used by other Kotlin applications, you can build the **.jar** file without including the Kotlin runtime:

```
kotlinc hello.kt -d hello.jar
```

Since binaries compiled this way depend on the Kotlin runtime, you should ensure that it is present in the classpath whenever your compiled library is used.

You can also use the `kotlinr` script to run binaries produced by the Kotlin compiler:

```
kotlinr -classpath hello.jar HelloKt
```

`HelloKt` is the main class name that the Kotlin compiler generates for the file named `hello.kt`.

To compile a Kotlin/Native library, use the Kotlin/Native compiler.

Run the REPL

Run the compiler with the `-Xrepl` compiler option to have an interactive shell. In this shell, you can type any valid Kotlin code and see the results.

Run scripts

You can use Kotlin as a scripting language. A Kotlin script is a Kotlin source file (`.kts`) with top-level executable code.

```
import java.io.File

// Get the passed in path, i.e. "-d some/path" or use the current path.
val path = if (args.contains("-d")) args[1 + args.indexOf("-d")]
           else "."

val folders = File(path).listFiles { file -> file.isDirectory() }
folders?.forEach { folder -> println(folder) }
```

To run a script, pass the `-script` option to the compiler with the corresponding script file:

```
kotlinc -script list_folders.kts -- -d <path_to_folder_to_inspect>
```

Kotlin provides experimental support for script customization, such as adding external properties, providing static or dynamic dependencies, and so on. Customizations are defined by so-called *script definitions* – annotated kotlin classes with the appropriate support code. The script filename extension is used to select the appropriate definition. Learn more about Kotlin custom scripting.

Properly prepared script definitions are detected and applied automatically when the appropriate jars are included in the compilation classpath. Alternatively, you can specify definitions manually by passing the `-script-templates` option to the compiler:

```
kotlinc -script-templates org.example.CustomScriptDefinition -script custom.script1.kts
```

For additional details, see the KEEP-75.

What's next?

Create a console application based on Kotlin/JVM.

Create a console app – tutorial

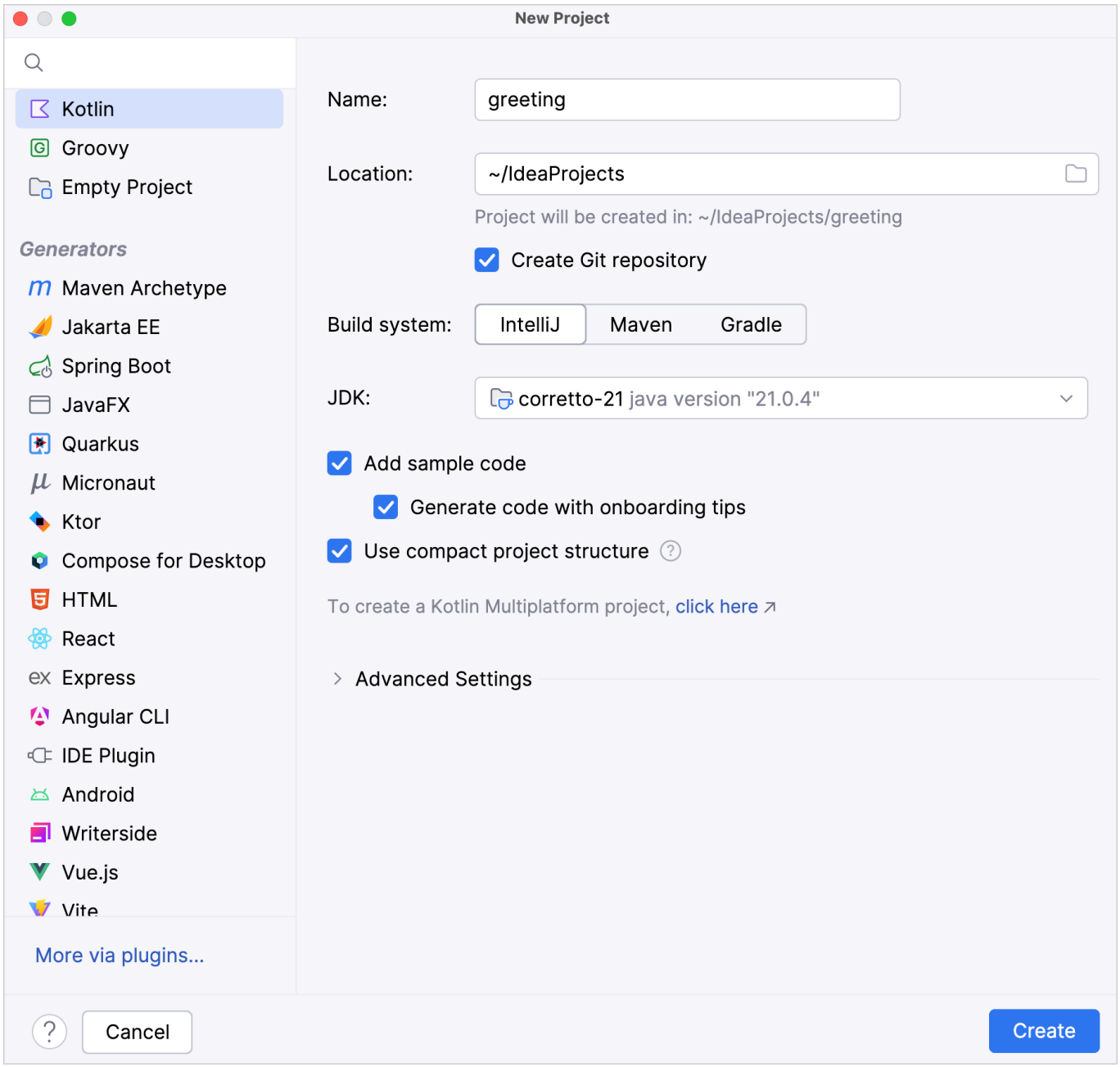
This tutorial demonstrates how to use IntelliJ IDEA for creating a console application.

To get started, first download and install the latest version of IntelliJ IDEA.

Create a project

1. In IntelliJ IDEA, select **File** | **New** | **Project**.
2. In the list on the left, select **Kotlin**.
3. Name the new project and change its location if necessary.

> Select the **Create Git repository** checkbox to place the new project under version control. You will be able to do it later at any time.



Create a console application

4. Select the **IntelliJ** build system. It's a native builder that doesn't require downloading additional artifacts.

   If you want to create a more complex project that needs further configuration, select Maven or Gradle. For Gradle, choose a language for the build script: Kotlin or Groovy.

5. From the **JDK** list, select the JDK that you want to use in your project.

   - If the JDK is installed on your computer, but not defined in the IDE, select **Add JDK** and specify the path to the JDK home directory.
   - If you don't have the necessary JDK on your computer, select **Download JDK**.

6. Enable the **Add sample code** option to create a file with a sample `"Hello World!"` application.

> You can also enable the **Generate code with onboarding tips** option to add some additional useful comments to your sample code.

7. Click **Create**.

If you chose the Gradle build system, you have in your project a build script file: `build.gradle(.kts)`. It includes the `kotlin("jvm")` plugin and dependencies required for your console application. Make sure that you use the latest version of the plugin:

Kotlin

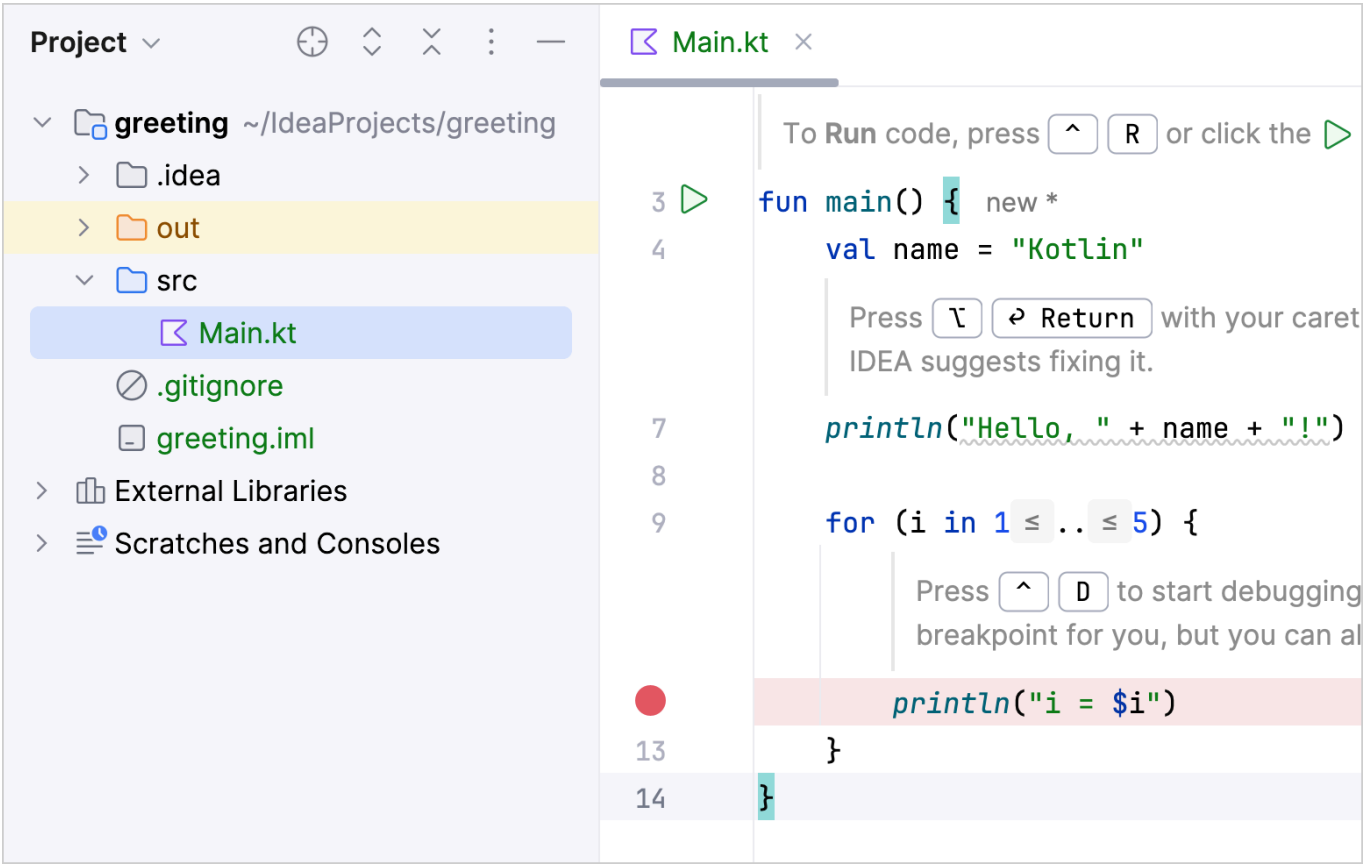
```
plugins {
    kotlin("jvm") version "2.4.20"
    application
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '2.4.20'
    id 'application'
}
```

Create an application

1. Open the `Main.kt` file in `src/main/kotlin`.
   The `src` directory contains Kotlin source files and resources. The `Main.kt` file contains sample code that will print `Hello, Kotlin!` as well as several lines with values of the cycle iterator.



Main.kt with main fun

2. Modify the code so that it requests your name and says `Hello` to you:

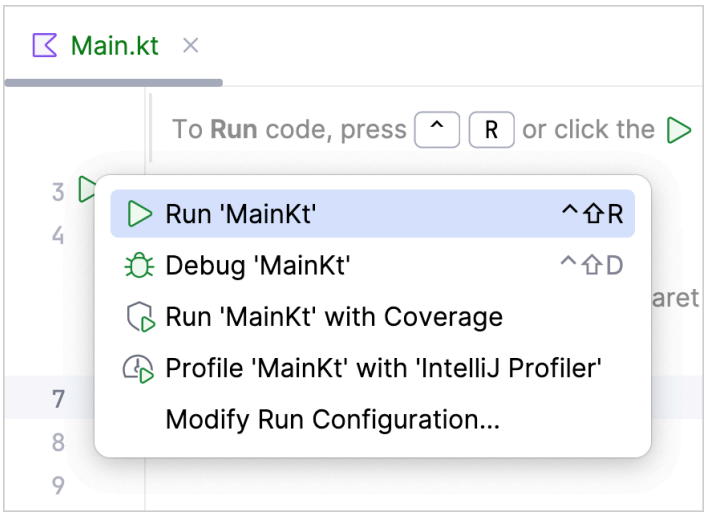
   - Create an input prompt and assign to the `name` variable the value returned by the `readln()` function.
   - Let's use a string template instead of concatenation by adding a dollar sign `$` before the variable name directly in the text output like this – `$name`.

```
fun main() {
    println("What's your name?")
    val name = readln()
    println("Hello, $name!")

    // ...
}
```

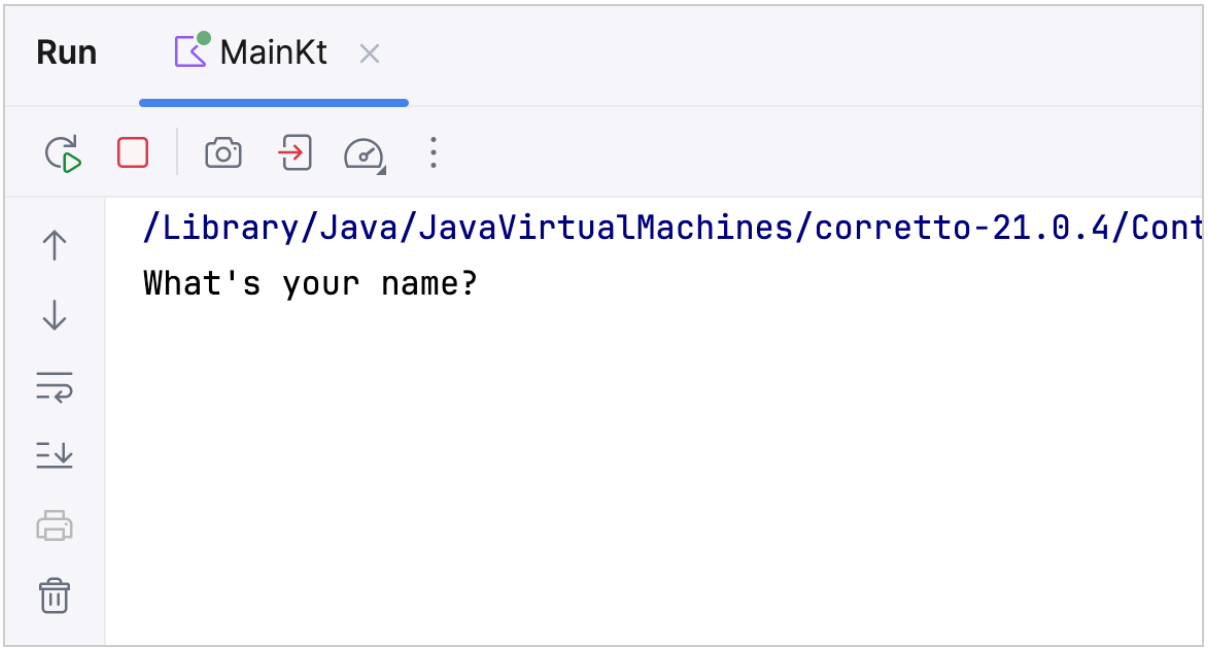
Run the application

Now the application is ready to run. The easiest way to do this is to click the green **Run** icon in the gutter and select **Run 'MainKt'**.



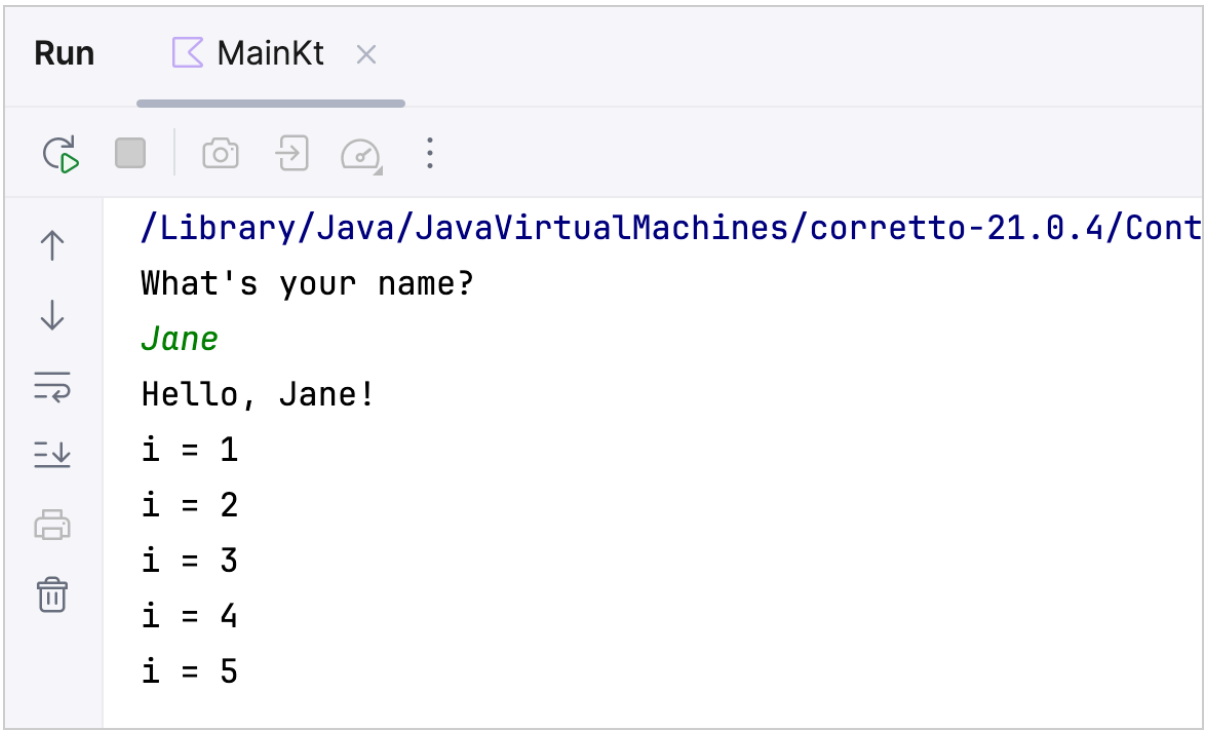
Running a console app

You can see the result in the **Run** tool window.



Kotlin run output

Enter your name and accept the greetings from your application!



Kotlin run output

Congratulations! You have just run your first Kotlin application.

What's next?

Once you've created this application, you can start to dive deeper into Kotlin syntax:

- Take the Kotlin tour
- Install the JetBrains Academy plugin for IDEA and complete exercises from the Kotlin Koans course

Kotlin compiler options

Each release of Kotlin includes compilers for the supported targets: JVM, JavaScript, and native binaries for supported platforms.

These compilers are used by:

- The IDE, when you click the **Compile** or **Run** button for your Kotlin project.
- Gradle, when you call `gradle build` in a console or in the IDE.
- Maven, when you call `mvn compile` or `mvn test-compile` in a console or in the IDE.

You can also run Kotlin compilers manually from the command line as described in the Working with command-line compiler tutorial.

Compiler options

Kotlin compilers have a number of options for tailoring the compiling process. Compiler options for different targets are listed on this page together with a description of each one.

There are several ways to set the compiler options and their values (*compiler arguments*):

- In IntelliJ IDEA, write in the compiler arguments in the **Additional command line parameters** text box in **Settings/Preferences** | **Build, Execution, Deployment** | **Compiler** | **Kotlin Compiler**.
- If you're using Gradle, specify the compiler arguments in the `compilerOptions` property of the Kotlin compilation task. For details, see Gradle compiler options.
- If you're using Maven, specify the compiler arguments in the `<configuration>` element of the Maven plugin node. For details, see Maven.
- If you run a command-line compiler, add the compiler arguments directly to the utility call or write them into an argfile.

  For example:

  ```
  $ kotlinc hello.kt -include-runtime -d hello.jar
  ```

  On Windows, when you pass compiler arguments that contain delimiter characters (whitespace, `=`, `;`, `,`), surround these arguments with double quotes (`"`).

  ```
  $ kotlinc.bat hello.kt -include-runtime -d "My Folder\hello.jar"
  ```

Schema for compiler options

A common schema for all compiler options is published under `org.jetbrains.kotlin:kotlin-compiler-arguments-description` as a JAR artifact. This artifact includes both a code representation and a JSON equivalent of all compiler option descriptions (for non-Kotlin consumers). As well as metadata, such as the version in which each option was introduced or stabilized.

Common options

The following options are common for all Kotlin compilers.

-api-version *version*

Set the API version to control which Kotlin APIs your code can use at runtime. For example, if you use Kotlin compiler version 2.4.0 with `-api-version=2.1`, your code remains compatible with Kotlin standard library 2.1.0.

You can't set the `-api-version` value higher than the `-language-version` value.

In most cases, the API version and language version should be the same. One exception is when you develop a library for consumers who must run an older version of the Kotlin standard library. In that case, set an older API version to avoid accidentally using APIs that aren't available to those consumers.

For more information about how the API version affects compatibility, see Backward compatibility guidelines for library authors.

-help (-h)

Display usage information and exit. Only standard options are shown. To show advanced options, use `-X`.

-kotlin-home *path*

Specify a custom path to the Kotlin compiler used for the discovery of runtime libraries.

-language-version *version*

Set the language version to control which Kotlin language features are available during compilation.

For example, if you want to benefit from new compilation performance improvements without changing compiler behavior, you can use a new compiler version with an older language version. With an older language version, you can't use newer language features, but you also don't see new errors and deprecations introduced after that version. This approach is especially useful for library authors who need to preserve compatibility with older Kotlin versions. For more information, see Backward compatibility guidelines for library authors.

You can configure one of the latest three stable versions of Kotlin as the language version. For example, Kotlin 2.5.0 supports language versions as old as 2.2.

If you use an older language version, you also need to use an older API version. For more information, see -api-version version .

> Technically, you can configure a newer language version to try upcoming language features before they are stabilized. However, it's best to enable individual features by following their dedicated instructions.

-opt-in *annotation*

Enable usages of API that requires opt-in with a requirement annotation with the given fully qualified name.

-P plugin:pluginId:optionName=value

Pass an option to a Kotlin compiler plugin. Core plugins and their options are listed in the Core compiler plugins section of the documentation.

-progressive

Enable the progressive mode for the compiler.

In the progressive mode, deprecations and bug fixes for unstable code take effect immediately, instead of going through a graceful migration cycle. Code written in the progressive mode is backwards compatible; however, code written in a non-progressive mode may cause compilation errors in the progressive mode.

-script

Evaluate a Kotlin script file. When called with this option, the compiler executes the first Kotlin script (`*.kts`) file among the given arguments.

-verbose

Enable verbose logging output which includes details of the compilation process.

-version

Display the compiler version.

Experimental

-X

Display information about the advanced options and exit. These options are currently unstable: their names and behavior may be changed without notice.

Experimental

Kotlin contract options

The following options enable experimental Kotlin contract features.

-Xallow-contracts-on-more-functions

Enables contracts in additional declarations, including property accessors, specific operator functions, and type assertions on generic types.

-Xallow-condition-implies-returns-contracts

Allows using the `returnsNotNull()` function in contracts to assume a non-null return value for specified conditions.

-Xallow-holdsin-contract

Allows using the `holdsIn` keyword in contracts to assume that a boolean condition is `true` inside a lambda.

-Xallow-returns-result-of

Allows using the `returnsResultOf()` contract so that the unused return value checker can distinguish between results that can be ignored and meaningful results from higher-order functions.

Experimental

-Xallow-reified-type-in-catch

Enables support for reified `Throwable` type parameters in `catch` clauses of `inline` functions.

Experimental

-Xcollection-literals

Enables support for collection literals with brackets syntax `[]`.

Experimental

-Xcompiler-plugin-order={plugin.before>plugin.after}

Configure the running order of compiler plugins. The compiler runs `plugin.before` first, and then `plugin.after`:

You can define multiple ordering rules for three or more plugins. For example:

```
kotlinc -Xcompiler-plugin-order=plugin.first>plugin.middle
kotlinc -Xcompiler-plugin-order=plugin.middle>plugin.last
```

This results in the following running order:

1. `plugin.first`
2. `plugin.middle`
3. `plugin.last`

If a compiler plugin isn't present, the corresponding rule is ignored.

You can configure the following plugins by their IDs:

Compiler plugin	Plugin ID
`all-open`, `kotlin-spring`	`org.jetbrains.kotlin.allopen`
AtomicFU	`org.jetbrains.kotlinx.atomicfu`
Compose	`androidx.compose.compiler.plugins.kotlin`
`js-plain-objects`	`org.jetbrains.kotlinx.jspo`
`jvm-abi-gen`	`org.jetbrains.kotlin.jvm.abi`
kapt	`org.jetbrains.kotlin.kapt3`
Lombok	`org.jetbrains.kotlin.lombok`
`no-arg`, `kotlin-jpa`	`org.jetbrains.kotlin.noarg`
Parcelize	`org.jetbrains.kotlin.parcelize`
Power-assert	`org.jetbrains.kotlin.powerassert`
SAM with receiver	`org.jetbrains.kotlin.samWithReceiver`
Serialization	`org.jetbrains.kotlinx.serialization`

This running order controls only the backend of compiler plugins and not the frontend.

Experimental

-Xdata-flow-based-exhaustiveness

Enables data-flow–based exhaustiveness checks for `when` expressions.

Experimental

-Xexplicit-context-arguments

Enables explicit context arguments for context parameters.

This lets you resolve overload ambiguity by passing context arguments at the call site.

Experimental

-Xklib-ir-inliner

Configure whether intra-module inlining is enabled for Kotlin/Native, Kotlin/JS, and Kotlin/Wasm. By default, it's enabled.

The option supports the following modes:

- `disabled`: disables intra-module inlining for Kotlin/Native, Kotlin/JS, and Kotlin/Wasm.
- `full`: enables cross-module inlining.

Experimental

-Xintrinsic-const-evaluation

Enables improved compile-time constants.

Experimental

-Xname-based-destructuring

Configure how the compiler interprets destructuring declarations based on property names.

The option supports the following modes:

- `only-syntax`: enables the explicit form of name-based destructuring without changing the behavior of existing destructuring declarations.
- `name-mismatch`: reports warnings when position-based destructuring in data classes uses variable names that don't match the property names.
- `complete`: enables short-form name-based destructuring with parentheses and continues supporting position-based destructuring with square bracket syntax.

Experimental

-Xphases-to-dump-before

Set to `ExternalPackageParentPatcherLowering` to create a dump file after the IR lowering compilation stage. Configure the output directory for the Kotlin/JVM with the `-Xdump-directory` compiler option.

Experimental

-Xrepl

Activates the Kotlin REPL.

```
kotlinc -Xrepl
```

Experimental

-Xreturn-value-checker

Configure how the compiler reports ignored results:

- `disable` : disables the unused return value checker (default).
- `check` : enables the checker and reports warnings for ignored results from marked functions.
- `full` : enables the checker, treats all functions in your project as marked, and reports warnings for ignored results.

Warning management

-nowarn

Suppress all warnings during compilation.

-Werror

Treat all warnings as compilation errors.

-Wextra

Enable additional declaration, expression, and type compiler checks that emit warnings if true.

Experimental

-Xrender-internal-diagnostic-names

Prints internal diagnostic names alongside warnings. This is useful for identifying the `DIAGNOSTIC_NAME` configured for the `-Xwarning-level` option.

Experimental

-Xwarning-level

Configure the severity level of specific compiler warnings:

```
kotlinc -Xwarning-level=DIAGNOSTIC_NAME:(error|warning|disabled)
```

- `error` : raises only the specified warning to an error.
- `warning` : emits a warning for the specified diagnostic and is enabled by default.
- `disabled` : suppresses only the specified warning module-wide.

You can adjust warning reporting in your project by combining module-wide rules with specific ones:

Command	Description
`-nowarn -Xwarning-level=DIAGNOSTIC_NAME:warning`	Suppress all warnings except for the specified ones.

Command	Description
`-Werror -Xwarning-level=DIAGNOSTIC_NAME:warning`	Raise all warnings to errors except for the specified ones.
`-Wextra -Xwarning-level=DIAGNOSTIC_NAME:disabled`	Enable all additional checks except for the specified ones.

If you have many warnings to exclude from the general rules, you can list them in a separate file using `@argfile`.

You can use `-Xrender-internal-diagnostic-names` to discover the `DIAGNOSTIC_NAME`.

@argfile

Read the compiler options from the given file. Such a file can contain compiler options with values and paths to the source files. Options and paths should be separated by whitespaces. For example:

```
-include-runtime -d hello.jar hello.kt
```

To pass values that contain whitespaces, surround them with single (') or double (") quotes. If a value contains quotation marks in it, escape them with a backslash (\).

```
-include-runtime -d 'My folder'
```

You can also pass multiple argument files, for example, to separate compiler options from source files.

```
$ kotlinc @compiler.options @classes
```

If the files reside in locations different from the current directory, use relative paths.

```
$ kotlinc @options/compiler.options hello.kt
```

Kotlin/JVM compiler options

The Kotlin compiler for JVM compiles Kotlin source files into Java class files. The command-line tools for Kotlin to JVM compilation are `kotlinc` and `kotlinc-jvm`. You can also use them for executing Kotlin script files.

In addition to the common options, Kotlin/JVM compiler has the options listed below.

-classpath *path* (-cp *path*)

Search for class files in the specified paths. Separate elements of the classpath with system path separators (**;** on Windows, **:** on macOS/Linux). The classpath can contain file and directory paths, ZIP, or JAR files.

-d *path*

Place the generated class files into the specified location. The location can be a directory, a ZIP, or a JAR file.

-include-runtime

Include the Kotlin runtime into the resulting JAR file. Makes the resulting archive runnable on any Java-enabled environment.

-jdk-home *path*

Use a custom JDK home directory to include into the classpath if it differs from the default `JAVA_HOME`.

Experimental

-Xjdk-release=version

Specify the target version of the generated JVM bytecode. Limit the API of the JDK in the classpath to the specified Java version. Automatically sets `-jvm-target version`. Possible values are `1.8`, `9`, `10`, ..., `26`.

> This option is not guaranteed to be effective for each JDK distribution.

-jvm-default *mode*

Control how functions declared in interfaces are compiled to default methods on the JVM.

Mode	Description
`enable`	Generates default implementations in interfaces and includes bridge functions in subclasses and `DefaultImpls` classes. (Default)
`no-compatibility`	Generates only default implementations in interfaces, skipping compatibility bridges and `DefaultImpls` classes.
`disable`	Generates only compatibility bridges and `DefaultImpls` classes, skipping default methods.

-jvm-target *version*

Specify the target version of the generated JVM bytecode. Possible values are `1.8`, `9`, `10`, ..., `26`. The default value is `1.8`.

-java-parameters

Generate metadata for Java 1.8 reflection on method parameters.

-module-name *name* (JVM)

Set a custom name for the generated `.kotlin_module` file.

-no-jdk

Don't automatically include the Java runtime into the classpath.

-no-reflect

Don't automatically include the Kotlin reflection (`kotlin-reflect.jar`) into the classpath.

-no-stdlib (JVM)

Don't automatically include the Kotlin/JVM stdlib (`kotlin-stdlib.jar`) and Kotlin reflection (`kotlin-reflect.jar`) into the classpath.

-script-templates *classnames[,]*

Script definition template classes. Use fully qualified class names and separate them with commas (,).

Experimental

-Xdump-directory

Configure the dump file directory for the -Xphases-to-dump-before` compiler option.

Experimental

-Xjvm-expose-boxed

Generate boxed versions of all inline value classes in the module, along with boxed variants of functions that use them, making both accessible from Java. For more information, see Inline value classes in the guide to calling Kotlin from Java.

Experimental

-Xnullability-annotations

Configure how the Kotlin compiler interprets nullability annotations from specific Java packages.

For the full list of supported annotations and configuration options, see Nullability annotations.

Kotlin/JS compiler options

The Kotlin compiler for JS compiles Kotlin source files into JavaScript code. The command-line tool for Kotlin to JS compilation is `kotlinc-js` .

In addition to the common options, Kotlin/JS compiler has the options listed below.

-libraries *path*

Paths to Kotlin libraries with `.meta.js` and `.kjsm` files, separated by the system path separator.

-main *{call|noCall}*

Define whether the `main` function should be called upon execution.

-meta-info

Generate `.meta.js` and `.kjsm` files with metadata. Use this option when creating a JS library.

-module-kind {umd|commonjs|amd|plain}

The kind of JS module generated by the compiler:

- `umd` - a Universal Module Definition module
- `commonjs` - a CommonJS module
- `amd` - an Asynchronous Module Definition module
- `plain` - a plain JS module

To learn more about the different kinds of JS modules and the distinctions between them, see this article.

-no-stdlib (JS)

Don't automatically include the default Kotlin/JS stdlib into the compilation dependencies.

-output *filepath*

Set the destination file for the compilation result. The value must be a path to a `.js` file including its name.

-output-postfix *filepath*

Add the content of the specified file to the end of the output file.

-output-prefix *filepath*

Add the content of the specified file to the beginning of the output file.

-source-map

Generate the source map.

-source-map-base-dirs *path*

Use the specified paths as base directories. Base directories are used for calculating relative paths in the source map.

-source-map-embed-sources *{always|never|inlining}*

Embed source files into the source map.

-source-map-names-policy *{simple-names|fully-qualified-names|no}*

Add variable and function names that you declared in Kotlin code into the source map.

Setting	Description	Example output
`simple-names`	Variable names and simple function names are added. (Default)	`main`

Setting	Description	Example output
`fully-qualified-names`	Variable names and fully qualified function names are added.	`com.example.kjs.playground.main`
`no`	No variable or function names are added.	N/A

-source-map-prefix

Add the specified prefix to paths in the source map.

-target {es5|es2015}

Generate JS files for the specified ECMA version.

Experimental

-Xenable-implementing-interfaces-from-typescript

Allow implementing Kotlin interfaces exported with the `@JsExport` annotation from JavaScript/TypeScript.

Experimental

-Xes-long-as-bigint

Enable support for the JavaScript `BigInt` type to represent Kotlin `Long` values when compiling to modern JavaScript (ES2020).

Experimental

-Xsuspend-lambda-exporting

Allow exporting suspending lambda expressions declared in `@JsExport` declarations as JavaScript `async` functions.

Kotlin/Native compiler options

Kotlin/Native compiler compiles Kotlin source files into native binaries for the supported platforms. The command-line tool for Kotlin/Native compilation is `kotlinc-native`.

In addition to the common options, Kotlin/Native compiler has the options listed below.

-enable-assertions (-ea)

Enable runtime assertions in the generated code.

-entry *name* (-e *name*)

Specify the qualified entry point name.

-g

Enable emitting debug information. This option lowers the optimization level and should not be combined with the `-opt` option.

-generate-test-runner (-tr)

Produce an application for running unit tests from the project.

-generate-no-exit-test-runner (-trn)

Produce an application for running unit tests without an explicit process exit.

-include-binary *path* (-ib *path*)

Pack external binary within the generated klib file.

-library *path* (-l *path*)

Link with the library. To learn about using libraries in Kotlin/native projects, see Kotlin/Native libraries.

-library-version *version* (-lv *version*)

Set the library version.

-linker-option

Pass an argument to the linker during binary building. This can be used for linking against some native library.

-linker-options *args*

Pass multiple arguments to the linker during binary building. Separate arguments with whitespaces.

-list-targets

List the available hardware targets.

-manifest *path*

Provide a manifest addend file.

-module-name *name* (Native)

Specify a name for the compilation module. This option can also be used to specify a name prefix for the declarations exported to Objective-C: How do I specify a custom Objective-C prefix/name for my Kotlin framework?

-native-library *path* (-nl *path*)

Include the native bitcode library.

-no-default-libs

Disable linking user code with the prebuilt platform libraries distributed with the compiler.

-nomain

Assume the `main` entry point to be provided by external libraries.

-nopack

Don't pack the library into a klib file.

-nostdlib

Don't link with stdlib.

-opt

Enable compilation optimizations and produce a binary with better runtime performance. It's not recommended to combine it with the `-g` option, which lowers the optimization level.

-output *name* (-o *name*)

Set the name for the output file.

-produce *output* (-p *output*)

Specify output file kind:

- `program`
- `static`
- `dynamic`
- `framework`
- `library`
- `bitcode`

-repo *path* (-r *path*)

Library search path. For more information, see Library search sequence.

-target *target*

Set the hardware target. To see the list of available targets, use the `-list-targets` option.

Experimental

-Xccall-mode

Enable the new interoperability mode for C or Objective-C libraries imported via cinterop.

Experimental

-Xoverride-konan-properties=min.version.*

Configure lower supported versions of Apple targets than the Kotlin defaults. For example:

```
kotlinc -Xoverride-konan-properties=minVersion.ios=14.0
kotlinc -Xoverride-konan-properties=minVersion.macos=11.0
kotlinc -Xoverride-konan-properties=minVersion.tvos=14.0
kotlinc -Xoverride-konan-properties=minVersion.watchos=7.0
```

Compiler execution strategy

The *Kotlin compiler execution strategy* defines where the Kotlin compiler runs. Build tools such as Gradle or Maven configure the strategy.

There are two compiler execution strategies:

Strategy	Where the Kotlin compiler runs	Other characteristics and notes
Kotlin daemon	Inside its own daemon process	*The default and fastest strategy* in Gradle and Maven. The daemon process can be shared between different build system processes and multiple parallel compilations.
In process	Inside the build tool's process	The simplest strategy from the perspective of memory management, but it's less isolated from other logic running in the same process because it shares state, such as JVM system properties.

Configure in Gradle

You can define the Kotlin compiler execution strategy using one of the following properties:

- The `kotlin.compiler.execution.strategy` Gradle property.
- The `compilerExecutionStrategy` compile task property.

Use the Gradle property

The possible values for the `kotlin.compiler.execution.strategy` property are:

- `daemon` (default)
- `in-process`

Set the `kotlin.compiler.execution.strategy` property in `gradle.properties`:

```
kotlin.compiler.execution.strategy=in-process
```

Use the compile task property

The `compilerExecutionStrategy` task property takes priority over the `kotlin.compiler.execution.strategy` Gradle property.

The possible values for the `compilerExecutionStrategy` task property are:

- `DAEMON` (default)
- `IN_PROCESS`

Set the `compilerExecutionStrategy` task property in your build scripts:

Kotlin

```
import org.jetbrains.kotlin.gradle.tasks.CompileUsingKotlinDaemon
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy

// ...

tasks.withType<CompileUsingKotlinDaemon>().configureEach {
    compilerExecutionStrategy.set(KotlinCompilerExecutionStrategy.IN_PROCESS)
}
```

Groovy

```
import org.jetbrains.kotlin.gradle.tasks.CompileUsingKotlinDaemon
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy

// ...

tasks.withType(CompileUsingKotlinDaemon)
        .configureEach {
            compilerExecutionStrategy = KotlinCompilerExecutionStrategy.IN_PROCESS
        }
```

Fallback strategy

If communication with the Kotlin daemon fails, the compiler falls back to the "In process" strategy.

When this fallback happens, Gradle prints the following warning in the build output:

```
Failed to compile with Kotlin daemon: java.lang.RuntimeException: Could not connect to Kotlin compile
daemon
[exception stacktrace]
Using fallback strategy: Compile without Kotlin daemon
Try ./gradlew --stop if this issue persists.
```

A silent fallback can consume a lot of system resources or lead to non-deterministic builds. For more information, see this YouTrack issue.

To prevent fallback, use the `kotlin.daemon.useFallbackStrategy` Gradle property. The default value is `true`. When set to `false`, builds fail if there are problems with the daemon's startup or communication. Declare this property in `gradle.properties`:

```
kotlin.daemon.useFallbackStrategy=false
```

There is also a `useDaemonFallbackStrategy` property in Kotlin compile tasks. If you use both properties, the `useDaemonFallbackStrategy` property takes priority.

Kotlin

```
tasks {
    compileKotlin {
        useDaemonFallbackStrategy.set(false)
    }
}
```

Groovy

```
tasks.named("compileKotlin").configure {
    useDaemonFallbackStrategy = false
}
```

If there is not enough memory to run the compilation, the logs show a related message.

Configure in Maven

By default, Maven uses the Kotlin daemon compiler execution strategy. To switch to the "in process" strategy, set the following property in your `pom.xml` file:

```
<properties>
    <kotlin.compiler.daemon>false</kotlin.compiler.daemon>
</properties>
```

Compiler plugins

Compiler plugins hook into the compilation process to analyze or change code while it's being compiled, without modifying the compiler itself. For example, they can annotate code or generate new code to make it compatible with other frameworks or APIs.

This page explains the Kotlin compiler plugins available to you and what you can do if none of them suits your use case.

The Kotlin team maintains the following compiler plugins:

Plugin	Description
All-open	Automatically makes annotated classes and their members `open` so frameworks can extend them at runtime.
AtomicFU	Transforms atomic operations into efficient, platform-specific implementations for lock-free concurrency.
DataFrame	Generates typed APIs that let you work with a `DataFrame` in a safe, Kotlin-friendly way.

Plugin	Description
`jvm-abi-gen`	Generates Application Binary Interface (ABI) JARs.
`js-plain-objects`	Exposes Kotlin classes as plain JavaScript objects to improve interop with JS tooling and libraries.
kapt	Runs Java annotation processors on Kotlin code and generates additional source files.
Lombok	Enables Kotlin code to understand and use code generated by Lombok annotations in Java sources.
`no-arg`	Generates no-argument constructors for annotated classes to support frameworks that require them.
Power-assert	Enhances assertion failures by showing detailed values for each part of an expression.
SAM with receiver	Allows SAM interfaces to use lambdas with receivers for a more DSL-like syntax.
Serialization	Generates code that serializes and deserializes Kotlin objects without reflection.

The Android team at Google maintains:

Plugin	Description
Compose compiler Gradle plugin	Integrates the Compose compiler with Gradle to enable declarative UI features and Compose-specific optimizations.
Parcelize plugin	Generates `Parcelable` implementations automatically so you can pass Kotlin objects between Android components.

If you need to adapt the compilation process in a way that these plugins don't cover, first check whether you can use the Kotlin Symbol Processing (KSP) API or an external linter such as Android lint. You can browse our Kotlin Slack or reach out and ask for advice about your use case.

If you *still* can't find what you need, you can create a custom compiler plugin. Use this approach only as a last resort, because the Kotlin compiler plugin API is **unstable**. If you create a custom compiler plugin, you need to invest significant ongoing effort to maintain it, since each new compiler release introduces breaking changes.

All-open compiler plugin

Kotlin has classes and their members `final` by default, which makes it inconvenient to use frameworks and libraries such as Spring AOP that require classes to be `open`. The `all-open` compiler plugin adapts Kotlin to the requirements of those frameworks and makes classes annotated with a specific annotation and their members open without the explicit `open` keyword.

For instance, when you use Spring, you don't need all the classes to be open, but only classes annotated with specific annotations like `@Configuration` or `@Service`. The `all-open` plugin allows you to specify such annotations.

Kotlin provides `all-open` plugin support both for Gradle and Maven with the complete IDE integration.

For Spring, you can use the `kotlin-spring` compiler plugin.

Gradle

Add the plugin in your `build.gradle(.kts)` file:

Kotlin

```
plugins {
    kotlin("plugin.allopen") version "2.4.20"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.plugin.allopen" version "2.4.20"
}
```

Then specify the list of annotations that will make classes open:

Kotlin

```
allOpen {
    annotation("com.my.Annotation")
    // annotations("com.another.Annotation", "com.third.Annotation")
}
```

Groovy

```
allOpen {
    annotation("com.my.Annotation")
    // annotations("com.another.Annotation", "com.third.Annotation")
}
```

If the class (or any of its superclasses) is annotated with `com.my.Annotation`, the class itself and all its members will become open.

It also works with meta-annotations:

```
@com.my.Annotation
annotation class MyFrameworkAnnotation

@MyFrameworkAnnotation
class MyClass // will be all-open
```

`MyFrameworkAnnotation` is annotated with the all-open meta-annotation `com.my.Annotation`, so it becomes an all-open annotation as well.

Maven

Add the plugin in your `pom.xml` file:

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- Or "spring" for the Spring support -->
            <plugin>all-open</plugin>
        </compilerPlugins>

        <pluginOptions>
            <!-- Each annotation is placed on its own line -->
            <option>all-open:annotation=com.my.Annotation</option>
            <option>all-open:annotation=com.their.AnotherAnnotation</option>
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-allopen</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

Please refer to the Gradle section for the detailed information about how all-open annotations work.

Spring support

If you use Spring, you can enable the `kotlin-spring` compiler plugin instead of specifying Spring annotations manually. The `kotlin-spring` is a wrapper on top of `all-open`, and it behaves exactly the same way.

Add the `spring` plugin in your `build.gradle(.kts)` file:

Kotlin

```
plugins {
    id("org.jetbrains.kotlin.plugin.spring") version "2.4.20"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.plugin.spring" version "2.4.20"
}
```

In Maven, the `spring` plugin is provided by the `kotlin-maven-allopen` plugin dependency, so to enable it in your `pom.xml` file:

```
<compilerPlugins>
    <plugin>spring</plugin>
</compilerPlugins>

<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-maven-allopen</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

The plugin specifies the following annotations:

- `@Component`
- `@Async`
- `@Transactional`
- `@Cacheable`
- `@SpringBootTest`

Thanks to meta-annotations support, classes annotated with `@Configuration`, `@Controller`, `@RestController`, `@Service` or `@Repository` are automatically opened since these annotations are meta-annotated with `@Component`.

Of course, you can use both `kotlin-allopen` and `kotlin-spring` in the same project.

If you generate the project template by the start.spring.io service, the `kotlin-spring` plugin will be enabled by default.

Command-line compiler

All-open compiler plugin JAR is available in the binary distribution of the Kotlin compiler. You can attach the plugin by providing the path to its JAR file using the `-Xplugin` kotlinc option:

```
-Xplugin=$KOTLIN_HOME/lib/allopen-compiler-plugin.jar
```

You can specify all-open annotations directly, using the `annotation` plugin option, or enable the *preset*:

```
# The plugin option format is: "-P plugin:<plugin id>:<key>=<value>".
# Options can be repeated.

-P plugin:org.jetbrains.kotlin.allopen:annotation=com.my.Annotation
-P plugin:org.jetbrains.kotlin.allopen:preset=spring
```

Presets that available for the `all-open` plugin are: `spring`, `micronaut`, and `quarkus`.

kapt compiler plugin

The kapt compiler plugin allows you to use existing Java annotation processors in Kotlin and works with both Maven and Gradle. It generates stub files from Kotlin source code and then runs the Java annotation processors on those stubs.

This enables Java-based annotation processing in your Kotlin projects for libraries like MapStruct and Data Binding.

> kapt is not supported in the IntelliJ build system. To re-run annotation processing in IntelliJ IDEA, launch the build from the **Maven** tool window.

Set up the plugin

You can configure the kapt plugin for Gradle, Maven, or use it from the command line.

Gradle

To use kapt in Gradle, follow these steps:

1. Apply the `kapt` Gradle plugin in your build script file `build.gradle(.kts)`:

Kotlin

```
plugins {
    kotlin("kapt") version "2.4.20"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.kapt" version "2.4.20"
}
```

2. Add the respective dependencies using the `kapt` configuration in the `dependencies {}` block:

Kotlin

```
dependencies {
    kapt("groupId:artifactId:version")
}
```

Groovy

```
dependencies {
    kapt 'groupId:artifactId:version'
}
```

3. If you previously used the Android support for annotation processors, replace usages of the `annotationProcessor` configuration with `kapt`. If your project contains Java classes, the kapt plugin processes them as well.

   If you use annotation processors for your `androidTest` or `test` sources, the respective `kapt` configurations are named `kaptAndroidTest` and `kaptTest`. Note that `kaptAndroidTest` and `kaptTest` extend `kapt`, so you can provide the `kapt` dependency, and it will be available both for production sources and tests.

Maven

You can set up kapt using either the `<extensions>` option to simplify setup or manually to get full control over kapt's execution.

Automatic configuration

You can simplify kapt configuration by enabling the `<extensions>` option for the Kotlin Maven plugin. In this case, you don't need to manually set up kapt's `<execution>` section with goals or source directories.

To automatically configure kapt, set the `<extensions>` option to `true` for the `kotlin-maven-plugin` in your `pom.xml` build file:

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <extensions>true</extensions>
    <configuration>
        <annotationProcessorPaths>
            <!-- Specify your annotation processors here -->
            <annotationProcessorPath>
                <groupId>org.mapstruct</groupId>
                <artifactId>mapstruct-processor</artifactId>
                <version>1.6.3</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
</plugin>
```

For more information on the `<extensions>` option, see Automatic configuration.

Manual configuration

To manually set up kapt in your Kotlin Maven project, add an execution of the `kapt` goal from `kotlin-maven-plugin` before the `compile` execution:

```
<execution>
    <id>kapt</id>
    <goals>
        <goal>kapt</goal>
    </goals>
    <configuration>
        <sourceDirs>
            <sourceDir>src/main/kotlin</sourceDir>
            <sourceDir>src/main/java</sourceDir>
        </sourceDirs>
        <annotationProcessorPaths>
            <!-- Specify your annotation processors here -->
            <annotationProcessorPath>
                <groupId>org.mapstruct</groupId>
                <artifactId>mapstruct-processor</artifactId>
                <version>1.6.3</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
</execution>
```

To configure the annotation processing mode, set the `aptMode` option in the `<configuration>` block. For example:

```
<configuration>
   ...
   <aptMode>stubs</aptMode>
</configuration>
```

CLI

kapt is available as a standalone CLI tool in the binary distribution of the Kotlin compiler.

To run kapt from the command line, use:

```
kapt <options> <source files>
```

For example:

```
kapt -Kapt-mode=stubsAndApt \
  -Kapt-sources=build/kapt/sources \
  -Kapt-classes=build/kapt/classes \
  -Kapt-stubs=build/kapt/stubs \
  -Kapt-classpath=lib/ap.jar \
  -Kapt-classpath=lib/anotherAp.jar \
  src/main/kotlin
```

- See the full list of kapt-specific compiler options.
- You can also pass all valid Kotlin compiler options. Run `kotlinc -help` to see them.

Configure annotation processors

kapt provides options to control how annotation processors are discovered, organized, and executed, including managing the processor classpath, inheriting processors from shared configurations, and keeping javac-specific processors active.

For more configuration options, such as passing options to annotation processors and javac, see Annotation processor configuration.

Configure processor classpath and discovery

You can disable the discovery of annotation processors that aren't included in kapt's processor path. This excludes unnecessary annotation processors from the compile classpath.

Gradle

Gradle uses compile avoidance to skip annotation processing during project rebuild, improving incremental build times with kapt. Particularly, annotation processing is skipped when:

- The project's source files are unchanged.
- The changes in dependencies are ABI-compatible. For example, when only function bodies change.

However, compile avoidance can't be used for annotation processors discovered on the compile classpath, since changes in their internal implementation require running the annotation processing tasks, even if the processors' ABI remains unchanged.

That's why we don't recommend using annotation processors from the compile classpath. To exclude these processors from kapt processing, add the `kapt.include.compile.classpath` property to your `gradle.properties` file:

```
# gradle.properties
kapt.include.compile.classpath=false
```

With the option set to `false`, annotation processor dependencies that aren't included in the processor path (the `kapt*` configurations) are excluded from kapt processing.

Maven

To exclude annotation processors that aren't included in the kapt's processor path, set the `includeCompileClasspath` option to `false` in the `<execution>` section of the kapt plugin:

```
<execution>
    <id>kapt</id>
    <goals>
        <goal>kapt</goal>
    </goals>
    <configuration>
        <includeCompileClasspath>false</includeCompileClasspath>
        <sourceDirs>...</sourceDirs>
        <annotationProcessorPaths>...</annotationProcessorPaths>
    </configuration>
</execution>
```

Alternatively, you can use the `kapt.include.compile.classpath` property in the `<properties>` section of your `pom.xml`:

```
<properties>
    <kapt.include.compile.classpath>false</kapt.include.compile.classpath>
</properties>
```

With the option set to `false`, annotation processors that aren't included in the `<annotationProcessorPaths>` section are excluded from kapt processing.

If the `includeCompileClasspath` option isn't set and kapt detects an annotation processor on the compile classpath that isn't explicitly defined in the processor path, you'll see a deprecation warning:

```
[WARNING] Annotation processors discovery from compile classpath is deprecated.
Set 'kapt.include.compile.classpath=false' to disable discovery.
```

To see the list of annotation processors that aren't included on the kapt classpath, run the build with the `--info` log level option.

Inherit annotation processors from superconfigurations

You can define a common set of annotation processors in a separate Gradle configuration as a superconfiguration and extend it further in kapt-specific configurations for your subprojects.

As an example, for a subproject using MapStruct, in your `build.gradle(.kts)` file, use the following configuration:

```
val commonAnnotationProcessors by configurations.creating
configurations.named("kapt") { extendsFrom(commonAnnotationProcessors) }

dependencies {
    implementation("org.mapstruct:mapstruct:1.6.3")
    commonAnnotationProcessors("org.mapstruct:mapstruct-processor:1.6.3")
}
```

In this example, the `commonAnnotationProcessors` Gradle configuration is the common superconfiguration for annotation processing that you want to be used for all your projects. You use the `extendsFrom()` method to add `commonAnnotationProcessors` as a superconfiguration. kapt sees that the `commonAnnotationProcessors` Gradle configuration has a dependency on the MapStruct annotation processor. Therefore, kapt includes the MapStruct annotation processor in its configuration for annotation processing.

Keep Java compiler's annotation processors

By default, kapt runs all annotation processors and disables annotation processing by javac. However, you may need javac to run some annotation processors, such as Lombok.

In the Gradle build file, use the option `keepJavacAnnotationProcessors`:

```
kapt {
    keepJavacAnnotationProcessors = true
}
```

If you use Maven, configure the plugin explicitly. See this example of setting up the Lombok compiler plugin.

Optimize kapt builds

kapt offers several Gradle-specific strategies to reduce annotation processing time, including running tasks in parallel, leveraging the build cache, caching processor classloaders, and using incremental annotation processing.

For more options that affect build behavior, such as error type correction, stub metadata stripping, and compile classpath scanning, see Behavioral options.

Run kapt tasks in parallel

kapt uses the Gradle Worker API to run annotation processing tasks. Using the Worker API lets Gradle run independent annotation processing tasks from a single project in parallel, which in some cases significantly decreases the execution time.

If you set the custom JDK version in the Kotlin Gradle plugin, kapt task workers use only the `processIsolation()` mode.

If you want to provide additional JVM arguments for a kapt worker process, use the input `kaptProcessJvmArgs` of the `KaptWithoutKotlincTask`:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.internal.KaptWithoutKotlincTask>()
    .configureEach {
        kaptProcessJvmArgs.add("-Xmx512m")
    }
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.internal.KaptWithoutKotlincTask.class)
    .configureEach {
        kaptProcessJvmArgs.add('-Xmx512m')
    }
```

Use Gradle build cache safely

Gradle caches kapt annotation processing tasks by default. However, annotation processors can run arbitrary code. This may result in unnecessary transformations of task inputs into outputs, or accessing and modifying files that Gradle doesn't track.

When annotation processors used in the build can't be properly cached, you can disable caching to prevent false-positive hits for kapt tasks. To do this, use the `useBuildCache` property in the build script:

```
kapt {
    useBuildCache = false
}
```

Experimental

Cache annotation processors' classloaders

Caching for annotation processors' classloaders helps kapt perform faster if you run many Gradle tasks consecutively.

To enable this feature, use the following properties in your `gradle.properties` file:

```
# gradle.properties
#
# Any positive value enables caching
# Use the same value as the number of modules that use kapt
kapt.classloaders.cache.size=5

# Disable for caching to work
kapt.include.compile.classpath=false
```

If you run into any problems with caching for annotation processors, disable caching for them:

```
# Specify annotation processors' full names to disable caching for them
kapt.classloaders.cache.disableForProcessors=[annotation processors full names]
```

If you encounter any issues with the feature, we would appreciate your feedback in YouTrack.

Use incremental annotation processing

With Gradle, kapt supports incremental annotation processing by default so that only the changed files are re-processed.

Currently, incremental annotation processing works only if:

- Incremental compilation is enabled.
- All annotation processors in the build are incremental.

To disable incremental annotation processing, add this line to your `gradle.properties` file:

```
kapt.incremental.apt=false
```

Currently, incremental annotation processing for kapt isn't supported in Maven or the CLI.

Analyze performance

kapt provides built-in diagnostics to help you understand annotation processing performance, including per-processor execution time reports and generated file counts to identify unused processors.

For more diagnostic options, such as file read history for debugging incremental processing and memory leak detection, see Diagnostics and statistics options.

Measure the performance of annotation processors

To get performance statistics on the annotation processors execution, use the `showProcessorStats` option. An example output:

```
Kapt Annotation Processing performance report:
com.example.processor.TestingProcessor: total: 133 ms, init: 36 ms, 2 round(s): 97 ms, 0 ms
com.example.processor.AnotherProcessor: total: 100 ms, init: 6 ms, 1 round(s): 93 ms
```

You can dump this report to a file with the `dumpProcessorStats` option. For example, the following CLI command runs kapt and dumps the statistics to the `ap-perf-report.file` file:

```
kapt -Kapt-mode=stubsAndApt \
  -Kapt-classpath=processor/build/libs/processor.jar \
  -Kapt-dump-processor-stats=ap-perf-report.file \
  sample/src/main/
```

Track the number of generated files

The kapt plugin can report statistics on the number of generated files for each annotation processor.

This helps track whether any unused annotation processors are included in the build. You can use the generated report to find modules that trigger unnecessary annotation processors and update the modules to avoid that.

To enable statistics reporting:

1. In your Gradle build file, set the `showProcessorStats` option to `true`:

   ```
   // build.gradle(.kts)
   kapt {
       showProcessorStats = true
   }
   ```

2. In your `gradle.properties` file, set the `verbose` compiler option to `true`:

   ```
   # gradle.properties
   kapt.verbose=true
   ```

The statistics appear in the logs with the `info` level. You can see the `Annotation processor stats:` line followed by statistics on the execution time of each annotation processor. After these lines, there is the `Generated files report:` line followed by statistics on the number of generated files for each annotation processor. For example:

```
[INFO] Annotation processor stats:
[INFO] org.mapstruct.ap.MappingProcessor: total: 290 ms, init: 1 ms, 3 round(s): 289 ms, 0 ms, 0 ms
[INFO] Generated files report:
[INFO] org.mapstruct.ap.MappingProcessor: total sources: 2, sources per round: 2, 0, 0
```

Currently, tracking the number of generated files with the `showProcessorStats` and `verbose` compiler options isn't supported in Maven or the CLI.

Generate Kotlin sources

kapt can generate Kotlin sources. To do that, write the generated Kotlin source files to the specified directory using `processingEnv.options["kapt.kotlin.generated"]`. Kotlin source files are then compiled together with the main sources.

kapt doesn't support multiple rounds of annotation processing for the generated Kotlin files.

Compiler options

Annotation processor configuration

Option	Description	How to set up
`aptMode`	Controls the execution of kapt workflow stages: • `stubsAndApt` generates stubs and runs annotation processing (default) • `stubs` only generates Java stubs from Kotlin • `apt` only runs annotation processors (assumes stubs already exist)	Gradle: not directly available; Gradle runs stubs and apt as separate tasks Maven: `<aptMode>stubsAndApt</aptMode>` CLI: `-Kapt-mode=stubsAndApt`

`classpath`	Classpath entries where annotation processors are discovered.	Gradle:
		`dependencies { kapt("com.example:processor:1.0") }`
		Maven:
		`<annotationProcessorPaths> <annotationProcessorPath>...</annotationProcessorPath> </annotationProcessorPaths>`
		CLI: `-Kapt-classpath=lib/my-processor.jar`

`processors`	Comma-separated fully qualified class	Gradle:

Option	Description	How to set up

	names of processors to run, bypassing discovery.	`kapt { annotationProcessor("com.example.MyProcessor") }`
		Maven:
		`<annotationProcessors><annotationProcessor>com.example.MyProcessor</annotationProcessor></annotationProcessors>`
		CLI: `-Kapt-processors=com.example.MyProcessor`

`apOption`	Key-value options passed to annotation processors.	Gradle:
		`kapt { arguments { arg("room.schemaLocation", "$projectDir/schemas") } }`
		Maven:
		`<annotationProcessorArgs><annotationProcessorArg>room.schemaLocation=/schemas</annotationProcessorArg></annotationProcessorArgs>`
		CLI: `-Kapt-options:room.schemaLocation=/schemas`

`javacOption`	Key-value options passed to the Java compiler.	Gradle:
		`kapt { javacOptions { option("-source", "11") } }`
		Maven:
		`<javacOptions><javacOption>-source=11</javacOption></javacOptions>`
		CLI: `-Kapt-javac-option:-source=11`

`processIncrementally`	Enables incremental annotation processing; only reprocesses	Gradle: `# gradle.properties kapt.incremental.apt=true` Maven: currently not supported

Option	Description	How to set up
	files affected by changes.	CLI: currently not supported

Output directory options

Option	Description	How to set up
`sources`	Directory where annotation processors generate `.java` source files.	Gradle: set automatically to `build/generated/source/kapt/main` Maven: set automatically to `target/generated-sources/kapt/` CLI: `-Kapt-sources=build/kapt/sources`
`classes`	Directory for `.class` files compiled from generated sources.	Gradle: managed automatically Maven: managed automatically CLI: `-Kapt-classes=build/kapt/classes`
`stubs`	Directory for Java stub files generated from Kotlin sources, used as input to annotation processors.	Gradle: managed automatically Maven: managed automatically CLI: `-Kapt-stubs=build/kapt/stubs`
`incrementalData`	Stores state for incremental builds.	Gradle: managed automatically Maven: currently not supported CLI: currently not supported

Behavioral options

Option	Description	How to set up
`correctErrorTypes`	By default, kapt replaces every unknown type (including types for the generated classes) with `NonExistentClass`. You can enable error type interference in stubs to replace unresolved	Gradle: `kapt { correctErrorTypes = true }` Maven:

Option	Description	How to set up
	error types with types from generated sources. `false` by default	`<correctErrorTypes>true</correctErrorTypes>` CLI: `-Kapt-correct-error-types=true`
`dumpDefaultParameterValues`	Includes default parameter initializers as field values in generated stubs. `false` by default	Gradle: `kapt { dumpDefaultParameterValues = true }` Maven: not available CLI: `-Kapt-dump-default-parameter-values=true`

`mapDiagnosticLocations`	Maps error messages from stub files back to their original Kotlin source locations.
	`false` by default
		Gradle:
		`kapt { mapDiagnosticLocations = true }`
		Maven:
		`<mapDiagnosticLocations>true</mapDiagnosticLocations>`
		CLI: `-Kapt-map-diagnostic-locations=true`

`strict`	Turns stub generation incompatibilities into errors instead of warnings. `false` by default	Gradle: `kapt { strictMode = true }` Maven: not available CLI: `-Kapt-strict=true`
`stripMetadata`	Removes `@kotlin.Metadata` annotations from generated stubs, reducing stub size and hiding Kotlin-specific info from processors. `false` by default	Gradle: `kapt { stripMetadata = true }` Maven: not available CLI: `-Kapt-strip-metadata=true`
`verbose`	Enables verbose kapt logging. `false` by default	Gradle:

Option	Description	How to set up

		`# gradle.properties`
		`kapt.verbose=true`
		
		Maven: currently not supported
		
		CLI: currently not supported

`infoAsWarnings`	Promotes info-level kapt messages to warnings. `false` by default	Gradle: not directly available Maven: currently not supported CLI: currently not supported

`includeCompileClasspath`	Scans the compile classpath for annotation processors. Set to `false` for reproducibility.
	
	`true` by default
		Gradle:
		`kapt {`
		`includeCompileClasspath = false`
		`}`
		
		Maven:
		`<includeCompileClasspath>false</includeCompileClasspath>`
		
		CLI: currently not supported

Diagnostics and statistics options

Option	Description	How to set up

`showProcessorStats`	Prints per-processor execution time to stdout.	Gradle:
		`kapt {`
		`showProcessorStats = true`
		`}`
		
		Maven: not available
		
		CLI: `-Kapt-show-processor-stats=true`

`dumpProcessorStats`	Writes processor timing stats to a file.	Gradle: not available Maven: not available CLI: `-Kapt-dump-processor-stats=build/kapt-stats.txt`

Option	Description	How to set up
`dumpFileReadHistory`	Writes a list of files read by processors to a file, useful for incremental annotation processor debugging.	Gradle: not available Maven: not available CLI: `-Kapt-dump-file-read-history=build/kapt-reads.txt`
`detectMemoryLeaks`	Memory leak detection mode: `none`, `default`, or `paranoid`.	Gradle: `kapt { detectMemoryLeaks = "paranoid" }` Maven: currently not supported CLI: currently not supported

What's next

- Use kapt with the MapStruct annotation processor
- See how to migrate from kapt to KSP

Lombok compiler plugin

The Kotlin Lombok compiler plugin allows the generation and use of Java's Lombok declarations by Kotlin code in the same mixed Java/Kotlin module. If you call such declarations from another module, then you don't need to use this plugin for the compilation of that module.

The Lombok compiler plugin cannot replace Lombok, but it helps Lombok work in mixed Java/Kotlin modules. Thus, you still need to configure Lombok as usual when using this plugin. Learn more about how to configure the Lombok compiler plugin.

Supported annotations

The plugin supports the following annotations:

- `@Getter`, `@Setter`
- `@Builder`, `@SuperBuilder`
- `@NoArgsConstructor`, `@RequiredArgsConstructor`, and `@AllArgsConstructor`
- `@Data`
- `@With`
- `@Value`

We're continuing to work on this plugin. To find out the detailed current state, visit the Lombok compiler plugin's README.

Currently, we don't have plans to support the `@Tolerate` annotation. However, we can consider this if you vote for the @Tolerate issue in YouTrack.

Kotlin compiler ignores Lombok annotations if you use them in Kotlin code.

Gradle

Apply the `kotlin-plugin-lombok` Gradle plugin in the `build.gradle(.kts)` file:

Kotlin

```
plugins {
    kotlin("plugin.lombok") version "2.4.20"
    id("io.freefair.lombok") version "9.5.0"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.plugin.lombok' version '2.4.20'
    id 'io.freefair.lombok' version '9.5.0'
}
```

See this test project with examples of the Lombok compiler plugin in use.

Using the Lombok configuration file

If you use a Lombok configuration file `lombok.config`, you need to set the file's path so that the plugin can find it. The path must be relative to the module's directory. For example, add the following code to your `build.gradle(.kts)` file:

Kotlin

```
kotlinLombok {
    lombokConfigurationFile(file("lombok.config"))
}
```

Groovy

```
kotlinLombok {
    lombokConfigurationFile file("lombok.config")
}
```

See this test project with examples of the Lombok compiler plugin and `lombok.config` in use.

Maven

To use the Lombok compiler plugin, add the plugin `lombok` to the `compilerPlugins` section and the dependency `kotlin-maven-lombok` to the `dependencies` section. If you use a Lombok configuration file `lombok.config`, provide a path to it to the plugin in the `pluginOptions`. Add the following lines to the `pom.xml` file:

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <configuration>
        <compilerPlugins>
            <plugin>lombok</plugin>
        </compilerPlugins>
        <pluginOptions>
            <option>lombok:config=${project.basedir}/lombok.config</option>
        </pluginOptions>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-lombok</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.20</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</plugin>
```

See this test project example of the Lombok compiler plugin and `lombok.config` in use.

Using with kapt

By default, the kapt compiler plugin runs all annotation processors and disables annotation processing by javac. To run Lombok along with kapt, set up kapt to keep javac's annotation processors working.

If you use Gradle, add the option to the `build.gradle(.kts)` file:

```
kapt {
    keepJavacAnnotationProcessors = true
}
```

In Maven, use the following settings to launch Lombok with Java's compiler:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.5.1</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <annotationProcessorPaths>
            <annotationProcessorPath>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>${lombok.version}</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
</plugin>
```

The Lombok compiler plugin works correctly with kapt if annotation processors don't depend on the code generated by Lombok.

Look through the test project examples of kapt and the Lombok compiler plugin in use:

- Using Gradle.
- Using Maven

Command-line compiler

Lombok compiler plugin JAR is available in the binary distribution of the Kotlin compiler. You can attach the plugin by providing the path to its JAR file using the `Xplugin` kotlinc option:

```
-Xplugin=$KOTLIN_HOME/lib/lombok-compiler-plugin.jar
```

If you want to use the `lombok.config` file, replace `<PATH_TO_CONFIG_FILE>` with a path to your `lombok.config`:

```
# The plugin option format is: "-P plugin:<plugin id>:<key>=<value>".
# Options can be repeated.

-P plugin:org.jetbrains.kotlin.lombok:config=<PATH_TO_CONFIG_FILE>
```

No-arg compiler plugin

The *no-arg* compiler plugin generates an additional zero-argument constructor for classes with a specific annotation.

The generated constructor is synthetic, so it can't be directly called from Java or Kotlin, but it can be called using reflection.

This allows the Java Persistence API (JPA) to instantiate a class, although it doesn't have the zero-parameter constructor from Kotlin or Java point of view (see the description of `kotlin-jpa` plugin below).

In your Kotlin file

Add new annotations to mark the code that needs a zero-argument constructor:

```
package com.my

annotation class Annotation
```

Gradle

Add the plugin using Gradle's plugins DSL:

Kotlin

```
plugins {
    kotlin("plugin.noarg") version "2.4.20"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.plugin.noarg" version "2.4.20"
}
```

Then specify the list of no-arg annotations that must lead to generating a no-arg constructor for the annotated classes:

```
noArg {
    annotation("com.my.Annotation")
}
```

Enable `invokeInitializers` option if you want the plugin to run the initialization logic from the synthetic constructor. By default, it is disabled.

```
noArg {
    invokeInitializers = true
}
```

Maven

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- Or "jpa" for JPA support -->
            <plugin>no-arg</plugin>
        </compilerPlugins>

        <pluginOptions>
            <option>no-arg:annotation=com.my.Annotation</option>
            <!-- Call instance initializers in the synthetic constructor -->
            <!-- <option>no-arg:invokeInitializers=true</option> -->
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-noarg</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

JPA support

As with the `kotlin-spring` plugin wrapped on top of `all-open`, `kotlin-jpa` is wrapped on top of `no-arg`. The plugin specifies `@Entity`, `@Embeddable`, and `@MappedSuperclass` *no-arg* annotations automatically.

Add the plugin using the Gradle plugins DSL:

Kotlin

```
plugins {
    kotlin("plugin.jpa") version "2.4.20"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.plugin.jpa" version "2.4.20"
}
```

In Maven, enable the `jpa` plugin:

```
<compilerPlugins>
    <plugin>jpa</plugin>
</compilerPlugins>
```

Command-line compiler

Add the plugin JAR file to the compiler plugin classpath and specify annotations or presets:

```
-Xplugin=$KOTLIN_HOME/lib/noarg-compiler-plugin.jar
-P plugin:org.jetbrains.kotlin.noarg:annotation=com.my.Annotation
-P plugin:org.jetbrains.kotlin.noarg:preset=jpa
```

Experimental

Power-assert compiler plugin

The Kotlin Power-assert compiler plugin improves the debugging experience by providing detailed failure messages with contextual information. It simplifies the process of writing tests by automatically generating intermediate values in failure messages. It helps you understand why a test failed without needing complex assertion libraries.

This is an example message provided by the plugin:

```
Incorrect length
assert(hello.length == world.substring(1, 4).length) { "Incorrect length" }
       |     |      |  |     |               |
       |     5      |  |     "orl"           3
       "Hello"      |  "world!"
                    false
```

The Power-assert plugin key features:

- **Enhanced error messages**: The plugin captures and displays the values of variables and sub-expressions within the assertion to clearly identify the cause of failure.
- **Runtime library**: The library provides the `@PowerAssert` annotation and the `CallExplanation` class. They make Power-assert capable functions more discoverable and easier to configure by integrating them directly with the compiler plugin transformations.
- **Simplified testing**: Automatically generates informative failure messages, reducing the need for complex assertion libraries.
- **Support for multiple functions**: By default, it transforms `assert()` function calls but can also transform other functions, such as `require()`, `check()`, and `assertTrue()`.

Apply the plugin

Gradle

To enable the Power-assert plugin, configure your `build.gradle(.kts)` file as follows:

Kotlin

```kotlin
// build.gradle.kts
plugins {
    kotlin("multiplatform") version "2.4.20"
    kotlin("plugin.power-assert") version "2.4.20"
}
```

Groovy

```groovy
// build.gradle
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.4.20'
    id 'org.jetbrains.kotlin.plugin.power-assert' version '2.4.20'
}
```

The Power-assert plugin provides several options to customize its behavior:

- **`functions`** lists the fully qualified paths to functions that the Power-assert plugin transforms when they're called. If not specified, the plugin transforms only `kotlin.assert()` calls.
- **`compilationFilter`** controls which Kotlin compilations the Power-assert plugin is applied to. You can create your own custom filter or use the predefined options:
  - `PowerAssertCompilationFilter.TESTS` applies to all test source sets (default).
  - `PowerAssertCompilationFilter.ALL` applies to all source sets.

> The `compilationFilter` option has replaced the deprecated `includedSourceSets`, which listed Gradle source sets that the Power-assert plugin transforms. These two options are mutually exclusive: if `includedSourceSets` is specified, `compilationFilter` is ignored.

To customize the behavior, add the `powerAssert {}` block to your build script file:

Kotlin

```kotlin
// build.gradle.kts
powerAssert {
    functions = listOf("kotlin.assert", "kotlin.test.assertTrue", "kotlin.test.assertEquals",
"kotlin.test.assertNull")
    compilationFilter = PowerAssertCompilationFilter {
        it.name in setOf("commonMain", "jvmMain", "jsMain", "nativeMain")
    }
}
```

Groovy

```
// build.gradle
powerAssert {
    functions = ["kotlin.assert", "kotlin.test.assertTrue", "kotlin.test.assertEquals",
"kotlin.test.assertNull"]
    compilationFilter = PowerAssertCompilationFilter {
        it.name in ["commonMain", "jvmMain", "jsMain", "nativeMain"]
    }
}
```

Since the plugin is Experimental, you will see warnings every time you build your app. To exclude these warnings, add this `@OptIn` annotation before declaring the `powerAssert {}` block:

```
import org.jetbrains.kotlin.gradle.ExperimentalKotlinGradlePluginApi

@OptIn(ExperimentalKotlinGradlePluginApi::class)
powerAssert {
    ...
}
```

Maven

To enable the Power-assert compiler plugin in a Maven project, update the `<plugin>` section of `kotlin-maven-plugin` in the `pom.xml` file:

```
<build>
    <plugins>
        <plugin>
            <artifactId>kotlin-maven-plugin</artifactId>
            <groupId>org.jetbrains.kotlin</groupId>
            <version>2.4.20</version>
            <executions>
                <execution>
                    <id>compile</id>
                    <phase>process-sources</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>test-compile</id>
                    <phase>process-test-sources</phase>
                    <goals>
                        <goal>test-compile</goal>
                    </goals>
                </execution>
            </executions>

            <configuration>
                <!-- Specify the Power-assert plugin -->
                <compilerPlugins>
                    <plugin>power-assert</plugin>
                </compilerPlugins>
            </configuration>

            <!-- Add the Power-assert plugin dependency -->
            <dependencies>
                <dependency>
                    <groupId>org.jetbrains.kotlin</groupId>
                    <artifactId>kotlin-maven-power-assert</artifactId>
                    <version>2.4.20</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
```

You can customize which functions the Power-assert plugin transforms by using the `function` option. For example, you can include `kotlin.test.assertTrue()`, `kotlin.test.assertEquals()`, and others. If not specified, only `kotlin.assert()` calls are transformed by default.

Specify this option in the `<configuration>` section of `kotlin-maven-plugin`:

```
<configuration>
    <!-- Specify the functions to transform -->
    <pluginOptions>
        <option>power-assert:function=kotlin.assert</option>
        <option>power-assert:function=kotlin.test.assertTrue</option>
        <option>power-assert:function=kotlin.test.AssertEquals</option>
    </pluginOptions>
</configuration>
```

Use the Power-assert plugin

This section provides examples of using the Power-assert compiler plugin.

See the complete code of the build script file `build.gradle.kts` or `pom.xml` for all these examples:

Gradle (Kotlin)

```
// build.gradle.kts

import org.jetbrains.kotlin.gradle.ExperimentalKotlinGradlePluginApi

plugins {
    kotlin("multiplatform") version "2.4.20"
    kotlin("plugin.power-assert") version "2.4.20"
}

group = "com.example"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

dependencies {
    testImplementation(kotlin("test"))
}

tasks.test {
    useJUnitPlatform()
}

@OptIn(ExperimentalKotlinGradlePluginApi::class)
powerAssert {
    functions = listOf("kotlin.assert", "kotlin.test.assertEquals", "kotlin.test.assertTrue",
"kotlin.test.assertNull", "kotlin.require", "com.example.AssertScope.assert")
}
```

Gradle (Groovy)

```
// build.gradle
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.4.20'
    id 'org.jetbrains.kotlin.plugin.power-assert' version '2.4.20'
}

group = 'com.example'
version = '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'org.jetbrains.kotlin:kotlin-test'
}

test {
    useJUnitPlatform()
}

powerAssert {
    functions = [
            'kotlin.assert',
            'kotlin.test.assertEquals',
            'kotlin.test.assertTrue',
            'kotlin.test.assertNull',
            'kotlin.require',
            'com.example.AssertScope.assert'
    ]
}
```

Maven

```
<!-- pom.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>maven-power-assert-plugin-demo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <kotlin.code.style>official</kotlin.code.style>
        <kotlin.compiler.jvmTarget>1.8</kotlin.compiler.jvmTarget>
    </properties>

    <repositories>
        <repository>
            <id>mavenCentral</id>
            <url>https://repo1.maven.org/maven2/</url>
        </repository>
    </repositories>

    <build>
        <sourceDirectory>src/main/kotlin</sourceDirectory>
        <testSourceDirectory>src/test/kotlin</testSourceDirectory>
        <plugins>
            <plugin>
                <groupId>org.jetbrains.kotlin</groupId>
                <artifactId>kotlin-maven-plugin</artifactId>
                <version>2.4.20</version>
                <executions>
                    <execution>
                        <id>compile</id>
                        <phase>compile</phase>
                        <goals>
                            <goal>compile</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>test-compile</id>
                        <phase>test-compile</phase>
                        <goals>
                            <goal>test-compile</goal>
                        </goals>
                    </execution>
                </executions>

                <configuration>
                    <compilerPlugins>
                        <plugin>power-assert</plugin>
                    </compilerPlugins>

                    <pluginOptions>
                        <option>power-assert:function=kotlin.assert</option>
                        <option>power-assert:function=kotlin.require</option>
                        <option>power-assert:function=kotlin.test.assertTrue</option>
                        <option>power-assert:function=kotlin.test.assertEquals</option>
```

```
                        <option>power-assert:function=kotlin.test.assertNull</option>
                        <option>power-assert:function=com.example.AssertScope.assert</option>
                    </pluginOptions>
                </configuration>

                <dependencies>
                    <dependency>
                        <groupId>org.jetbrains.kotlin</groupId>
                        <artifactId>kotlin-maven-power-assert</artifactId>
                        <version>2.4.20</version>
                    </dependency>
                </dependencies>

            </plugin>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.2</version>
            </plugin>
            <plugin>
                <artifactId>maven-failsafe-plugin</artifactId>
                <version>2.22.2</version>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>1.6.0</version>
                <configuration>
                    <mainClass>MainKt</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-test-junit5</artifactId>
            <version>2.4.20</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.10.0</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-stdlib</artifactId>
            <version>2.4.20</version>
        </dependency>
    </dependencies>
</project>
```

`@PowerAssert`-annotated functions

If a function is annotated with `@PowerAssert`, the Power-assert plugin transforms calls to it automatically. You don't need to register the function in your build configuration.

You can add `@PowerAssert` annotations when declaring assertion functions yourself, or use libraries that support Power-assert and provide annotated functions.

To get detailed failure messages, call the function with the Power-assert plugin enabled in your project:

```
import kotlin.test.Test

data class Mascot(val name: String)

class SampleTest {

    @Test
    fun testAnnotatedFunction() {
        val subject: Any? = Mascot(name = "Unknown")
        // If assertThat() is annotated with @PowerAssert in the library,
        // the plugin transforms this call automatically
        assertThat(subject) {
            require(subject is Mascot)
            check(subject.name == "Kodee")
        }
    }
}
```

The plugin provides detailed failure messages with intermediate expression values:

```
check(subject.name == "Kodee")
      |       |    |
      |       |    false
      |       "Unknown"
      Mascot(name=Unknown)
```

Assert function

Consider the following test with the `assert()` function:

```
import kotlin.test.Test

class SampleTest {

    @Test
    fun testFunction() {
        val hello = "Hello"
        val world = "world!"
        assert(hello.length == world.substring(1, 4).length) { "Incorrect length" }
    }
}
```

If you run the `testFunction()` test with the Power-assert plugin enabled, you get the explicit failure message:

```
Incorrect length
assert(hello.length == world.substring(1, 4).length) { "Incorrect length" }
       |     |      |  |     |               |
       |     5      |  |     "orl"           3
       "Hello"      |  "world!"
                    false
```

To get a more complete error message, always inline the variable into the test function parameters. Consider the following test function:

```
class ComplexExampleTest {

    data class Person(val name: String, val age: Int)

    @Test
    fun testComplexAssertion() {
        val person = Person("Alice", 10)
        val isValidName = person.name.startsWith("A") && person.name.length > 3
        val isValidAge = person.age in 21..28
        assert(isValidName && isValidAge)
    }
}
```

The output of the executed code doesn't provide enough information to find the cause of the problem:

```
assert(isValidName && isValidAge)
       |              |
       true           false
```

Inline the variable into the `assert()` function:

```
class ComplexExampleTest {

    data class Person(val name: String, val age: Int)

    @Test
    fun testComplexAssertion() {
        val person = Person("Alice", 10)
        assert(person.name.startsWith("A") && person.name.length > 3 && person.age > 20 && person.age
< 29)
    }
}
```

After execution, you get more explicit information about what went wrong:

```
assert(person.name.startsWith("A") && person.name.length > 3 && person.age > 20 && person.age < 29)
       |      |    |                 |      |    |      |      |      |   |
       |      |    true              |      |    5      true   |      10  false
       |      "Alice"                |      "Alice"            Person(name=Alice, age=10)
       Person(name=Alice, age=10)    Person(name=Alice, age=10)
```

Beyond assert function

The Power-assert plugin can transform various functions beyond `assert` which is transformed by default. Functions like `require()`, `check()`, `assertTrue()`, `assertEqual()` and others can also be transformed, if they have a form that allows taking a `String` or `() -> String` value as the last parameter.

Before using a new function in a test, add the function to your build file. For example, the `require()` function:

Gradle (Kotlin)

```kotlin
// build.gradle.kts
import org.jetbrains.kotlin.gradle.ExperimentalKotlinGradlePluginApi

@OptIn(ExperimentalKotlinGradlePluginApi::class)
powerAssert {
    functions = listOf("kotlin.assert", "kotlin.require")
}
```

Gradle (Groovy)

```groovy
powerAssert {
    functions = [
            'kotlin.assert',
            'kotlin.require'
    ]
}
```

Maven

```xml
<!-- pom.xml -->
<configuration>
    <pluginOptions>
        <option>power-assert:function=kotlin.assert</option>
        <option>power-assert:function=kotlin.require</option>
    </pluginOptions>
</configuration>
```

After adding the function, you can use it in your tests:

```kotlin
class RequireExampleTest {

    @Test
    fun testRequireFunction() {
        val value = ""
        require(value.isNotEmpty()) { "Value should not be empty" }
    }
}
```

The output for this example uses the Power-assert plugin to provide detailed information about the failed test:

```
Value should not be empty
require(value.isNotEmpty()) { "Value should not be empty" }
        |     |
        ""    false
```

The message shows the intermediate values that lead to the failure, making it easier to debug.

Soft assertions

The Power-assert plugin supports soft assertions, which do not immediately fail the test but instead collect assertion failures and report them at the end of the test run. This can be useful when you want to see all assertion failures in a single run without stopping at the first failure.

To enable soft assertions, implement the way you will collect error messages:

```
fun <R> assertSoftly(block: AssertScope.() -> R): R {
    val scope = AssertScopeImpl()
    val result = scope.block()
    if (scope.errors.isNotEmpty()) {
        throw AssertionError(scope.errors.joinToString("\n"))
    }
    return result
}

interface AssertScope {
    fun assert(assertion: Boolean, message: (() -> String)? = null)
}

class AssertScopeImpl : AssertScope {
    val errors = mutableListOf<String>()
    override fun assert(assertion: Boolean, message: (() -> String)?) {
        if (!assertion) {
            errors.add(message?.invoke() ?: "Assertion failed")
        }
    }
}
```

Add these functions to your build file to make them available for the Power-assert plugin:

Gradle (Kotlin)

```
// build.gradle.kts
import org.jetbrains.kotlin.gradle.ExperimentalKotlinGradlePluginApi

@OptIn(ExperimentalKotlinGradlePluginApi::class)
powerAssert {
    functions = listOf("kotlin.assert", "kotlin.test.assert", "com.example.AssertScope.assert")
}
```

Gradle (Groovy)

```
powerAssert {
    functions = [
            'kotlin.assert',
            'kotlin.test.assert',
            'com.example.AssertScope.assert'
    ]
}
```

Maven

```
<!-- pom.xml -->
<configuration>
    <pluginOptions>
        <option>power-assert:function=kotlin.assert</option>
        <option>power-assert:function=kotlin.require</option>
        <option>power-assert:function=com.example.AssertScope.assert</option>
    </pluginOptions>
</configuration>
```

You should specify the full name of the package where you declare the `AssertScope.assert()` function.

After that, you could use it in your test code:

```
// Import the assertSoftly() function
import com.example.assertSoftly

class SoftAssertExampleTest1 {

    data class Employee(val name: String, val age: Int, val salary: Int)

    @Test
    fun `test employees data`() {
        val employees = listOf(
            Employee("Alice", 30, 60000),
            Employee("Bob", 45, 80000),
            Employee("Charlie", 55, 40000),
            Employee("Dave", 150, 70000)
        )

        assertSoftly {
            for (employee in employees) {
                assert(employee.age < 100) { "${employee.name} has an invalid age: ${employee.age}" }
                assert(employee.salary > 50000) { "${employee.name} has an invalid salary:
${employee.salary}" }
            }
        }
    }
}
```

In the output, all the `assert()` function error messages will be printed one after another:

```
Charlie has an invalid salary: 40000
assert(employee.salary > 50000) { "${employee.name} has an invalid salary: ${employee.salary}" }
       |        |      |
       |        40000  false
       Employee(name=Charlie, age=55, salary=40000)

Dave has an invalid age: 150
assert(employee.age < 100) { "${employee.name} has an invalid age: ${employee.age}" }
       |        |   |
       |        150 false
       Employee(name=Dave, age=150, salary=70000)
```

Add support for Power-assert to your library

If you are a library author, you can add out-of-the-box Power-assert support to your library using the `@PowerAssert` annotation and the `CallExplanation` class from the Power-assert runtime library.

The `@PowerAssert` annotation

The `@PowerAssert` annotation marks a function as Power-assert capable. If users of your library have the Power-assert compiler plugin in their projects and make calls to your annotated functions, the calls are automatically transformed without additional build configuration.

To add support for Power-assert to your library:

1. In your build file, apply the Power-assert plugin.
2. For Maven, add the Power-assert runtime library as a dependency:

```
<!-- pom.xml -->
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-power-assert-runtime</artifactId>
        <version>2.4.20</version>
    </dependency>
</dependencies>
```

   For Gradle, this dependency is automatically added with the Power-assert compiler plugin.

3. Annotate your assertion functions with `@PowerAssert`:

```
import kotlin.powerassert.PowerAssert
import kotlin.powerassert.toDefaultMessage
import kotlin.contracts.ExperimentalContracts
import kotlin.contracts.contract

@OptIn(ExperimentalContracts::class)
@PowerAssert
fun powerAssert(condition: Boolean, @PowerAssert.Ignore message: String? = null) {
    contract { returns() implies condition }
    if (!condition) {
        val explanation = PowerAssert.explanation
            ?: fail(message)

        val equalityErrors = buildList {
            for (expression in explanation.expressions) {
                if (expression is EqualityExpression && expression.value == false) {
                    add(expression)
                }
            }
        }

        val failureMessage = buildString {
            if (message?.isNotBlank() == true) appendLine(message)
            append(explanation.toDefaultMessage())
        }

        fail(failureMessage, equalityErrors)
    }
}
```

- The `PowerAssert.explanation` property provides access to the `CallExplanation` object containing the call site information.
- The `toDefaultMessage()` function renders the standard Power-assert failure message.
- The `@PowerAssert.Ignore` annotation on the `message` parameter excludes it from the failure message.

The compiler plugin detects the `@PowerAssert` annotation and transforms calls to the function at compile time.

For a complete example, see the `kotlin-test-power-assert` project.

The `CallExplanation` class

The `CallExplanation` class provides detailed information about the call site, including intermediate expression values. This enables dynamic message rendering for assertion failures and better integration with external tools.

When a function in your library is annotated with `@PowerAssert` and the compiler plugin is applied, the transformation is performed automatically at each call site. The `PowerAssert.explanation` property provides access to the `CallExplanation` object inside the function body.

The `PowerAssert.explanation` property can return `null` if the annotated function is called from Java, from a project where the Power-assert plugin is not applied, or via reflection.

Here is an example of how to use `CallExplanation` inside `@PowerAssert`-annotated functions to extract source code information and build custom failure messages:

```
package kotlinx.test.fluent

import kotlin.powerassert.PowerAssert
import kotlin.contracts.ExperimentalContracts
import kotlin.contracts.contract

@PowerAssert
fun AssertScope<*>.check(condition: Boolean) {
    if (!condition) {
        val explanation = PowerAssert.explanation
        val message = if (explanation == null) null else {
            val conditionArg = explanation.arguments.last()!!
            val source = explanation.source.substring(conditionArg.startOffset,
conditionArg.endOffset)
            "Condition failed: $source"
        }
        collect(message, explanation)
    }
}

@OptIn(ExperimentalContracts::class)
@PowerAssert
fun AssertScope<*>.require(condition: Boolean) {
    contract { returns() implies condition }
    if (!condition) {
        val explanation = PowerAssert.explanation
        val message = if (explanation == null) null else {
            val conditionArg = explanation.arguments.last()!!
            val source = explanation.source.substring(conditionArg.startOffset,
conditionArg.endOffset)
            "Condition failed: $source"
        }
        fail(message, explanation)
    }
}
```

In this example, the `check()` function collects failures for later reporting, while the `require()` function fails immediately. Both functions use `CallExplanation` to extract the source code of the failed condition and include it in the failure message.

For a complete example, see the `fluent-assert` project.

What's next

Look through our sample projects:

- A simple project with the plugin enabled
- A more complex project with multiple source sets
- Example collection for experimenting with runtime library features

SAM-with-receiver compiler plugin

The *sam-with-receiver* compiler plugin makes the first parameter of the annotated Java "single abstract method" (SAM) interface method a receiver in Kotlin. This conversion only works when the SAM interface is passed as a

Kotlin lambda, both for SAM adapters and SAM constructors (see the SAM conversions documentation for more details).

Here is an example:

```
public @interface SamWithReceiver {}

@SamWithReceiver
public interface TaskRunner {
    void run(Task task);
}
```

```
fun test(context: TaskContext) {
    val runner = TaskRunner {
        // Here 'this' is an instance of 'Task'

        println("$name is started")
        context.executeTask(this)
        println("$name is finished")
    }
}
```

Gradle

The usage is the same to all-open and no-arg, except the fact that sam-with-receiver does not have any built-in presets, and you need to specify your own list of special-treated annotations.

Kotlin

```
plugins {
    kotlin("plugin.sam.with.receiver") version "2.4.20"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.plugin.sam.with.receiver" version "2.4.20"
}
```

Then specify the list of SAM-with-receiver annotations:

```
samWithReceiver {
    annotation("com.my.SamWithReceiver")
}
```

Maven

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <plugin>sam-with-receiver</plugin>
        </compilerPlugins>

        <pluginOptions>
            <option>
                sam-with-receiver:annotation=com.my.SamWithReceiver
            </option>
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-sam-with-receiver</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

Command-line compiler

Add the plugin JAR file to the compiler plugin classpath and specify the list of sam-with-receiver annotations:

```
-Xplugin=$KOTLIN_HOME/lib/sam-with-receiver-compiler-plugin.jar
-P plugin:org.jetbrains.kotlin.samWithReceiver:annotation=com.my.SamWithReceiver
```

Custom compiler plugins

The Kotlin compiler plugin API is unstable and introduces breaking changes in every release.

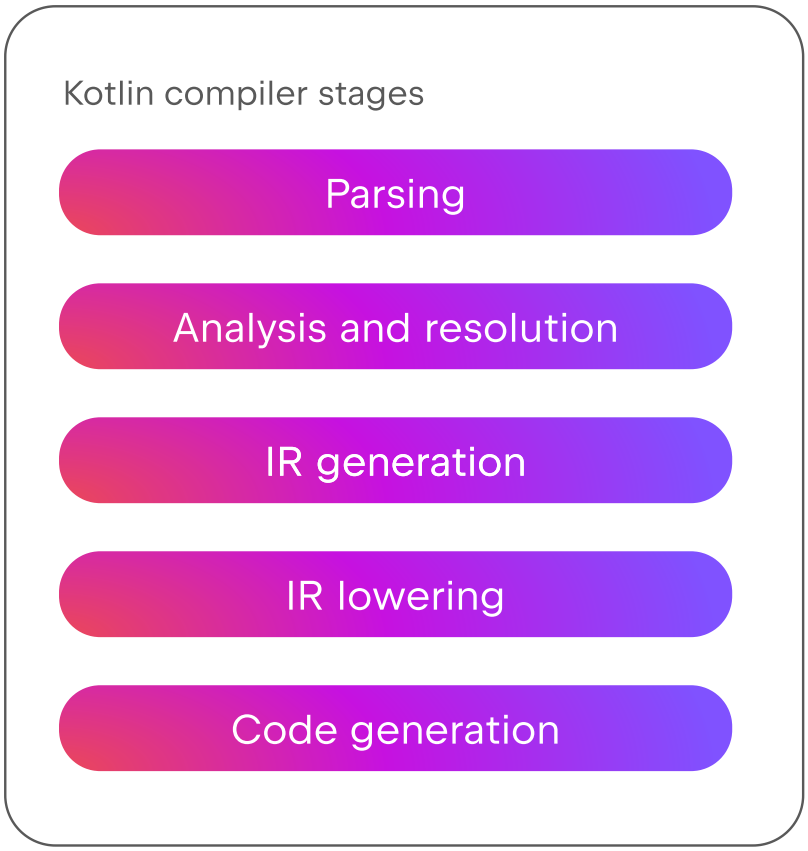
Compiler plugins hook into the compilation process to analyze or change code while it's being compiled, without modifying the compiler itself. For example, they can annotate code or generate new code to make it compatible with other frameworks or APIs.

Before you create your own custom compiler plugin, check the list of available compiler plugins to see if one is already available that suits your use case.

You can also check whether you can use the Kotlin Symbol Processing (KSP) API or an external linter such as Android lint to achieve your goals.

If you *still* can't find what you need, you can create a custom compiler plugin. Be aware that the Kotlin compiler plugin API is **unstable**. You need to invest significant ongoing effort to maintain it, since each new compiler release introduces breaking changes.

The Kotlin compiler and compiler plugins



Kotlin compiler stages

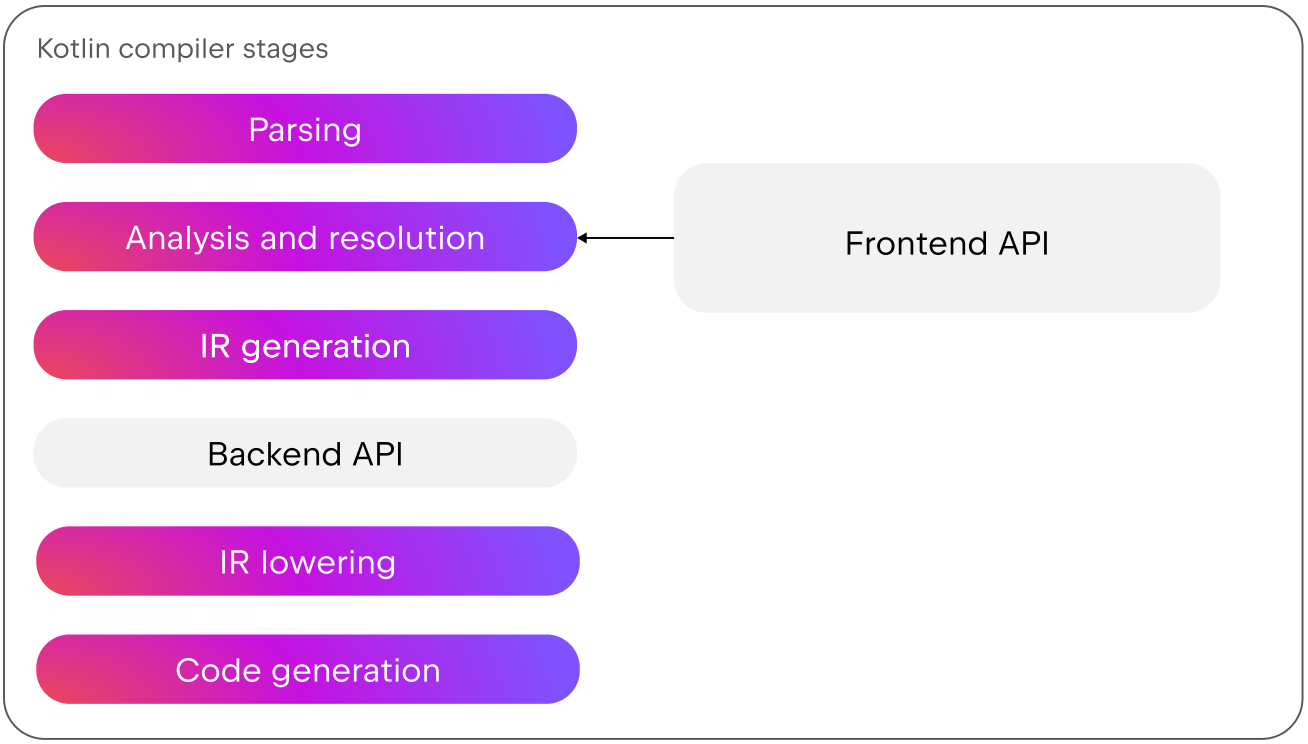
The Kotlin compiler:

1. Parses the source code and turns it into a structured syntax tree.
2. Analyzes and resolves the code by determining what it means, resolving names, checking types, and enforcing visibility rules.
3. Generates an Intermediate Representation (IR), a data structure that acts as a bridge between source code and machine code.
4. Progressively lowers the IR into simpler forms.
5. Translates the lowered IR into target-specific output, such as JVM bytecode, JavaScript, or native machine code.

Plugins can affect the initial compiler stages through the frontend API, changing how the compiler resolves code. For example, a plugin can add annotations or introduce new methods without bodies, or change visibility modifiers. These changes are visible in the IDE.

Plugins can also affect the later stages through the backend API, modifying the behavior of declarations. These changes appear in the binaries produced after compilation completes.

In practice, compiler plugins affect the stages from analysis and resolution through code generation, covering both frontend and backend. For example, the frontend part generates declarations, and the backend part adds bodies for those declarations.



Kotlin compiler stages with plugins

The Kotlin serialization plugin is a good example. The plugin's frontend part adds a companion object and a serializer function, as well as checks to prevent name conflicts. The backend part implements the desired serialization behavior through `KSerializer` objects.

Kotlin compiler plugin template

To start writing a custom compiler plugin, you can use the Kotlin compiler plugin template. You then register extension points from the frontend and backend plugin APIs.

Currently, you can develop custom compiler plugins only with Gradle.

Frontend plugin API

The frontend plugin API, also known as frontend intermediate representation (FIR), has the following specialized extension points to customize resolution:

Extension name	Description
`FirAdditionalCheckersExtension`	Adds custom compiler checkers.
`FirDeclarationGenerationExtension`	Generates new declarations.
`FirExtensionSessionComponent`	Registers custom components in the `FirSession` to be used by other parts of the plugin.

Extension name	Description
`FirFunctionTypeKindExtension`	Defines new families of functional types.
`FirMetadataSerializerPlugin`	Reads and writes information to declaration metadata.
`FirStatusTransformerExtension`	Modifies declaration status attributes such as visibility or modality.
`FirSupertypeGenerationExtension`	Adds new supertypes to an existing class.
`FirTypeAttributeExtension`	Adds special attributes to certain types based on their type annotations.

IDE integration

Resolution changes affect IDE behavior such as code highlighting and suggestions, so it's important that your plugin is compatible with the IDE. Each version of IntelliJ IDEA and Android Studio includes a development version of the Kotlin compiler. This version is specific to the IDE and is not binary compatible with the released Kotlin compiler. As a result, when you update your IDE, you also need to update your compiler plugin to keep it working. For this reason, community plugins aren't loaded by default.

To ensure that your custom compiler plugin works with different IDE versions, test it against each IDE version and fix any issues you find.

Supporting multiple IDE versions could become easier if a devkit for Kotlin compiler plugins were available. If you're interested in this feature, share your feedback in our issue tracker.

Backend plugin API

> Backend plugin development is difficult to do correctly without degrading IDE or debugger performance, so be careful and conservative with your changes.

The backend plugin API, also known as IR, has a single extension point: `IrGenerationExtension`. Use this extension point and override the `generate()` function to add bodies to declarations already generated by the frontend or change existing declaration bodies.

Changes made through this extension point are **not checked** by the compiler. You must ensure that your changes don't break the compiler's expectations at this stage. For example, you might accidentally introduce an invalid type, an incorrect function reference, or a reference outside the correct scope.

Explore backend plugin code

You can explore the Kotlin serialization plugin code to see what backend plugin compiler code looks like in practice. For example, `SerializableCompanionIrGenerator.kt` fills in missing bodies for key serializer members. One example is the `generateChildSerializersGetter()` function, which collects a list of `KSerializer` expressions and returns them in an array.

Check your backend plugin code for problems

You can check for problems in your backend plugin code in three ways:

1. **Verify the IR**

   Build the IR tree and enable the `Xverify-ir` compiler option. This option has a performance impact on compilation speed, so use it only during testing.

2. **Dump and compare IR output**

   Create a dump file after the IR lowering compilation stage with the `-Xphases-to-dump-before=ExternalPackageParentPatcherLowering` compiler option. For the JVM backend, configure the dump directory with the `-Xdump-directory=<your-file-directory>` compiler option. Write the expected code manually, generate another dump file, and compare the two to see if there are differences.

3. **Debug the compiler code**

   In the `convertToIr.kt` file, add breakpoints in the `convertToIrAndActualize()` function and run the compiler in debug mode to get more detailed information during compilation.

Test your plugin

Once you implement your plugin, test it thoroughly. The Kotlin compiler plugin template is already set up to use the Kotlin compiler test framework. You can add tests in the following directories:

- `compiler-plugin/testData`
- `compiler-plugin/testData/box` for code generation tests
- `compiler-plugin/testData/diagnostics` for diagnostic tests

When a test runs, the framework:

1. Parses the test source file. For example, `anotherBoxTest.kt`
2. Builds the FIR and IR for each file.
3. Writes these as textual dump files. For example, `anotherBoxTest.fir.txt` and `anotherBoxTest.fir.ir.txt` .
4. Compares these files with previously created files, if they exist.

You can use these files to check if any changes in the generated diff weren't intended. If there are no problems, the new dump files become your latest *golden* files: an approved and trusted source that you can compare future changes against.

Get help

If you run into issues developing a custom compiler plugin, reach out in Kotlin Slack in the #compiler channel. We can't promise a solution but we will try to help if we can.

Compose compiler migration guide

The Compose compiler is supplemented by a Gradle plugin, which simplifies setup and offers easier access to compiler options. When applied with the Android Gradle plugin (AGP), this Compose compiler plugin will override the coordinates of the Compose compiler supplied automatically by AGP.

The Compose compiler has been merged into the Kotlin repository since Kotlin 2.0.0. This helps smooth the migration of your projects to Kotlin 2.0.0 and later, as the Compose compiler ships simultaneously with Kotlin and will always be compatible with Kotlin of the same version.

To use the new Compose compiler plugin in your project, apply it for each module that uses Compose. Read on for details on how to migrate a Jetpack Compose project. For a Compose Multiplatform project, refer to the multiplatform migration guide.

Migrating a Jetpack Compose project

When migrating to Kotlin 2.0.0 or newer from 1.9, you should adjust your project configuration depending on the way you deal with the Compose compiler. We recommend using the Kotlin Gradle plugin and the Compose compiler Gradle plugin to automate configuration management.

Managing the Compose compiler with Gradle plugins

For Android modules:

1. Add the Compose compiler Gradle plugin to the Gradle version catalog:

```
[versions]
# ...
kotlin = "2.4.20"

[plugins]
# ...
org-jetbrains-kotlin-android = { id = "org.jetbrains.kotlin.android", version.ref = "kotlin" }
compose-compiler = { id = "org.jetbrains.kotlin.plugin.compose", version.ref = "kotlin" }
```

> If you're using AGP 9.0.0 or later, you no longer need the `org-jetbrains-kotlin-android` plugin because the AGP has built-in Kotlin support.

2. Add the Gradle plugin to the root `build.gradle.kts` file:

```
plugins {
    // ...
    alias(libs.plugins.compose.compiler) apply false
}
```

3. Apply the plugin to every module that uses Jetpack Compose:

```
plugins {
    // ...
    alias(libs.plugins.compose.compiler)
}
```

4. If you are using compiler options for the Jetpack Compose compiler, set them in the `composeCompiler {}` block. See the list of compiler options for reference.

5. If you reference Compose compiler artifacts directly, you can remove these references and let the Gradle plugins take care of things.

Using Compose compiler without Gradle plugins

If you are not using Gradle plugins to manage the Compose compiler, update any direct references to old Maven artifacts in your project:

- Change `androidx.compose.compiler:compiler` to `org.jetbrains.kotlin:kotlin-compose-compiler-plugin-embeddable`
- Change `androidx.compose.compiler:compiler-hosted` to `org.jetbrains.kotlin:kotlin-compose-compiler-plugin`

What's next

- See Google's announcement about the Compose compiler moving to the Kotlin repository.
- If you are using Jetpack Compose to build an Android app, check out our guide on how to make it multiplatform.

Compose compiler options DSL

The Compose compiler Gradle plugin offers a DSL for various compiler options. You can use it to configure the compiler in the `composeCompiler {}` block of the `build.gradle.kts` file for the module you're applying the plugin to.

There are two kinds of options you can specify:

- General compiler settings, which can be disabled or enabled as needed in any given project.
- Feature flags that enable or disable new and experimental features, which should eventually become part of the baseline.

You can find the list of available general settings and the list of supported feature flags in the Compose compiler Gradle plugin API reference.

Here's an example configuration:

```
composeCompiler {
    includeSourceInformation = true

    featureFlags = setOf(
        ComposeFeatureFlag.StrongSkipping.disabled(),
        ComposeFeatureFlag.OptimizeNonSkippingGroups
    )
}
```

The Gradle plugin provides defaults for several Compose compiler options that were only specified manually before Kotlin 2.0. If you have any of them set up with `freeCompilerArgs`, for example, Gradle reports a duplicate options error.

Purpose and use of feature flags

Feature flags are organized into a separate set of options to minimize changes to top-level properties as new flags are continuously rolled out and deprecated.

To enable a feature flag that is disabled by default, specify it in the set, for example:

```
featureFlags = setOf(ComposeFeatureFlag.OptimizeNonSkippingGroups)
```

To disable a feature flag that is enabled by default, call the `disabled()` function on it, for example:

```
featureFlags = setOf(ComposeFeatureFlag.StrongSkipping.disabled())
```

If you are configuring the Compose compiler directly, use the following syntax to pass feature flags to it:

```
-P plugin:androidx.compose.compiler.plugins.kotlin:featureFlag=<flag name>
```

See the list of supported feature flags in the Compose compiler Gradle plugin API reference.

Kotlin Symbol Processing API

Kotlin Symbol Processing (KSP) is a source code generation framework for Kotlin. With the KSP API, you can create processors that generate code based on annotations in your source code.

KSP aims to simplify the creation of lightweight compiler plugins. Its well-defined API hides compiler changes, so you don't need to spend much effort maintaining your processors. However, this approach comes with trade-offs. For example, KSP-based processors can't examine expressions or statements, and they can't modify the source code.

Typical use cases for KSP-based plugins include:

- Dependency injection (Dagger)
- Serialization (Moshi)
- Database management (Room)

To learn how to create your first KSP-based processor, see the KSP quickstart.

Overview

The KSP API processes Kotlin programs idiomatically. KSP understands Kotlin-specific features, such as extension functions, declaration-site variance, and local functions. It also models types explicitly and provides basic type checking, such as equivalence and assign-compatibility.

The API models Kotlin program structures at the symbol level according to Kotlin grammar. When KSP-based plugins process source programs, constructs like classes, class members, functions, and associated parameters are accessible for the processors, while things like `if` blocks and `for` loops are not.

Conceptually, KSP is similar to KType in Kotlin reflection. The API allows processors to navigate from class declarations to corresponding types with specific type arguments and vice versa. You can also substitute type arguments, specify variances, apply star projections, and mark nullabilities of types.

Another way to think of KSP is as a preprocessor framework of Kotlin programs. By considering KSP-based plugins as *symbol processors*, or simply *processors*, the data flow in a compilation can be described in the following steps:

1. Processors read and analyze source programs and resources.
2. Processors generate code or other forms of output.

3. The Kotlin compiler compiles the source programs together with the generated code.

Unlike a full-fledged compiler plugin, processors cannot modify the code. A compiler plugin that changes language semantics can sometimes be very confusing. KSP avoids that by treating the source programs as read-only.

You can also get an overview of KSP in this video:

Watch video online.

How KSP looks at source files

Most processors navigate through the various program structures of the input source code. Before diving into usage of the API, let's see at how a file might look from KSP's point of view:

```
KSFile
  packageName: KSName
  fileName: String
  annotations: List<KSAnnotation>  (File annotations)
  declarations: List<KSDeclaration>
    KSClassDeclaration // class, interface, object
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      classKind: ClassKind
      primaryConstructor: KSFunctionDeclaration
      superTypes: List<KSTypeReference>
      // contains inner classes, member functions, properties, etc.
      declarations: List<KSDeclaration>
    KSFunctionDeclaration // top level function
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      functionKind: FunctionKind
      extensionReceiver: KSTypeReference?
      returnType: KSTypeReference
      parameters: List<KSValueParameter>
      // contains local classes, local functions, local variables, etc.
      declarations: List<KSDeclaration>
    KSPropertyDeclaration // global variable
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      extensionReceiver: KSTypeReference?
      type: KSTypeReference
      getter: KSPropertyGetter
        returnType: KSTypeReference
      setter: KSPropertySetter
        parameter: KSValueParameter
```

This view lists common things that are declared in the file: classes, functions, properties, and so on.

SymbolProcessorProvider: the entry point

KSP expects an implementation of the `SymbolProcessorProvider` interface to instantiate `SymbolProcessor`:

```
interface SymbolProcessorProvider {
    fun create(environment: SymbolProcessorEnvironment): SymbolProcessor
}
```

While `SymbolProcessor` is defined as:

```
interface SymbolProcessor {
    fun process(resolver: Resolver): List<KSAnnotated> // Let's focus on this
    fun finish() {}
    fun onError() {}
}
```

A `Resolver` provides `SymbolProcessor` with access to compiler details such as symbols. A processor that finds all top-level functions and non-local functions in top-level classes might look something like the following:

```
class HelloFunctionFinderProcessor : SymbolProcessor() {
    // ...
    val functions = mutableListOf<KSFunctionDeclaration>()
    val visitor = FindFunctionsVisitor()

    override fun process(resolver: Resolver) {
        resolver.getAllFiles().forEach { it.accept(visitor, Unit) }
    }

    inner class FindFunctionsVisitor : KSVisitorVoid() {
        override fun visitClassDeclaration(classDeclaration: KSClassDeclaration, data: Unit) {
            classDeclaration.getDeclaredFunctions().forEach { it.accept(this, Unit) }
        }

        override fun visitFunctionDeclaration(function: KSFunctionDeclaration, data: Unit) {
            functions.add(function)
        }

        override fun visitFile(file: KSFile, data: Unit) {
            file.declarations.forEach { it.accept(this, Unit) }
        }
    }
    // ...

    class Provider : SymbolProcessorProvider {
        override fun create(environment: SymbolProcessorEnvironment): SymbolProcessor = TODO()
    }
}
```

Resources

- Quickstart
- Examples
- How KSP models Kotlin code
- Reference for Java annotation processor authors
- Incremental processing notes

- Multiple round processing notes
- KSP on multiplatform projects
- Running KSP from command line
- FAQ

Supported libraries

The table includes a list of popular libraries on Android and their various stages of support for KSP:

Library	Status
Room	Officially supported
Moshi	Officially supported
RxHttp	Officially supported
Kotshi	Officially supported
Lyricist	Officially supported
Lich SavedState	Officially supported
gRPC Dekorator	Officially supported
EasyAdapter	Officially supported
Koin Annotations	Officially supported
Glide	Officially supported
Micronaut	Officially supported
Epoxy	Officially supported
Paris	Officially supported
Auto Dagger	Officially supported
SealedX	Officially supported
Ktorfit	Officially supported

Library	Status
Mockative	Officially supported
Kotest	Officially supported
DeeplinkDispatch	Supported via airbnb/DeepLinkDispatch#323
Dagger	Alpha
Motif	Alpha
Hilt	In progress
Auto Factory	Not yet supported

Getting started with KSP

In this guide, you will learn:

- How to add KSP-based annotation processors to your project.
- How to create your own annotation processor with the KSP API.
- Where to find the code generated by the processor.

Add a KSP-based processor to your project

To use an external processor in your project, add KSP to the `plugins {}` block in your `build.gradle(.kts)` file. If the processor is only needed in a specific module, add it to that module's `build.gradle(.kts)` file instead:

Kotlin

```
// build.gradle.kts

plugins {
    kotlin("jvm") version "2.4.20"
    id("com.google.devtools.ksp") version "2.3.10"
}
```

Groovy

```
// build.gradle

plugins {
    id 'org.jetbrains.kotlin.jvm' version '2.4.20'
```

```
    id 'com.google.devtools.ksp' version '2.3.10'
}
```

To find the latest version of KSP, check the GitHub Releases.

In the top-level `dependencies {}` block, add the processor you want to use. This example uses Moshi, but the approach is the same for other processors:

Kotlin

```
// build.gradle.kts

dependencies {
    ksp("com.squareup.moshi:moshi-kotlin-codegen:1.15.2")
}
```

Groovy

```
// build.gradle

dependencies {
    ksp 'com.squareup.moshi:moshi-kotlin-codegen:1.15.2'
}
```

The `ksp(...)` configuration applies the processor only to application sources. To process test sources, add the processor with the `kspTest(...)` configuration.

The `ksp(...)` configuration is available only in single-platform projects. To learn how to configure processors for individual Kotlin Multiplatform targets and compilations, see KSP with Kotlin Multiplatform.

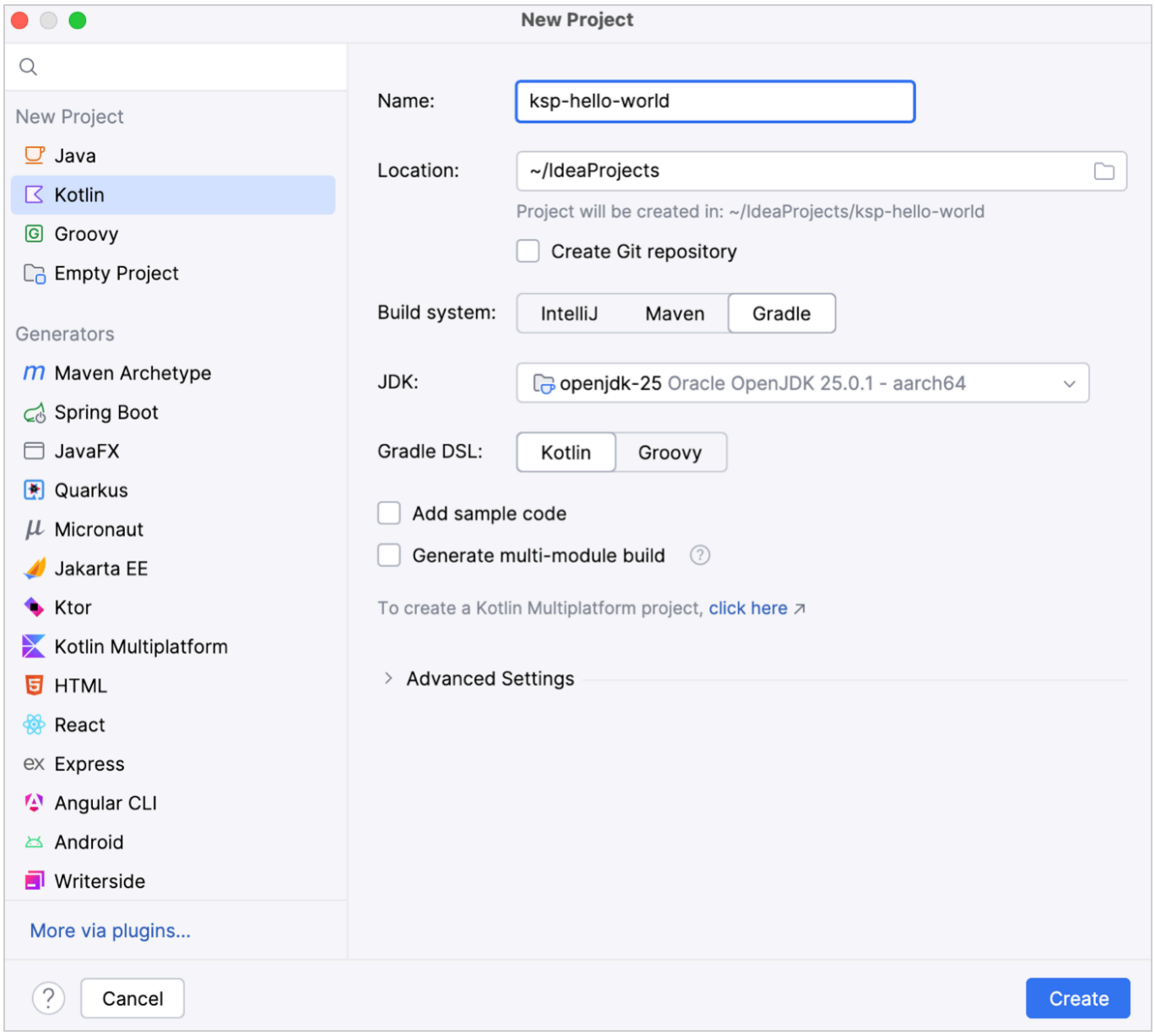
Create your own processor

By following these steps you will create a simple annotation processor that will generate a `helloWorld()` function. While not very useful in practice, it demonstrates the basics of creating your own processors and annotations.

Add KSP to the project

Create a new Kotlin project and add the KSP plugin:

1. In IntelliJ IDEA, select **File** | **New** | **Project**.
2. In the list on the left, choose **Kotlin**.
3. Choose **Gradle** as the build system and click **Create**.



Creating a new project

4. Add the KSP plugin to the `build.gradle(.kts)` file:

Kotlin

```kotlin
// build.gradle.kts

plugins {
    kotlin("jvm") version "2.4.20"
    id("com.google.devtools.ksp") version "2.3.10" apply false
}
```

Groovy

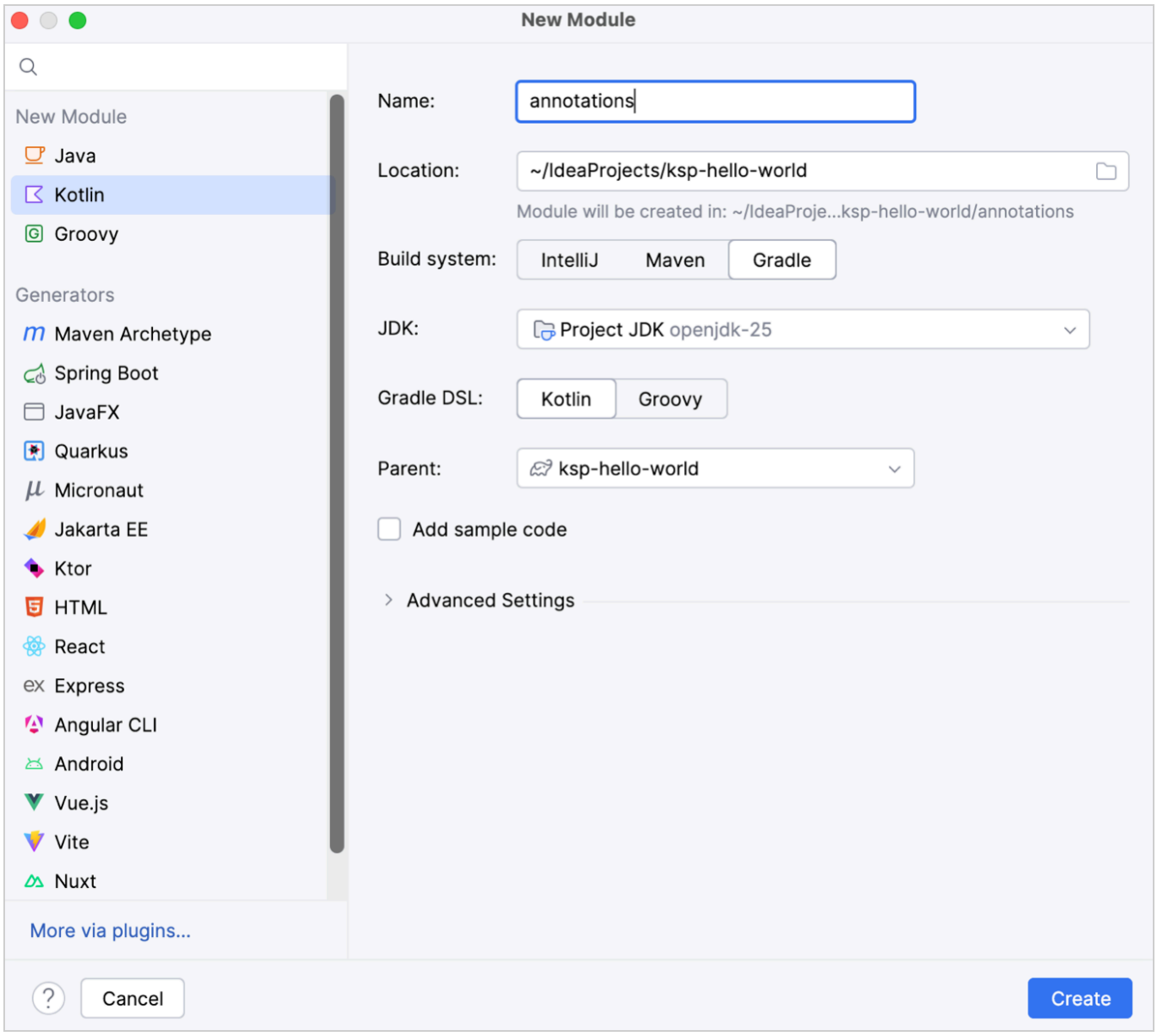
```
// build.gradle

plugins {
    id 'org.jetbrains.kotlin.jvm' version '2.4.20'
    id 'com.google.devtools.ksp' version '2.3.10' apply false
}
```

Create an annotation

Create a new module at the root of the project and declare an annotation:

1. Select **File** | **New** | **Module**.
2. In the list on the left, select **Kotlin**.
3. Specify the following fields and click **create**:
   - **Name**: annotations
   - **Build system**: Gradle



Creating a new module

4. In the module, create a `HelloWorldAnnotation.kt` file and declare an annotation called `HelloWorldAnnotation`:

```
// annotations/src/main/kotlin/com/example/annotations/HelloWorldAnnotation.kt

package com.example.annotations

annotation class HelloWorldAnnotation
```

Create and register a processor

1. Create another module at the root of the project called **processor**.
2. Add the KSP API and the annotation you declared as dependencies in the module's `build.gradle(.kts)` file:

Kotlin

```kotlin
// processor/build.gradle.kts

plugins {
    kotlin("jvm")
}

dependencies {
    implementation(project(":annotations"))
    implementation("com.google.devtools.ksp:symbol-processing-api:2.3.6")
}
```

Groovy

```groovy
// processor/build.gradle

plugins {
    id 'org.jetbrains.kotlin.jvm'
}

dependencies {
    implementation project ':annotations'
    implementation 'com.google.devtools.ksp:symbol-processing-api:2.3.6'
}
```

3. In the processor module, create a new `HelloWorldProcessor.kt` file and add the following code:

```kotlin
// processor/src/main/kotlin/HelloWorldProcessor.kt

class HelloWorldProcessor(val codeGenerator: CodeGenerator) : SymbolProcessor {
    // 1️⃣ process() function
    override fun process(resolver: Resolver): List<KSAnnotated> {
        resolver
            .getSymbolsWithAnnotation("com.example.annotations.HelloWorldAnnotation")
            .filter { it.validate() }
            .filterIsInstance<KSFunctionDeclaration>()
            .forEach { it.accept(HelloWorldVisitor(), Unit) }

        return emptyList()
    }

    // 2️⃣ Visitor
    inner class HelloWorldVisitor : KSVisitorVoid() {
        override fun visitFunctionDeclaration(function: KSFunctionDeclaration, data: Unit) {
            createNewFileFrom(function).use { file ->
                file.write(
                    """
                        fun helloWorld(): Unit {
                            println("Hello world from function generated by KSP")
                        }
                    """.trimIndent()
                )
            }
        }
    }

    // 3️⃣ createNewFileFrom() function
    private fun createNewFileFrom(function: KSFunctionDeclaration): OutputStream {
        return codeGenerator.createNewFile(
            dependencies = createDependencyOn(function),
            packageName = "",
            fileName = "GeneratedHelloWorld"
        )
    }

    // 3️⃣ createDependencyOn() function
    private fun createDependencyOn(function: KSFunctionDeclaration): Dependencies {
        return Dependencies(aggregating = false, function.containingFile!!)
    }
}

// Utility function for writing string to OutputStream
fun OutputStream.write(string: String): Unit {
    this.write(string.toByteArray())
}
```

Add the imports that are suggested by the IDE. Make sure to import the `Resolver` and `Dependencies` classes from `com.google.devtools.ksp.processing`. Alternatively, copy these lines at the top of `HelloWorldProcessor.kt`:

```
// processor/src/main/kotlin/HelloWorldProcessor.kt

import com.google.devtools.ksp.processing.CodeGenerator
import com.google.devtools.ksp.processing.Dependencies
import com.google.devtools.ksp.processing.Resolver
import com.google.devtools.ksp.processing.SymbolProcessor
import com.google.devtools.ksp.symbol.KSAnnotated
import com.google.devtools.ksp.symbol.KSFunctionDeclaration
import com.google.devtools.ksp.symbol.KSVisitorVoid
import com.google.devtools.ksp.validate
import java.io.OutputStream
```

Let's go through the code:

- 1️⃣ The `process()` function contains the main logic of the processor. It gets all symbols annotated with `HelloWorldAnnotation` and calls `HelloWorldVisitor` for each one.

  The `process()` function returns a list of unprocessed symbols to process in a later round. In this example, it safely returns `emptyList()`. For more information, see Multiple round processing.

- 2️⃣ Processors traverse KSP's view of the Kotlin abstract syntax tree (AST) using visitors. Inside the `HelloWorldPocessor` class, the `HelloWorldVisitor` class is the visitor. Since the `HelloWorldAnnotation` is only used on a function, only `visitFunctionDeclaration()` is overridden.

  > ℹ️ `KSVisitorVoid` is one of the visitor classes provided by KSP that you can override and adapt. You can also create your own visitor by implementing the `KSVisitor<D, R>` interface.

- 3️⃣ `createNewFileFrom()` creates the file where KSP generates code. `createDependencyOn()` makes the output file depend on the source file where the annotation is used.

  > ℹ️ To learn more about how KSP creates and manages files, visit the source code for the `CodeGenerator` interface

4. Create a `HelloWorldProcessorProvider.kt` file. In it, declare a `HelloWorldProcessorProvider` class which inherits from `SymbolProcessorProvider`:

```
// processor/src/main/kotlin/HelloWorldProcessorProvider.kt

import com.google.devtools.ksp.processing.SymbolProcessor
import com.google.devtools.ksp.processing.SymbolProcessorEnvironment
import com.google.devtools.ksp.processing.SymbolProcessorProvider

class HelloWorldProcessorProvider : SymbolProcessorProvider {
    override fun create(environment: SymbolProcessorEnvironment): SymbolProcessor {
        return HelloWorldProcessor(environment.codeGenerator)
    }
}
```

5. Register the processor provider. In the `resources/META-INF/services` directory, create a `com.google.devtools.ksp.processing.SymbolProcessorProvider` file and add the provider's fully qualified name:

```
## processor/src/main/resources/META-
INF/services/com.google.devtools.ksp.processing.SymbolProcessorProvider

HelloWorldProcessorProvider
```

Use your processor

Now you are ready to test your processor. Follow these steps to create a client module and have your processor generate code based on an annotated element:

1. Create a module called `app` at the root of your project.
2. In the module's `build.gradle(.kts)` file:
   - Add the KSP plugin to the `plugins {}` block.
   - Add your processor and annotation to the `dependencies {}` block.

   For example:

Kotlin

```
// app/build.gradle.kts

plugins {
    kotlin("jvm")
    id("com.google.devtools.ksp")
}

dependencies {
    implementation(project(":annotations"))
    ksp(project(":processor"))
}
```

Groovy

```
// app/build.gradle

plugins {
    id 'com.google.devtools.ksp'
}

dependencies {
    implementation project (':annotations')
    ksp project (':processor')
}
```

3. In the project-level `settings.gradle(.kts)` file, ensure that all the submodules were automatically included:

Kotlin

```
// settings.gradle.kts

include("annotations")
include("app")
include("processor")
```

Groovy

```
// settings.gradle

include 'processor'
include 'annotations'
include 'app'
```

4. In the `app` module, create a `Main.kt` file and add the following code:

```
// app/src/main/kotlin/Main.kt

import com.example.annotations.HelloWorldAnnotation

@HelloWorldAnnotation
fun main() {
    helloWorld()
}
```

> The `main()` function calls `helloWorld()`, even though this function does not exist yet. Your IDE highlights `helloWorld()` as an undefined reference. This is expected: KSP generates the `helloWorld()` function when you build and run the project.

5. Run the program. You see the output of the `helloWorld()` function in your console:

```
Hello world from function generated by KSP
```

KSP generates code in the `GeneratedHelloWorld.kt` file:

```
app/build/generated/ksp/main/kotlin/GeneratedHelloWorld.kt
```

Explore the project structure

Your project's final file structure should look like this:

```
.
├── app
│   ├── build.gradle.kts
│   └── src
│       └── main
│           └── kotlin
│               └── Main.kt
├── annotations
│   ├── build.gradle.kts
│   └── src
│       └── main
│           └── kotlin
│                               └── com
│                       └── example
│                                               └── annotations
│                                                       └── HelloWorldAnnotation.kt
├── processor
│   ├── build.gradle.kts
│   └── src
│       └── main
│           ├── kotlin
│           │   ├── HelloWorldProcessor.kt
│           │   └── HelloWorldProcessorProvider.kt
│           └── resources/META-INF/services
│                               └── com.google.devtools.ksp.processing.SymbolProcessorProvider
├── build.gradle.kts
└── settings.gradle.kts
```

You might have additional files and directories.

What's next?

- Explore the full code for this example in the KSP repository.
- Find more complex, real-world examples in the KSP repository.
- Browse the list of KSP supported libraries.

Migrate from kapt to KSP

In this guide, you will learn how to migrate your annotation processors from kapt to KSP so that your project can take full advantage of Kotlin features and improve build performance.

kapt (Kotlin Annotation Processing Tool) is a useful tool that lets you use Java annotation processors in Kotlin. It works by translating Kotlin source code into Java "stubs" and then running the annotation processors on those stubs. However, this process is expensive, significantly increases build time, and loses some Kotlin-specific features in translation.

In contrast, KSP (Kotlin Symbol Processing) is an alternative to kapt designed specifically for Kotlin. KSP understands all Kotlin features and analyzes the source code directly, reducing build time.

Before you begin, check whether the processors in your project support KSP. See the list of supported libraries or consult their documentation.

KSP and kapt can run alongside each other, so you can migrate your project in stages, one library or module at a time.

Add the KSP plugin to your project

Add KSP to the `plugins {}` block in the project-level `build.gradle(.kts)` file:

Kotlin

```
plugins {
    id("com.google.devtools.ksp") version "2.3.10" apply false
}
```

Groovy

```
plugins {
    id 'com.google.devtools.ksp' version '2.3.10' apply false
}
```

To find the latest version of KSP, check the GitHub Releases.

Update your processor

Find the module that uses the processor you want to migrate. In the module's `build.gradle(.kts)` file:

1. Add KSP to the `plugins {}` block:

   Kotlin

   ```
   plugins {
       id("com.google.devtools.ksp")
   }
   ```

   Groovy

   ```
   plugins {
       id 'com.google.devtools.ksp'
   }
   ```

2. In the `dependencies {}` block, replace `kapt` with `ksp`:

Kotlin

```
dependencies {
    implementation("com.google.dagger:dagger:2.48")
    // kapt("com.google.dagger:dagger-compiler:2.48")

    // KSP processor dependency:
    ksp("com.google.dagger:dagger-compiler:2.48")
}
```

Groovy

```
dependencies {
    implementation 'com.google.dagger:dagger:2.48'
    // kapt 'com.google.dagger:dagger-compiler:2.48'

    // KSP processor dependency:
    ksp 'com.google.dagger:dagger-compiler:2.48'
}
```

For most libraries, this replacement is enough. Check each library's documentation to see if you need to make any additional changes.

Remove the kapt plugin

After migrating all your processors to KSP, you can safely remove the kapt plugin from all build files:

Kotlin

```
plugins {
   // Delete this line:
    id("org.jetbrains.kotlin.kapt")
}
```

Groovy

```
plugins {
    // Delete this line:
    id 'org.jetbrains.kotlin.kapt'
}
```

Delete any leftover kapt configurations if you have them.

What's next?

- Learn how to make your own KSP-based annotation processor in Getting started with KSP.
- Explore example projects that use KSP in the KSP repository.

- Read more about KSP in the Overview.

KSP examples

Get all member functions

```
fun KSClassDeclaration.getDeclaredFunctions(): Sequence<KSFunctionDeclaration> =
    declarations.filterIsInstance<KSFunctionDeclaration>()
```

Check whether a class or function is local

```
fun KSDeclaration.isLocal(): Boolean =
    parentDeclaration ≠ null && parentDeclaration !is KSClassDeclaration
```

Find the actual class or interface declaration that the type alias points to

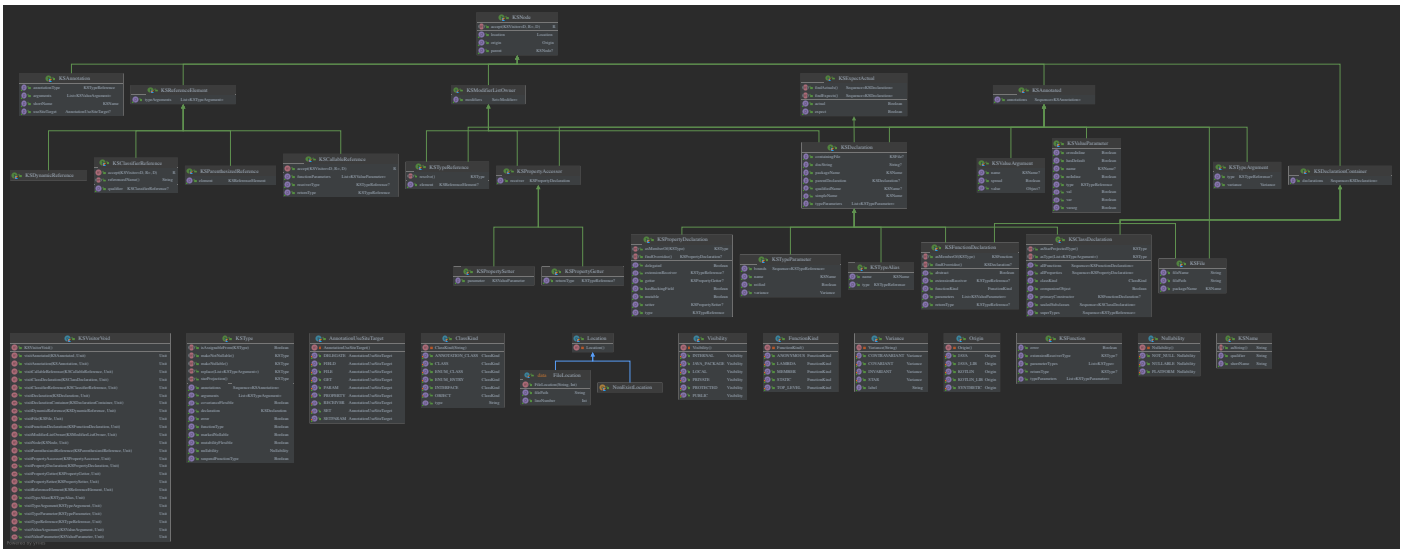
```
fun KSTypeAlias.findActualType(): KSClassDeclaration {
    val resolvedType = this.type.resolve().declaration
    return if (resolvedType is KSTypeAlias) {
        resolvedType.findActualType()
    } else {
        resolvedType as KSClassDeclaration
    }
}
```

Collect suppressed names in a file annotation

```
// @file:kotlin.Suppress("Example1", "Example2")
fun KSFile.suppressedNames(): Sequence<String> = annotations
    .filter {
        it.shortName.asString() == "Suppress" &&
        it.annotationType.resolve().declaration.qualifiedName?.asString() == "kotlin.Suppress"
    }.flatMap {
        it.arguments.flatMap {
            (it.value as Array<String>).toList()
        }
    }
```

How KSP models Kotlin code

You can find the API definition in the KSP GitHub repository. The diagram shows an overview of how Kotlin is modeled in KSP:



class diagram

> See the full-sized diagram.

Type and resolution

The resolution takes most of the cost of the underlying API implementation. So type references are designed to be resolved by processors explicitly (with a few exceptions). When a *type* (such as `KSFunctionDeclaration.returnType` or `KSAnnotation.annotationType`) is referenced, it is always a `KSTypeReference`, which is a `KSReferenceElement` with annotations and modifiers.

```
interface KSFunctionDeclaration : ... {
  val returnType: KSTypeReference?
  // ...
}

interface KSTypeReference : KSAnnotated, KSModifierListOwner {
  val type: KSReferenceElement
}
```

A `KSTypeReference` can be resolved to a `KSType`, which refers to a type in Kotlin's type system.

A `KSTypeReference` has a `KSReferenceElement`, which models Kotlin's program structure: namely, how the reference is written. It corresponds to the `type` element in Kotlin's grammar.

A `KSReferenceElement` can be a `KSClassifierReference` or `KSCallableReference`, which contains a lot of useful information without the need for resolution. For example, `KSClassifierReference` has `referencedName`, while `KSCallableReference` has `receiverType`, `functionArguments`, and `returnType`.

If the original declaration referenced by a `KSTypeReference` is needed, it can usually be found by resolving to `KSType` and accessing through `KSType.declaration`. Moving from where a type is mentioned to where its class is defined looks like this:

```
val ksType: KSType = ksTypeReference.resolve()
val ksDeclaration: KSDeclaration = ksType.declaration
```

Type resolution is costly and therefore has explicit form. Some of the information obtained from resolution is already available in `KSReferenceElement`. For example, `KSClassifierReference.referencedName` can filter out

a lot of elements that are not interesting. You should resolve type only if you need specific information from `KSDeclaration` or `KSType`.

`KSTypeReference` pointing to a function type has most of its information in its element. Although it can be resolved to the family of `Function0`, `Function1`, and so on, these resolutions don't bring any more information than `KSCallableReference`. One use case for resolving function type references is dealing with the identity of the function's prototype.

Java annotation processing to KSP reference

Program elements

Java	Closest facility in KSP	Notes
`AnnotationMirror`	`KSAnnotation`	
`AnnotationValue`	`KSValueArguments`	
`Element`	`KSDeclaration` / `KSDeclarationContainer`	
`ExecutableElement`	`KSFunctionDeclaration`	
`PackageElement`	`KSFile`	KSP doesn't model packages as program elements
`Parameterizable`	`KSDeclaration`	
`QualifiedNameable`	`KSDeclaration`	
`TypeElement`	`KSClassDeclaration`	
`TypeParameterElement`	`KSTypeParameter`	
`VariableElement`	`KSValueParameter` / `KSPropertyDeclaration`	

Types

KSP requires explicit type resolution, so some functionalities in Java can only be carried out by `KSType` and the corresponding elements before resolution.

Java	Closest facility in KSP	Notes
`ArrayType`	`KSBuiltIns.arrayType`	
`DeclaredType`	`KSType` / `KSClassifierReference`	
`ErrorType`	`KSType.isError`	
`ExecutableType`	`KSType` / `KSCallableReference`	
`IntersectionType`	`KSType` / `KSTypeParameter`	
`NoType`	`KSType.isError`	N/A in KSP
`NullType`		N/A in KSP
`PrimitiveType`	`KSBuiltIns`	Not exactly same as primitive type in Java
`ReferenceType`	`KSTypeReference`	
`TypeMirror`	`KSType`	
`TypeVariable`	`KSTypeParameter`	
`UnionType`	N/A	Kotlin has only one type per catch block. `UnionType` is also not observable by even Java annotation processors
`WildcardType`	`KSType` / `KSTypeArgument`	

Misc

Java	Closest facility in KSP	Notes
`Name`	`KSName`	
`ElementKind`	`ClassKind` / `Function Kind`	
`Modifier`	`Modifier`	
`NestingKind`	`ClassKind` / `Function Kind`	
`AnnotationValueVisitor`		
`ElementVisitor`	`KSVisitor`	
`AnnotatedConstruct`	`KSAnnotated`	
`TypeVisitor`		
`TypeKind`	`KSBuiltIns`	Some can be found in builtins, otherwise check `KSClassDeclaration` for `DeclaredType`
`ElementFilter`	`Collection.filterIsInstance`	
`ElementKindVisitor`	`KSVisitor`	
`ElementScanner`	`KSTopDownVisitor`	
`SimpleAnnotationValueVisitor`		Not needed in KSP
`SimpleElementVisitor`	`KSVisitor`	
`SimpleTypeVisitor`		
`TypeKindVisitor`		

Java	Closest facility in KSP	Notes
`Types`	`Resolver` / `utils`	Some of the `utils` are also integrated into symbol interfaces
`Elements`	`Resolver` / `utils`	

Details

See how functionalities of Java annotation processing API can be carried out by KSP.

AnnotationMirror

Java	KSP equivalent
`getAnnotationType`	`ksAnnotation.annotationType`
`getElementValues`	`ksAnnotation.arguments`

AnnotationValue

Java	KSP equivalent
`getValue`	`ksValueArgument.value`

Element

Java	KSP equivalent
`asType`	`ksClassDeclaration.asType(...)` is available for `KSClassDeclaration` only. Type arguments need to be supplied.
`getAnnotation`	To be implemented
`getAnnotationMirrors`	`ksDeclaration.annotations`
`getEnclosedElements`	`ksDeclarationContainer.declarations`
`getEnclosingElements`	`ksDeclaration.parentDeclaration`

Java	KSP equivalent
`getKind`	Type check and cast following `ClassKind` or `FunctionKind`
`getModifiers`	`ksDeclaration.modifiers`
`getSimpleName`	`ksDeclaration.simpleName`

ExecutableElement

Java	KSP equivalent
`getDefaultValue`	To be implemented
`getParameters`	`ksFunctionDeclaration.parameters`
`getReceiverType`	`ksFunctionDeclaration.parentDeclaration`
`getReturnType`	`ksFunctionDeclaration.returnType`
`getSimpleName`	`ksFunctionDeclaration.simpleName`
`getThrownTypes`	Not needed in Kotlin
`getTypeParameters`	`ksFunctionDeclaration.typeParameters`
`isDefault`	Check whether parent declaration is an interface or not
`isVarArgs`	`ksFunctionDeclaration.parameters.any { it.isVarArg }`

Parameterizable

Java	KSP equivalent
`getTypeParameters`	`ksFunctionDeclaration.typeParameters`

QualifiedNameable

Java	KSP equivalent
`getQualifiedName`	`ksDeclaration.qualifiedName`

TypeElement

Java	KSP equivalent
`getEnclosedElements`	`ksClassDeclaration.declarations`
`getEnclosingElement`	`ksClassDeclaration.parentDeclaration`

`getInterfaces`	`// Should be able to do without resolution`
	`ksClassDeclaration.superTypes`
	`.map { it.resolve() }`
	`.filter { (it?.declaration as? KSClassDeclaration)?.classKind == ClassKind.INTERFACE }`

`getNestingKind`	Check `KSClassDeclaration.parentDeclaration` and `inner` modifier
`getQualifiedName`	`ksClassDeclaration.qualifiedName`
`getSimpleName`	`ksClassDeclaration.simpleName`

`getSuperclass`	`// Should be able to do without resolution`
	`ksClassDeclaration.superTypes`
	`.map { it.resolve() }`
	`.filter { (it?.declaration as? KSClassDeclaration)?.classKind == ClassKind.CLASS }`

`getTypeParameters`	`ksClassDeclaration.typeParameters`

TypeParameterElement

Java	KSP equivalent
`getBounds`	`ksTypeParameter.bounds`
`getEnclosingElement`	`ksTypeParameter.parentDeclaration`
`getGenericElement`	`ksTypeParameter.parentDeclaration`

VariableElement

Java	KSP equivalent
`getConstantValue`	To be implemented
`getEnclosingElement`	`ksValueParameter.parentDeclaration`
`getSimpleName`	`ksValueParameter.simpleName`

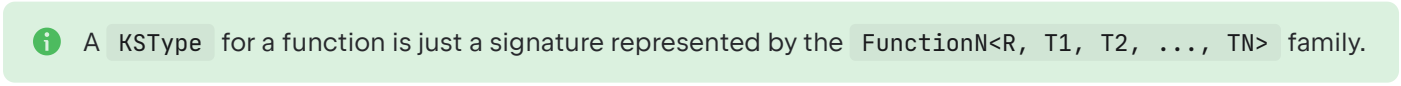
ArrayType

Java	KSP equivalent
`getComponentType`	`ksType.arguments.first()`

DeclaredType

Java	KSP equivalent
`asElement`	`ksType.declaration`
`getEnclosingType`	`ksType.declaration.parentDeclaration`
`getTypeArguments`	`ksType.arguments`

ExecutableType


A `KSType` for a function is just a signature represented by the `FunctionN<R, T1, T2, ..., TN>` family.

Java	KSP equivalent
`getParameterTypes`	`ksType.declaration.typeParameters`, `ksFunctionDeclaration.parameters.map { it.type }`
`getReceiverType`	`ksFunctionDeclaration.parentDeclaration.asType(...)`
`getReturnType`	`ksType.declaration.typeParameters.last()`
`getThrownTypes`	Not needed in Kotlin

Java	KSP equivalent
`getTypeVariables`	`ksFunctionDeclaration.typeParameters`

IntersectionType

Java	KSP equivalent
`getBounds`	`ksTypeParameter.bounds`

TypeMirror

Java	KSP equivalent
`getKind`	Compare with types in `KSBuiltIns` for primitive types, `Unit` type, otherwise declared types

TypeVariable

Java	KSP equivalent
`asElement`	`ksType.declaration`
`getLowerBound`	To be decided. Only needed if capture is provided and explicit bound checking is needed.
`getUpperBound`	`ksTypeParameter.bounds`

WildcardType

Java	KSP equivalent
`getExtendsBound`	`if (ksTypeArgument.variance == Variance.COVARIANT) ksTypeArgument.type else null`
`getSuperBound`	`if (ksTypeArgument.variance == Variance.CONTRAVARIANT) ksTypeArgument.type else null`

Elements

Java	KSP equivalent
`getAllAnnotationMirrors`	`KSDeclarations.annotations`
`getAllMembers`	`getAllFunctions` , `getAllProperties` is to be implemented
`getBinaryName`	To be decided, see Java Specification
`getConstantExpression`	There is constant value, not expression
`getDocComment`	To be implemented
`getElementValuesWithDefaults`	To be implemented
`getName`	`resolver.getKSNameFromString`
`getPackageElement`	Package not supported, while package information can be retrieved. Operation on package is not possible for KSP
`getPackageOf`	Package not supported
`getTypeElement`	`Resolver.getClassDeclarationByName`
`hides`	To be implemented
`isDeprecated`	`KsDeclaration.annotations.any { it.annotationType.resolve()!!.declaration.qualifiedName!!.asString() == Deprecated::class.qualifiedName }`
`overrides`	`KSFunctionDeclaration.overrides` / `KSPropertyDeclaration.overrides` (member function of respective class)
`printElements`	KSP has basic `toString()` implementation on most classes

Types

Java	KSP equivalent
`asElement`	`ksType.declaration`
`asMemberOf`	`resolver.asMemberOf`
`boxedClass`	Not needed
`capture`	To be decided
`contains`	`KSType.isAssignableFrom`
`directSuperTypes`	`(ksType.declaration as KSClassDeclaration).superTypes`
`erasure`	`ksType.starProjection()`
`getArrayType`	`ksBuiltIns.arrayType.replace(...)`
`getDeclaredType`	`ksClassDeclaration.asType`
`getNoType`	`ksBuiltIns.nothingType` / `null`
`getNullType`	Depending on the context, `KSType.markNullable` could be useful
`getPrimitiveType`	Not needed, check for `KSBuiltins`
`getWildcardType`	Use `Variance` in places expecting `KSTypeArgument`
`isAssignable`	`ksType.isAssignableFrom`
`isSameType`	`ksType.equals`
`isSubsignature`	`functionTypeA == functionTypeB` / `functionTypeA == functionTypeB.starProjection()`
`isSubtype`	`ksType.isAssignableFrom`
`unboxedType`	Not needed

Incremental processing

KSP supports incremental processing: KSP reprocesses a file only when one or more of its dependencies change. This avoids unnecessary reprocessing and therefore reduces compilation time.

Incremental processing is enabled by default. You can disable it when troubleshooting or when you need to force a full rebuild. To disable it, add the following line to your `gradle.properties` file:

```
ksp.incremental=false
```

Dirty files

A file is considered *dirty* (needing to be reprocessed) if it is either directly modified by a developer or indirectly affected by changes in other dirty files.

To determine which sources are dirty, KSP relies on processors, which associate generated outputs with their corresponding input sources. KSP uses these associations to identify the sources that must be reprocessed when a change occurs.

KSP requires only a minimal set of root sources. Processors use these sources as entry points to navigate the code structure.

A root source is a source file whose symbols are obtained directly from any of the following methods:

- `Resolver.getAllFiles()`
- `Resolver.getSymbolsWithAnnotation()`
- `Resolver.getClassDeclarationByName()`
- `Resolver.getDeclarationsFromPackage()`

A processor can obtain additional symbols from other source files by resolving information from a root source. KSP automatically tracks these dependencies.

When generating an output, the processor must declare the root sources that contribute to that output. KSP uses those root sources and their tracked dependencies to determine when the output needs to be regenerated.

Use the `CodeGenerator` interface to create output files and associate inputs with outputs. For more information, see `CodeGenerator.kt` in the source code.

Aggregating and isolating outputs

KSP classifies generated outputs into two types: aggregating and isolating.

Unlike Gradle annotation processing, KSP applies the categorization to individual outputs rather than entire processors.

Aggregating

Aggregating outputs can be affected by changes in any source file, except for removals that do not impact other files.

Any input change triggers a rebuild of all aggregating outputs and reprocessing of all corresponding registered, new, or modified source files.

For example, an output that collects all symbols with a particular annotation is aggregating.

Isolating

Isolating outputs depend only on their specified sources.

Changes to other sources do not affect the output. Multiple source files can be associated with a single output.

For example, a generated class that is dedicated to an interface it implements is isolating.

Dirtiness propagation

KSP propagates dirtiness in the following ways:

1. By **resolution tracing**: Type resolution is the only way to traverse from one file to another. When a processor resolves a type reference (explicitly or implicitly), KSP considers dependencies between the file containing the reference and any file that defines a symbol affecting that resolution. As a result, a change in a resolved symbol may mark the referencing file as dirty.
2. By **input-output correspondence**: If a source file is changed or affected, all other source files that share generated outputs with it are also marked as affected. This groups related files into equivalence classes based on shared outputs.

> Rules (1) and (2) can trigger each other repeatedly. For example, rule (1) can trigger rule (2), which can then trigger rule (1) again.

Implementation

Dependencies are determined by the many-to-many relationship between input and output files.

This is how KSP determines which files need to be reprocessed:

- If an input file is changed, it will always be reprocessed.

  Why? If an input is changed, new information can be introduced. Processors need to run again with the input.

- If an input file is changed and is associated with an output, then all other input files associated with the same output will also be reprocessed. This happens repeatedly until there is no new dirty file.

  Why? An output is made out of a set of inputs. Processors may need all the inputs to regenerate the output.

- If an unchanged input file isn't associated with any aggregating outputs, it won't be reprocessed.

  Why? This file can't affect any outputs because it is unchanged and isn't associated with an aggregating output. It won't be reprocessed unless one of the above rules applies.

For example, consider a project with the following structure:

```
.
├── src
│   ├── sourceA.kt
│   └── sourceB.kt
└── generated
    ├── outputA
    └── outputB
```

A processor:

1. reads `sourceA`.

2. generates `outputA`.
3. reads `sourceB`.
4. generates `outputB`.

When `sourceA` changes:

- If `outputB` is aggregating, KSP reprocesses both `sourceA` and `sourceB`.
- If `outputB` is isolating, KSP reprocesses `sourceA` only.

If `sourceC` is added:

- If `outputB` is aggregating, KSP reprocesses `sourceC` and `sourceB`.
- If `outputB` is isolating, KSP reprocesses only `sourceC`.

If either `sourceA` or `sourceB` is removed, KSP doesn't need to reprocess any files.

Example processor

The following project contains classes `A` and `B`, where `A` extends `B`:

```
// A.kt
@Interesting
class A : B()

// B.kt
open class B

// Example1Processor.kt
class Example1Processor : SymbolProcessor {
   override fun process(resolver: Resolver) {
       val declA = resolver.getSymbolsWithAnnotation("Interesting").first() as KSClassDeclaration
       val declB = declA.superTypes.first().resolve().declaration
       // B.kt isn't required, because it can be deduced as a dependency by KSP
       val dependencies = Dependencies(aggregating = true, declA.containingFile!!)
       // outputForA.kt
       val outputName = "outputFor${declA.simpleName.asString()}"
       // outputForA depends on A.kt and B.kt
       val output = codeGenerator.createNewFile(dependencies, "com.example", outputName, "kt")
       output.write("// $declA : $declB\n".toByteArray())
       output.close()
   }
   // ...
}
```

To generate `outputForA`, the processor:

1. gets A by calling `Resolver.getSymbolsWithAnnotation`.
2. gets B by calling `KSClassDeclaration.superTypes` on A.

KSP tracks this relationship through resolution tracing and automatically records `B` as a dependency of `A`. Therefore, you don't need to explicitly declare `B.kt` as a dependency of `outputForA`.

Reporting bugs

If you encounter any error that occurs only when incremental processing is enabled, create an issue in the GitHub repository and attach the relevant log files.

1. Enable incremental processing logs by adding the following line to `gradle.properties`:

   ```
   ksp.incremental.log=true
   ```

2. Perform a clean build that completes successfully.
3. Save the generated log files by copying them to another location:
   - `build/kspCaches/<source set>/logs/kspDirtySet.log`
   - `build/kspCaches/<source set>/logs/kspSourceToOutputs.log`
4. Modify a source file that triggers the issue and run the build again.
5. Attach the log files from both the successful build and the build that reproduces the issue to the GitHub issue.

Visualizing the symbol dependency graph

To help debug incremental processing, KSP can generate a Graphviz DOT file that visualizes the symbol dependency graph starting from a specified symbol.

Enable incremental logging and specify the fully qualified name of the symbol to use as the starting point for visualizing the graph:

```
ksp.incremental.log=true
ksp.incremental.log.graph.origin=<fully-qualified-name>
```

If you're using KSP from the command line, add the following options:

```
-incremental-log=true -incremental-log.graph.origin=<fully-qualified-name>
```

The DOT file is generated in the `logs` directory.

Multiple-round processing

KSP supports multiple-round processing, or processing files over multiple rounds. The output from each processing round is used as additional input in each subsequent round.

To use multiple-round processing, return deferred symbols from `SymbolProcessor.process()` as a `List<KSAnnotated>`. KSP processes these symbols in the next round.

To defer invalid symbols, filter them with `KSAnnotated.validate()`, for example:

```
override fun process(resolver: Resolver): List<KSAnnotated> {
    val symbols = resolver.getSymbolsWithAnnotation("com.example.annotation.Builder")
    val result = symbols.filter { !it.validate() }
    symbols
        .filter { it is KSClassDeclaration && it.validate() }
        .map { it.accept(BuilderVisitor(), Unit) }
    return result
}
```

Multiple-round processing ends when an entire round generates no new files. If deferred symbols remain unprocessed, KSP logs an error for each processor with remaining deferred symbols.

Deferring symbols to the next round

Processors can defer symbols to a later round when additional information is required from other processors. A processor can continue deferring a symbol across multiple rounds until the required information becomes available. Once the information is available, the processor can process the symbol.

Defer symbols only when:

- Additional information is required before the symbol can be processed.
- The symbol originates from source code.

> Never defer symbols from the classpath. KSP filters out classpath symbols automatically.

For example, a processor that generates a builder for an annotated class might require all constructor parameter types to resolve to concrete types. In the first round, one of the parameter types might not be resolvable. In a later round, it can become resolvable because of files generated in the meantime. The processor can then process the class.

Validating symbols

Validation is a convenient way to determine whether to defer a symbol to a later round. A processor should define the information required to process a symbol correctly.

> Validation often requires type resolution, which can be expensive. Check only the information required to process the symbol.

Default validation behavior might not be suitable for all use cases. To customize validation, use `KSValidateVisitor` and provide a `predicate` lambda that selects the symbols to validate.

When implementing custom validation, use `KSType.isError` to determine whether a type is valid. If `isError` is `true`, KSP could not resolve the type. Use this information to decide whether to defer processing to a later round.

Accessing files and symbols

Both newly generated files and existing files are accessible through a `Resolver`.

KSP provides two APIs for accessing files:

- `Resolver.getAllFiles()` returns a list of both previously existing and newly generated files.
- `Resolver.getNewFiles()` returns only the files generated in the previous round.

Use `Resolver.getSymbolsWithAnnotation()` as the primary entry point for obtaining relevant symbols.

In each round, `Resolver.getSymbolsWithAnnotation()` returns only symbols from newly generated files and symbols deferred from the previous round. This helps avoid unnecessary reprocessing.

Processor instantiation

KSP creates a processor instance only once. You can store information in the processor instance and reuse it across multiple rounds.

However, not all KSP symbols can be reused across rounds. Symbol resolution results can change when processors generate new files, which can affect the validity of previously resolved symbols.

> Use only the `Resolver` instance passed to the processor in the current round. Do not store a `Resolver` and reuse it across rounds.

Error and exception handling

Errors

A processor reports an error by calling `KSPLogger.error()`.

When a processor reports an error, KSP calls `SymbolProcessor.onError()` instead of `SymbolProcessor.finish()`. Processing stops after the current round completes.

Other processors continue processing normally during that round. KSP handles errors only after all processors finish the current round.

Exceptions

KSP distinguishes between exceptions thrown by KSP and exceptions thrown by processors. Both types terminate processing immediately and are logged as errors through `KSPLogger`.

> Report exceptions thrown by KSP to the KSP developers for investigation. Create an issue in the KSP issue tracker.

At the end of a round in which an error or exception occurs, KSP calls `SymbolProcessor.onError()` on all processors. `SymbolProcessor` provides a default no-op implementation of `onError()`. Override this method to implement custom error-handling logic.

KSP with Kotlin Multiplatform

Here you'll learn how to use Kotlin Symbol Processing (KSP) in a Kotlin Multiplatform project. For a quick start, see an example of a multiplatform project with several targets using KSP in the source repository. The processor in this example generates a `Foo` class used by the project.

Add KSP to a multiplatform project

In the `build.gradle.kts` file of the client module (the module that uses the processor), add the appropriate KSP processor dependency for each target that requires symbol processing:

```
dependencies {
  add("ksp<Target>", <processor>)
}
```

- `<Target>` is one of the targets used in your multiplatform project.

  > For a full list of targets, see Multiplatform Gradle DSL reference and Kotlin/Native supported targets.

- `<processor>` is a Gradle project path. It can be:
  - a specific directory in your project that contains the logic for your symbol processor:

    ```
    add("kspJvm", project(":local-processor"))
    ```

  - an external processor such as Room:

    ```
    add("kspJvm", "androidx.room:room-compiler:2.6.1")
    ```

> Since KSP 2, the catch-all `ksp(...)` configuration is deprecated. Configure each target explicitly to avoid running processors where they are not needed.

Use multiple processors in a single target

You can add more than one processor to a target:

Kotlin

```
add("kspAndroid", project(":test-processor"))
add("kspAndroid", "androidx.room:room-compiler:2.6.1")
```

Groovy

```
add('kspAndroid', project(':test-processor'))
add('kspAndroid', 'androidx.room:room-compiler:2.6.1')
```

Use the same processor in multiple targets

You can add the same processor to more than one target:

Kotlin

```
add("kspIosX64", project(":test-processor"))
add("kspIosArm64", project(":test-processor"))
add("kspIosSimulatorArm64", project(":test-processor"))
```

Groovy

```
add('kspIosX64', project(':test-processor'))
add('kspIosArm64', project(':test-processor'))
add('kspIosSimulatorArm64', project(':test-processor'))
```

If you have many iOS targets, you can avoid repetition by looping:

Kotlin

```
kotlin.targets.filter { it.name.startsWith("ios") }.forEach { target →
    add(
        "ksp${target.name.replaceFirstChar { it.uppercaseChar() }}",
        project(":test-processor")
    )
}
```

Groovy

```
kotlin.targets.filter { it.name.startsWith("ios") }.forEach { target →
    add(
        "ksp${target.name.replaceFirstChar { it.uppercaseChar() }}",
        project(":test-processor")
    )
}
```

Configure KSP for test compilations

To run KSP during test compilation, add the processor to the corresponding test configurations:

Kotlin

```
add("kspJvmTest", project(":test-processor"))
add("kspJsTest", project(":test-processor"))
add("kspIosX64Test", project(":test-processor"))
```

Groovy

```
add('kspJvmTest', project(':test-processor'))
add('kspJsTest', project(':test-processor'))
add('kspIosX64Test', project(':test-processor'))
```

For Android host and device tests, KSP derives configuration names from the corresponding source set names:

Kotlin

```
add("kspAndroidHostTest", project(":test-processor"))
add("kspAndroidDeviceTest", project(":test-processor"))
```

Groovy

```
add('kspAndroidHostTest', project(':test-processor'))
add('kspAndroidDeviceTest', project(':test-processor'))
```

Find KSP configuration names

KSP derives configuration names from the Kotlin Multiplatform source sets. To view the complete list of KSP configurations for a module, run:

```
./gradlew :<your-module-name>:dependencies | grep ksp
```

Look for the configuration names that correspond to your target source sets.

Compilation and processing

In a multiplatform project, Kotlin creates a separate compilation for each target and source set, such as `main` and `test`. For each Kotlin compilation task with one or more configured KSP processors, KSP creates a corresponding symbol processing task.

The example project defines six targets. Each target has `main` and `test` compilations, resulting in the following compilation and symbol processing tasks:

- **JVM**: `jvmMain` and `jvmTest`
- **JS**: `jsMain` and `jsTest`
- **LinuxX64**: `linuxX64Main` and `linuxX64Test`
- **AndroidNativeX64**: `androidNativeX64Main` and `androidNativeX64Test`
- **AndroidNativeArm64**: `androidNativeArm64Main` and `androidNativeArm64Test`
- **MingwX64**: `mingwX64Main` and `mingwX64Test`

In the example's `workload/build.gradle.kts` file, KSP dependencies are declared for the following configurations:

- `kspJvm` and `kspJvmTest`
- `kspJs` and `kspJsTest`
- `kspAndroidNativeX64` and `kspAndroidNativeX64Test`
- `kspAndroidNativeArm64` and `kspAndroidNativeArm64Test`
- `kspLinuxX64`
- `kspMingwX64`

KSP creates a symbol processing task for each configuration where a KSP dependency is declared. In this example, the project creates at least 12 Kotlin compilation tasks and 10 symbol processing tasks. The remaining compilations don't have corresponding KSP tasks because KSP isn't configured for them.

Running KSP from the command line

Most projects use KSP through the Gradle plugin, which runs KSP automatically during compilation. You can use KSP from the command line but it's normally only used when you're integrating with other build systems, developing processors, testing, or debugging.

Since KSP is a JVM application, use the `java` command to launch KSP from the command line. Provide the classpath and any necessary arguments:

```
java -cp <classpath> <mainclass> <options> <processor>
```

Argument	Description
`<classpath>`	Path to KSP runtime JARs and their dependencies.
`<mainclass>`	One of the platform-specific KSP entry points.
`<options>`	Command-line options for KSP.
`<processor>`	Path to the processor JAR.

Classpath

Unlike the Gradle plugin, the `java` command doesn't resolve dependencies automatically. You must provide the KSP runtime JARs and their dependencies on the classpath.

Download `artifacts.zip` from the KSP release page. The archive contains the required KSP JAR files:

- `symbol-processing-aa-2.3.10.jar`
- `symbol-processing-common-deps-2.3.10.jar`
- `symbol-processing-api-2.3.10.jar`

You must also include the following runtime dependencies from the Maven Repository:

- `kotlin-stdlib-2.4.20.jar`
- `kotlinx-coroutines-core-jvm-1.11.0.jar`

Main class

Because KSP is a JVM application, you must specify the main class to launch. KSP provides a different entry point for each supported platform:

Entry point	Platforms
`KSPJvmMain`	Kotlin/JVM and Android
`KSPJsMain`	Kotlin/JS
`KSPNativeMain`	Kotlin/Native targets, such as iOS, macOS, Linux, and Windows.
`KSPCommonMain`	Common compilations in Kotlin Multiplatform projects.

When launching KSP with `java`, specify the fully qualified class name. For example:

```
java -cp <classpath> com.google.devtools.ksp.cmdline.KSPJvmMain <options> <processor>
```

The following example uses `KSPJvmMain` to run KSP for a JVM target:

```
java -cp \
symbol-processing-aa-2.3.10.jar:symbol-processing-common-deps-2.3.10.jar:symbol-processing-api-
2.3.10.jar:kotlin-stdlib-2.3.20.jar:kotlinx-coroutines-core-jvm-1.10.2.jar \
com.google.devtools.ksp.cmdline.KSPJvmMain \
-language-version=2.0 \
-api-version=2.0 \
-jvm-target=11 \
-module-name=main \
-source-roots=project_dir/src/kotlin/main \
-project-base-dir=project_dir/ \
-output-base-dir=project_dir/build/ \
-caches-dir=project_dir/build/caches/ \
-class-output-dir=project_dir/build/out/main/classes \
-kotlin-output-dir=project_dir/build/out/main/kotlin/ \
-java-output-dir=project_dir/build/out/main/java/ \
-resource-output-dir=project_dir/build/out/main/res/ \
path/to/processor.jar
```

Options

KSP requires the following options when running from the command line:

Option	Description
`-language-version=<version>`	The Kotlin language version used in the project.
`-api-version=<version>`	The Kotlin API version.
`-jvm-target=<version>`	The target JVM version.
`-module-name=<name>`	The module name.
`-source-roots=<paths>`	The source root directories. Use a colon-separated list for multiple directories.
`-project-base-dir=<path>`	The project root directory.
`-output-base-dir=<path>`	The base directory for KSP output.
`-caches-dir=<path>`	The directory for KSP caches.
`-java-output-dir=<path>`	The directory for generated Java files.
`-class-output-dir=<path>`	The directory for generated class files.

Option	Description
`-kotlin-output-dir=<path>`	The directory for generated Kotlin files.
`-resource-output-dir=<path>`	The directory for generated resources.
`<processor>`	The processor classpath.

Other useful options

- `-libraries=<path>` : The classpath used to resolve dependencies referenced by the source files. Typically the module's compile classpath.
- `-jdk-home=<path>` : The JDK home directory. Use when the processor resolves Java symbols and requires access to the Java standard library.
- `-friends=<path>` : The classpath of the current module's friend modules. This is typically the module's friend classpath. For more information, see Friend modules.

> KSP also supports the `-Dksp.logging` JVM system property, which sets the logging level. Valid values are `error` , `warn` or `warning` , `info` , and `debug` . The default value is `warn` . KSP treats unsupported values as `warn` .

To see the full list of options, run the following command:

```
java -cp <classpath> <mainclass> -h
```

KSP FAQ

Why KSP?

KSP has several advantages over kapt:

- It is faster.
- The API is more fluent for Kotlin users.
- It supports multiple round processing on generated Kotlin sources.
- It is being designed with multiplatform compatibility in mind.

Why is KSP faster than kapt?

kapt has to parse and resolve every type reference in order to generate Java stubs, whereas KSP resolves references on-demand. Delegating to javac also takes time.

Additionally, KSP's incremental processing model has a finer granularity than just isolating and aggregating. It finds more opportunities to avoid reprocessing everything. Also, because KSP traces symbol resolutions

dynamically, a change in a file is less likely to pollute other files and therefore the set of files to be reprocessed is smaller. This is not possible for kapt because it delegates processing to javac.

Is KSP Kotlin-specific?

KSP can process Java sources as well. The API is unified, meaning that when you parse a Java class and a Kotlin class, you get a unified data structure in KSP.

How to upgrade KSP?

KSP has API and implementation. The API rarely changes and is backward compatible: there can be new interfaces, but old interfaces never change. The implementation is tied to a specific compiler version. With the new release, the supported compiler version can change.

Processors only depend on API and therefore are not tied to compiler versions. However, users of processors need to bump KSP version when bumping the compiler version in their project. Otherwise, the following error will occur:

```
ksp-a.b.c is too old for kotlin-x.y.z. Please upgrade ksp or downgrade kotlin-gradle-plugin
```

Users of processors don't need to bump processor's version because processors only depend on API.

For example, some processor is released and tested with KSP 1.0.1, which depends strictly on Kotlin 1.6.0. To make it work with Kotlin 1.6.20, the only thing you need to do is bump KSP to a version (for example, KSP 1.1.0) that is built for Kotlin 1.6.20.

Can I use a newer KSP implementation with an older Kotlin compiler?

If the language version is the same, Kotlin compiler is supposed to be backward compatible. Bumping Kotlin compiler should be trivial most of the time. If you need a newer KSP implementation, please upgrade the Kotlin compiler accordingly.

How often do you update KSP?

KSP tries to follow Semantic Versioning as close as possible. With KSP version `major.minor.patch`,

- `major` is reserved for incompatible API changes. There is no pre-determined schedule for this.
- `minor` is reserved for new features. This is going to be updated approximately quarterly.
- `patch` is reserved for bug fixes and new Kotlin releases. It's updated roughly monthly.

Usually a corresponding KSP release is available within a couple of days after a new Kotlin version is released, including the pre-releases (Beta or RC).

Besides Kotlin, are there other version requirements for libraries?

Here is a list of requirements for libraries/infrastructures:

- Android Gradle Plugin 7.1.3+
- Gradle 6.8.3+

What is KSP's future roadmap?

The following items have been planned:

- Support new Kotlin compiler
- Improve support for multiplatform. For example, running KSP on a subset of targets/sharing computations between targets.
- Improve performance. There are a bunch of optimizations to be done!
- Keep fixing bugs.

Please feel free to reach out to us in the #ksp channel in Kotlin Slack (get an invite) if you would like to discuss any ideas. Filing GitHub issues/feature requests or pull requests are also welcome!

Kotlin roadmap

Last modified on August 2026

Next update February 2027

Welcome to the Kotlin roadmap! Get a sneak peek into the priorities of the JetBrains team.

Key priorities

The goal of this roadmap is to give you the big picture. Here's a list of our key focus areas – the most important directions we are focused on delivering:

- **Language evolution**: keep Kotlin concise and expressive, prioritizing ergonomics and safety over ceremony.
- **Kotlin Multiplatform**: become the foundation for modern cross-platform apps through a solid iOS experience, mature web targets, and reliable IDE tooling.
- **Staying agnostic**: support developers no matter their tools or targets.
- **Experience of third-party ecosystem authors:** simplify the development and publication process for Kotlin libraries, tools, and frameworks.

Kotlin roadmap by subsystem

If you have any questions or feedback about the roadmap or the items on it, feel free to post them to YouTrack tickets or in the #kotlin-roadmap channel of Kotlin Slack (request an invite).

Subsystem	In focus now
Language	See the full list of Kotlin language features and proposals or follow the YouTrack issue for upcoming language features
Compiler	• 🆕 Promote Kotlin/Wasm to Stable • 🆕 Improve KAPT performance to be comparable with Java APT •

Subsystem	In focus now
	Stabilize overload resolution by lambda return type • Support K2 multiplatform incremental compilation of common code • New JVM reflection: Investigation, prototype and implementation • Kotlin/Wasm: Switch the `wasm-wasi` target of libraries to WASI Preview 2 • Kotlin/Wasm: Support the Component Model
Multiplatform	• 🆕 Improved rendering reliability and future-proof GPU support via Graphite in Skiko • 🆕 Xcode integration for Kotlin/Native debugger • 🆕 Swift Export: from Alpha to Beta • 🆕 Make Native Text Input default in Compose Multiplatform for iOS • 🆕 Native compiler caches in release mode • Unify inline semantics between stable Kotlin targets • Kotlin/JS: Improve onboarding materials for Kotlin/JS • Kotlin/JS: Compile to the modern JavaScript • Kotlin/JS: Extend possibilities for exporting Kotlin declarations to JavaScript • Enable incremental compilation of klib artifacts by default
Tooling	• 🆕 Streamline the Kotlin-on-Maven onboarding with a unified compiler-plugins bundle • 🆕 Xcode integration for Kotlin/Native debugger • 🆕 Enable Native tasks parallelization without configuration cache enabled • 🆕 Kotlin Toolchain: a single entry point into Kotlin • Kotlin/Native debugger health and performance improvements • Support for importing Swift Package Manager packages in Kotlin • Replace Karma runner with a not-deprecated alternative • Support Kotlin/JS and Kotlin/Wasm in Gradle project isolation • Design the Build tools API • Support Kotlin LSP and VS Code
Ecosystem	• 🆕 Implement first-class JPA/Hibernate support for Kotlin stdlib types • Stabilize the experimental `kotlinx.serialization` API • Improve the Lombok compiler plugin experience with Kotlin for server side • Stabilize `kotlinx.collections.immutable` • Promote `kotlinx-datetime` to Beta • Implement KDoc machine-readable representation •

Subsystem	In focus now
	New multiplatform API for the standard library: Support for Unicode and codepoints • Stabilize the `kotlinx-io` library • Introduce default warnings/errors for Kotlin functions that return non-unit values that are unused • Release Kotlin DataFrame 1.0 • Release Kandy 0.9 Ktor: • Improve authentication in Ktor • Support HTTP/3 • Add gRPC support to Ktor with a generator plugin and tutorial • Improve Ktor administration and observability Exposed: • Release Exposed DAO 2.0

- This roadmap is not an exhaustive list of all things the team is working on, only the biggest projects.
- There's no commitment to delivering specific features or fixes in specific versions.
- We will adjust our priorities as we go and update the roadmap approximately every six months.

What's changed since February 2026

Completed items

We've **completed** the following items from the previous roadmap:

- ✅ Compiler: Evolve the Power-assert plugin
- ✅ Compiler: Kotlin/Wasm: Support multi-module compilation
- ✅ Multiplatform: Swift Export: Alpha release
- ✅ Multiplatform: Implement new `TextInputService` on iOS for Compose Multiplatform
- ✅ Multiplatform: Swift 6.3 support
- ✅ Multiplatform: Stabilize Navigation3 for Compose Multiplatform
- ✅ Tooling: Smart defaults for Kotlin on Maven (mixed Java + Kotlin)
- ✅ Tooling: Release Kotlin ecosystem plugin supporting declarative Gradle
- ✅ Ecosystem: Improve Kotlin distribution UX: Add code coverage and binary compatibility validation
- ✅ Ecosystem: Introduce 18-month support window for security fixes in the standard library
- ✅ Ecosystem: Create a migration Gradle plugin for Exposed

New items

We've **added** the following items to the roadmap:

- 🆕 Compiler: Promote Kotlin/Wasm to Stable
- 🆕 Compiler: Improve KAPT performance to be comparable with Java APT
- 🆕 Multiplatform: Improved rendering reliability and future-proof GPU support via Graphite in Skiko
- 🆕 Multiplatform: Swift Export: from Alpha to Beta
- 🆕 Multiplatform: Make Native Text Input default in Compose Multiplatform for iOS
- 🆕 Multiplatform: Native compiler caches in release mode
- 🆕 Tooling: Streamline the Kotlin-on-Maven onboarding with a unified compiler-plugins bundle
- 🆕 Tooling: Xcode integration for Kotlin/Native debugger
- 🆕 Tooling: Enable Native tasks parallelization without configuration cache enabled
- 🆕 Tooling: Kotlin Toolchain: a single entry point into Kotlin
- 🆕 Ecosystem: Implement first-class JPA/Hibernate support for Kotlin stdlib types

Removed items

We've **removed** the following items from the roadmap:

- ❌ Multiplatform: Implement the next generation distribution format of multiplatform libraries
- ❌ Tooling: Improve Kotlin scripting and experience with `.gradle.kts`
- ❌ Ecosystem: Stabilize Kotlin Notebooks

Kotlin language features and proposals

JetBrains evolves the Kotlin language according to the Kotlin language evolution principles, guided by pragmatic design.

> Language feature proposals are listed from Kotlin 1.7.0.
>
> See the explanation of language feature statuses in the Kotlin evolution principles documentation.

All

Exploration and design	**Support immutability** • KEEP notes: immutability • YouTrack issue: KT-77734
KEEP discussion	**`CoroutineContext` as context parameter** • KEEP proposal: CoroutineContext-context-parameter.md • YouTrack issue: KT-15555
KEEP discussion	**Rich Errors: motivation and rationale** • KEEP proposal: rich-errors-motivation.md • YouTrack issue: KT-68296
KEEP discussion	**Kotlin companion blocks and extensions (statics)** • KEEP proposal: companion-blocks.md • YouTrack issue: KT-11968
KEEP discussion	**Version overloading** • KEEP proposal: version-overloading.md
KEEP discussion	**Streamline KDoc ambiguity links** • KEEP proposal: streamline-KDoc-ambiguity-references.md • GitHub issues: dokka/#3451, dokka/#3179, dokka/#3334
KEEP discussion	**Resolution of links to extensions in KDoc** • KEEP proposal: links-to-extensions.md • GitHub issue: dokka/#3555
In preview	**Collection literals** • KEEP proposal: collection-literals.md • YouTrack issue: KT-43871 • Stability level: Experimental • Available since: 2.4.0
In preview	**Improve compile-time constants** • KEEP proposal: improve-compile-time-constants.md • YouTrack issue: KT-22505 • Stability level: Experimental • Available since: 2.4.0

In preview	**Name-based destructuring** • KEEP proposal: name-based-destructuring.md • YouTrack issue: KT-19627 • Stability level: Experimental • Available since: 2.3.20
In preview	**Unused return value checker** • KEEP proposal: unused-return-value-checker.md • YouTrack issue: KT-12719 • Stability level: Experimental • Available since: 2.3.0
In preview	**Context-sensitive resolution** • KEEP proposal: context-sensitive-resolution.md • YouTrack issue: KT-16768 • Stability level: Experimental • Available since: 2.2.0
In preview	**Expose boxed inline value classes in JVM** • KEEP proposal: jvm-expose-boxed.md • YouTrack issue: KT-28135 • Stability level: Experimental • Available since: 2.2.0
In preview	**Common Atomics and Atomic Arrays** • KEEP proposal: Common atomics • YouTrack issue: KT-62423 • Stability level: Experimental • Available since: 2.2.0
In preview	**KMP Kotlin-to-Java direct actualization** • KEEP proposal: kmp-kotlin-to-java-direct-actualization.md • YouTrack issue: KT-67202 • Stability level: Experimental • Available since: 2.1.0
Stable	**Context parameters: support for context-dependent declarations** • KEEP proposal: context-parameters.md • YouTrack issue: KT-10468 • Available since: 2.2.0, Stable since 2.4.0

Stable	**Improvements to annotation use-site targets on properties** • KEEP proposal: Improvements to annotation use-site targets on properties • YouTrack issue: KT-73255 • Available since: 2.2.0, Stable since 2.4.0
Stable	**Explicit backing fields** • KEEP proposal: explicit-backing-fields.md • YouTrack issue: KT-14663 • Available since: 2.3.0, Stable since 2.4.0
Stable	**Uuid** • KEEP proposal: uuid.md • YouTrack issue: KT-31880 • Available since: 2.0.20, Stable since 2.4.0
Stable	**Data flow-based exhaustiveness checking** • KEEP proposal: dfa-exhaustiveness.md • YouTrack issue: KT-8781 • Available since: 2.2.20, Stable since 2.3.0
Stable	**Nested (non-capturing) type aliases** • KEEP proposal: Nested (non-capturing) type aliases • YouTrack issue: KT-45285 • Available since: 2.2.0, Stable since 2.3.0
Stable	**kotlin.time.Instant** • KEEP proposal: Instant and Clock • YouTrack issue: KT-80778 • Available since: 2.1.0, Stable since 2.3.0
Stable	**Guard conditions in when-with-subject** • KEEP proposal: guards.md • YouTrack issue: KT-13626 • Available since: 2.2.0
Stable	**Multidollar interpolation: improved handling of `$` in string literals** • KEEP proposal: dollar-escape.md • YouTrack issue: KT-2425 • Available since: 2.2.0

Stable	**Non-local `break` and `continue`** • KEEP proposal: break-continue-in-inline-lambdas.md • YouTrack issue: KT-1436 • Available since: 2.2.0
Stable	**Stabilized `@SubclassOptInRequired`** • KEEP proposal: subclass-opt-in-required.md • YouTrack issue: KT-54617 • Available since: 2.1.0
Stable	**`Enum.entries`: performant replacement of the `Enum.values()`** • KEEP proposal: enum-entries.md • YouTrack issue: KT-48872 • Available since: 2.0.0
Stable	**Data objects** • KEEP proposal: data-objects.md • YouTrack issue: KT-4107 • Available since: 1.9.0
Stable	**RangeUntil operator `..<`** • KEEP proposal: open-ended-ranges.md • YouTrack issue: KT-15613 • Available since: 1.7.20
Stable	**Definitely non-nullable types** • KEEP proposal: definitely-non-nullable-types.md • YouTrack issue: KT-26245 • Available since: 1.7.0
Revoked	**Context receivers** • KEEP proposal: context-receivers.md • YouTrack issue: KT-10468 • Replaced with context-parameters.md
Revoked	**Java synthetic property references** • KEEP proposal: references-to-java-synthetic-properties.md • YouTrack issue: KT-8575

Exploration and design	Support immutability
	• KEEP notes: immutability • YouTrack issue: KT-77734

KEEP discussion

KEEP discussion	**`CoroutineContext` as context parameter** • KEEP proposal: CoroutineContext-context-parameter.md • YouTrack issue: KT-15555
KEEP discussion	**Rich Errors: motivation and rationale** • KEEP proposal: rich-errors-motivation.md • YouTrack issue: KT-68296
KEEP discussion	**Kotlin companion blocks and extensions (statics)** • KEEP proposal: companion-blocks.md • YouTrack issue: KT-11968
KEEP discussion	**Version overloading** • KEEP proposal: version-overloading.md
KEEP discussion	**Streamline KDoc ambiguity links** • KEEP proposal: streamline-KDoc-ambiguity-references.md • GitHub issues: dokka/#3451, dokka/#3179, dokka/#3334
KEEP discussion	**Resolution of links to extensions in KDoc** • KEEP proposal: links-to-extensions.md • GitHub issue: dokka/#3555

In preview

In preview	**Collection literals** • KEEP proposal: collection-literals.md • YouTrack issue: KT-43871 • Stability level: Experimental • Available since: 2.4.0
In preview	**Improve compile-time constants** • KEEP proposal: improve-compile-time-constants.md • YouTrack issue: KT-22505 • Stability level: Experimental • Available since: 2.4.0
In preview	**Name-based destructuring** • KEEP proposal: name-based-destructuring.md • YouTrack issue: KT-19627 • Stability level: Experimental • Available since: 2.3.20

In preview

Unused return value checker

- KEEP proposal: unused-return-value-checker.md
- YouTrack issue: KT-12719
- Stability level: Experimental
- Available since: 2.3.0

In preview

Context-sensitive resolution

- KEEP proposal: context-sensitive-resolution.md
- YouTrack issue: KT-16768
- Stability level: Experimental
- Available since: 2.2.0

In preview

Expose boxed inline value classes in JVM

- KEEP proposal: jvm-expose-boxed.md
- YouTrack issue: KT-28135
- Stability level: Experimental
- Available since: 2.2.0

In preview

Common Atomics and Atomic Arrays

- KEEP proposal: Common atomics
- YouTrack issue: KT-62423
- Stability level: Experimental
- Available since: 2.2.0

In preview

KMP Kotlin-to-Java direct actualization

- KEEP proposal: kmp-kotlin-to-java-direct-actualization.md
- YouTrack issue: KT-67202
- Stability level: Experimental
- Available since: 2.1.0

Stable

Stable

Context parameters: support for context-dependent declarations

- KEEP proposal: context-parameters.md
- YouTrack issue: KT-10468
- Available since: 2.2.0, Stable since 2.4.0

Stable

Improvements to annotation use-site targets on properties

- KEEP proposal: Improvements to annotation use-site targets on properties
- YouTrack issue: KT-73255
- Available since: 2.2.0, Stable since 2.4.0

Stable	**Explicit backing fields** • KEEP proposal: explicit-backing-fields.md • YouTrack issue: KT-14663 • Available since: 2.3.0, Stable since 2.4.0
Stable	**Uuid** • KEEP proposal: uuid.md • YouTrack issue: KT-31880 • Available since: 2.0.20, Stable since 2.4.0
Stable	**Data flow-based exhaustiveness checking** • KEEP proposal: dfa-exhaustiveness.md • YouTrack issue: KT-8781 • Available since: 2.2.20, Stable since 2.3.0
Stable	**Nested (non-capturing) type aliases** • KEEP proposal: Nested (non-capturing) type aliases • YouTrack issue: KT-45285 • Available since: 2.2.0, Stable since 2.3.0
Stable	**kotlin.time.Instant** • KEEP proposal: Instant and Clock • YouTrack issue: KT-80778 • Available since: 2.1.0, Stable since 2.3.0
Stable	**Guard conditions in when-with-subject** • KEEP proposal: guards.md • YouTrack issue: KT-13626 • Available since: 2.2.0
Stable	**Multidollar interpolation: improved handling of `$` in string literals** • KEEP proposal: dollar-escape.md • YouTrack issue: KT-2425 • Available since: 2.2.0
Stable	**Non-local `break` and `continue`** • KEEP proposal: break-continue-in-inline-lambdas.md • YouTrack issue: KT-1436 • Available since: 2.2.0

Stable	**Stabilized `@SubclassOptInRequired`** • KEEP proposal: subclass-opt-in-required.md • YouTrack issue: KT-54617 • Available since: 2.1.0
Stable	**`Enum.entries` : performant replacement of the `Enum.values()`** • KEEP proposal: enum-entries.md • YouTrack issue: KT-48872 • Available since: 2.0.0
Stable	**Data objects** • KEEP proposal: data-objects.md • YouTrack issue: KT-4107 • Available since: 1.9.0
Stable	**RangeUntil operator `..<`** • KEEP proposal: open-ended-ranges.md • YouTrack issue: KT-15613 • Available since: 1.7.20
Stable	**Definitely non-nullable types** • KEEP proposal: definitely-non-nullable-types.md • YouTrack issue: KT-26245 • Available since: 1.7.0

Revoked

Revoked	**Context receivers** • KEEP proposal: context-receivers.md • YouTrack issue: KT-10468 • Replaced with context-parameters.md
Revoked	**Java synthetic property references** • KEEP proposal: references-to-java-synthetic-properties.md • YouTrack issue: KT-8575

Kotlin evolution principles

Principles of pragmatic evolution

> *Language design is cast in stone,*
>
> *but this stone is reasonably soft,*
>
> *and with some effort we can reshape it later.*
>
> *Kotlin Design Team*

Kotlin is designed to be a pragmatic tool for programmers. When it comes to language evolution, its pragmatic nature is captured by the following principles:

- Keep the language modern over time.
- Maintain a continuous feedback loop with users.
- Make updating to new versions easy and comfortable for the users.

As this is key to understanding how Kotlin is moving forward, let's expand on these principles.

Keeping the Language Modern. We recognize that systems accumulate legacy over time. What was once cutting-edge technology can become hopelessly outdated today. We have to evolve the language to keep it relevant to the needs of our users and up-to-date with their expectations. This includes not only adding new features, but also phasing out old ones that are no longer recommended for production use and have become legacy.

Comfortable Updates. Incompatible changes, such as removing things from a language, may lead to painful migration from one version to the next if carried out without proper care. We will always announce such changes well in advance, mark things as deprecated and provide automated migration tools *before the change happens*. By the time the language changes, we want most of the code in the world to be already updated and thus have no issues migrating to the new version.

Feedback Loop. Going through deprecation cycles requires significant effort, so we want to minimize the number of incompatible changes we'll be making in the future. Apart from using our best judgement, we believe that trying things out in real life is the best way to validate a design. Before casting things in stone, we want them battle-tested. This is why we use every opportunity to make early versions of our designs available in production versions of the language, but in one of the *pre-stable* statuses: Experimental, Alpha, or Beta. Such features are not stable; they can be changed at any time, and the users that opt into using them do so explicitly to indicate that they are ready to deal with the future migration issues. These users provide invaluable feedback that we gather to iterate on the design and make it rock-solid.

Incompatible changes

If, upon updating from one version to another, some code that used to work doesn't work anymore, it is an *incompatible change* in the language (sometimes referred to as a "breaking change"). There can be debates as to what "doesn't work anymore" means precisely in some cases, but it definitely includes the following:

- Code that compiled and ran fine is now rejected with an error (at compile or link time). This includes removing language constructs and adding new restrictions.
- Code that executed normally is now throwing an exception.

The less obvious cases that belong to the "gray area" include handling corner cases differently, throwing an exception of a different type than before, changing behavior observable only through reflection, modifying undocumented or undefined behavior, renaming binary artifacts, and others. Sometimes such changes are crucial and affect migration experience dramatically; sometimes they are insignificant.

Some examples of what definitely isn't an incompatible change include:

- Adding new warnings.
- Enabling new language constructs or relaxing limitations for existing ones.
- Changing private/internal APIs and other implementation details.

The principles of Keeping the Language Modern and Comfortable Updates suggest that incompatible changes are sometimes necessary, but they should be introduced carefully. Our goal is to make users aware of upcoming changes well in advance to let them migrate their code comfortably.

Ideally, every incompatible change should be announced through a compile-time warning reported in the problematic code (usually referred to as a *deprecation warning*) and accompanied by automated migration aids. So, the ideal migration workflow is as follows:

- Update to version A (where the change is announced)
  - See warnings about the upcoming change
  - Migrate the code with help from tools
- Update to version B (where the change happens)
  - See no issues at all

In practice, some changes can't be accurately detected at compile time, so no warnings can be reported, but at least the users will be notified through the release notes of version A that a change is coming in version B.

Dealing with compiler bugs

Compilers are complicated software, and despite the best efforts of their developers, they have bugs. The bugs that cause the compiler itself to fail or report spurious errors or generate obviously failing code, though annoying and often embarrassing, are easy to fix, because the fixes do not constitute incompatible changes. Other bugs may cause the compiler to generate incorrect code that does not fail: for example, by missing some errors in the source or simply generating the wrong instructions. Fixes for such bugs are technically incompatible changes (some code used to compile fine, but now it won't anymore), but we are inclined to fix them as soon as possible to prevent the bad code patterns from spreading across user code. In our opinion, this supports the principle of Comfortable Updates, because fewer users have a chance of encountering the issue. Of course, this applies only to bugs that are found soon after appearing in a released version.

Decision making

JetBrains, the original creator of Kotlin, is driving its progress with the help of the community and in collaboration with the Kotlin Foundation.

All changes to the Kotlin Programming Language are overseen by the Lead Language Designer (currently Michail Zarečenskij). The Lead Designer has the final say in all matters related to language evolution. Additionally, incompatible changes to fully stable components have to be approved by the Language Committee, designated under the Kotlin Foundation (currently comprising Jeffrey van Gogh, Werner Dietl, and Michail Zarečenskij).

The Language Committee makes final decisions on which incompatible changes will be made and what exact measures should be taken to make user updates as seamless as possible. In doing so, it relies on a set of Language committee guidelines.

Language feature delivery

As described in the Kotlin release process, language features are shipped in *language releases* (2. *x*. *0*) or their following *tooling releases* (2. *x*. *20*).

We try to keep language and tooling releases compatible with each other, so changes to the compiler are mostly optimizations and warning additions/removals. Pre-stable features may be added, removed, or changed at any time.

Language releases often add new features, promote pre-stable features to stable, and may remove or change previously deprecated ones.

EAP builds

Before releasing stable versions of language and tooling releases, we publish a number of preview builds dubbed *EAP* (for "Early Access Preview") that let us iterate faster and gather feedback from the community. EAPs of language releases usually produce binaries that will be later rejected by the stable compiler to make sure that possible bugs in the binary format survive no longer than the preview period. Final release candidates, such as RC2 or RC3, normally don't bear this limitation. For more information, see Participate in the Kotlin Early Access Preview.

Pre-stable features

According to the Feedback Loop principle described above, we iterate on our designs in the open and release versions of the language where some features have one of the *pre-stable* statuses and *are supposed to change*. Such features can be added, changed or removed at any point and without warning. We do our best to ensure that pre-stable features can't be used accidentally by an unsuspecting user. Such features usually require some sort of explicit opt-in either in the code or in the project configuration.

A Kotlin language feature can have one of the following statuses:

- **Exploration and design**. We are considering the introduction of a new feature to the language. This involves discussing how it would integrate with existing features, gathering use cases, and assessing its potential impact. We need feedback from users on the problems this feature would solve and the use cases it addresses. Whenever possible, we also try to estimate how often these use cases and problems occur. Typically, ideas are documented as YouTrack issues, where the discussion continues.
- **KEEP discussion**. We are fairly certain that the feature should be added to the language. We aim to provide motivation, use cases, design, and other important details in a document called a *Kotlin Evolution and Enhancement Process* (KEEP). We expect feedback from users to focus on discussing all the information provided in the KEEP.
- **In preview**. A feature prototype is ready, and you can enable it using a feature-specific compiler option. We seek feedback on your experience with the feature, including how easily it integrates into your codebase, how it interacts with existing code, and any IDE support issues or suggestions. The feature's design may change significantly, or it could be completely revoked based on feedback. When a feature is *in preview*, it has either Experimental or Beta stability levels.
- **Stable**. The language feature is now a first-class citizen in the Kotlin language. We guarantee its backward compatibility and that we'll provide tooling support.

- **Revoked**. We have revoked the proposal and will not implement the feature in the Kotlin language. We may revoke a feature that is *in preview* if it is not a good fit for Kotlin.

See the full list of Kotlin language proposals and their statuses.

Status of different components

Learn more about the stability status of different components in Kotlin, such as Kotlin/JVM, JS, and Native compilers, and various libraries.

Libraries

A language is nothing without its ecosystem, so we pay extra attention to enable smooth library evolution.

Ideally, a new version of a library can be used as a "drop-in replacement" for an older version. This means that upgrading a binary dependency should not break anything, even if the application is not recompiled (this is possible under dynamic linking).

On the one hand, to achieve this, the compiler has to provide certain *Application Binary Interface* (ABI) stability guarantees under the constraints of separate compilation. This is why every change in the language is examined from a binary compatibility standpoint.

On the other hand, a lot depends on the library authors being careful about which changes are safe to make. Thus, it's crucial that library authors understand how source changes affect compatibility and follow certain best practices to keep both APIs and ABIs of their libraries stable. Here are some assumptions that we make when considering language changes from the library evolution standpoint:

- Library code should always specify return types of public/protected functions and properties explicitly, thus never relying on type inference for public API. Subtle changes in type inference may cause return types to change inadvertently, leading to binary compatibility issues.
- Overloaded functions and properties provided by the same library should do essentially the same thing. Changes in type inference may result in more precise static types being known at call sites, causing changes in overload resolution.

Library authors can use the `@Deprecated` and `@RequiresOptIn` annotations to control the evolution of their API surface. Note that `@Deprecated(level=HIDDEN)` can be used to preserve binary compatibility even for declarations removed from the API.

Also, by convention, packages named "internal" are not considered public API. All API residing in packages named "experimental" is considered pre-stable and can change at any moment.

We evolve the Kotlin Standard Library (`kotlin-stdlib`) for stable platforms according to the principles stated above. Changes to the contracts for its API undergo the same procedures as changes in the language itself.

Compiler options

Command line options accepted by the compiler are also a kind of public API, and they are subject to the same considerations. Supported options (those that don't have the "-X" or "-XX" prefix) can be added only in language releases and should be properly deprecated before removing them. The "-X" and "-XX" options are experimental and can be added and removed at any time.

Compatibility tools

As legacy features get removed and bugs fixed, the source language changes, and old code that has not been properly migrated may not compile anymore. The normal deprecation cycle allows a comfortable period of time for migration, and even when it's over and the change ships in a stable version, there's still a way to compile non-migrated code.

Compatibility options

We provide compatibility options that make a new Kotlin version emulate the behavior of an older one:

- `-language-version X.Y` – compatibility mode for Kotlin language version X.Y. The compiler reports errors when your code uses language features introduced in later versions.
- `-api-version X.Y` – compatibility mode for Kotlin API version X.Y. The compiler ignores declarations that use Kotlin standard library APIs introduced in later versions, including APIs referenced by compiler-generated code.

To give you more time to migrate, on the JVM we support at least three previous language and API versions in addition to the latest stable version. This lets library authors adopt newer compiler releases while remaining compatible with consumers using older compiler versions. On other platforms, you can also configure older language and API versions, but unlike on the JVM, consumers still need to use the latest compiler version.

In most projects, set both options to the same version. A lower API version is mainly useful when you need to remain compatible with an older version of the Kotlin standard library.

Actively maintained code bases can benefit from getting bug fixes as soon as possible, without waiting for a full deprecation cycle to complete. These projects can enable the `-progressive` option to adopt these changes in tooling releases before they become the default.

You can configure these options on the command line or with the Gradle or Maven build tools.

Evolving the binary format

Unlike sources that can be fixed by hand in the worst case, binaries are a lot harder to migrate, and this makes backwards compatibility crucial in the case of binaries. Incompatible changes to binaries can make updates very uncomfortable and thus should be introduced with even more care than those in the source language syntax.

For fully stable versions of the compiler, the default binary compatibility protocol is the following:

- All binaries are backwards compatible; that means a newer compiler can read older binaries (for example, 1.3 understands 1.0 through 1.2).
- Older compilers reject binaries that rely on new features (for example, the 1.0 compiler rejects binaries that use coroutines).
- Preferably (but we can't guarantee it), the binary format is mostly forwards compatible with the next language release, but not later ones (in the cases when new features are not used, for example, 1.9 can understand most binaries from 2.0, but not 2.1).

This protocol is designed for comfortable updates as no project can be blocked from updating its dependencies even if it's using a slightly outdated compiler.

Note that not all target platforms have reached this level of stability, but Kotlin/JVM has.

Kotlin klib binaries

Kotlin klib binaries have reached the Stable level in Kotlin 1.9.20. However, there are some compatibility details you need to keep in mind:

- klib binaries are backwards compatible starting with Kotlin 1.9.20. For example, the 2.0.x compiler can read binaries produced by the 1.9.2x compiler.
- Forward compatibility is *not* guaranteed. For example, the 2.0.x compiler is *not* guaranteed to read binaries produced by the 2.1.x compiler.

The Kotlin cinterop klib binaries are still in Beta. Currently, we cannot give specific compatibility guarantees between different Kotlin versions for cinterop klib binaries.

Stability of Kotlin components

The Kotlin language and toolset are divided into many components such as the compilers for the JVM, JS and Native targets, the Standard Library, various accompanying tools, and so on. Many of these components were officially released as **Stable**, which means that they were evolved in a backward-compatible way following the principles of *Comfortable Updates* and *Keeping the Language Modern*.

Following the *Feedback Loop* principle, we release many things early for the community to try out, so a number of components are not yet released as **Stable**. Some of them are at a very early stage, some are more mature. We mark them as **Experimental**, **Alpha** or **Beta** depending on how quickly each component evolves and the level of risk users take on when adopting it.

Stability levels explained

Here's a quick guide to these stability levels and their meaning:

Experimental means "try it only in toy projects":

- We're just testing an idea and its feasibility. We want some users to try it out and share their feedback. The scope is minimal, and the feature may change or disappear at any time. We don't guarantee stability or compatibility.

Alpha means "we are testing whether this should become production-ready":

- We intend to turn this into a product and are validating its user value and market fit to shape its final form. The feature set is still incomplete, and breaking changes are expected. If the hypothesis doesn't hold, we may significantly change or discontinue the feature.

Beta means "you can use it, we'll do our best to minimize migration issues for you":

- It's almost done, user feedback is especially important now.
- Still, it's not 100% finished, so changes are possible (including ones based on your own feedback).
- Watch for deprecation warnings in advance for the best update experience.

We collectively refer to *Experimental*, *Alpha* and *Beta* as **pre-stable** levels.

Stable means "use it even in most conservative scenarios":

- It's done. We will be evolving it according to our strict backward compatibility rules.

Please note that stability levels do not say anything about how soon a component will be released as Stable. Similarly, they do not indicate how much a component will be changed before release. They only say how fast a component is changing and how much risk of update issues users are running.

GitHub badges for Kotlin components

The Kotlin GitHub organization hosts different Kotlin-related projects. Some of them we develop full-time, while others are side projects.

Each Kotlin project has two GitHub badges describing its stability and support status:

- **Stability** status. This shows how quickly each project is evolving and how much risk the users are taking when adopting it. The stability status completely coincides with the stability level of the Kotlin language features and its components:
  - 

 stands for **Experimental**
  - 

 stands for **Alpha**
  - 

 stands for **Beta**
  - 

 stands for **Stable**
- **Support** status. This shows our commitment to maintaining a project and helping users to solve their problems. The level of support is unified for all JetBrains products.
  See the JetBrains Open Source document for details.

Stability of subcomponents

A stable component may have an experimental subcomponent, for example:

- a stable compiler may have an experimental feature;
- a stable API may include experimental classes or functions;
- a stable command-line tool may have experimental options.

We make sure to document precisely which subcomponents are not **Stable**. We also do our best to warn users where possible and ask to opt them in explicitly to avoid the accidental use of features that have not been released as stable.

Current stability of Kotlin components

By default, all new components have Experimental status.

Kotlin compiler

Component	Status	Status since version	Comments
Kotlin/JVM	Stable	1.0.0	

Component	Status	Status since version	Comments
Kotlin/Native	Stable	1.9.0	
Kotlin/JS	Stable	1.3.0	
Kotlin/Wasm	Beta	2.2.20	
Analysis API	Stable		

Core compiler plugins

Component	Status	Status since version	Comments
All-open	Stable	1.3.0	
No-arg	Stable	1.3.0	
SAM-with-receiver	Stable	1.3.0	
kapt	Stable	1.3.0	
Lombok	Alpha	2.3.20	
Power-assert	Experimental	2.0.0	

Kotlin libraries

Component	Status	Status since version	Comments
kotlin-stdlib (JVM)	Stable	1.0.0	
kotlinx-coroutines	Stable	1.3.0	
kotlinx-serialization	Stable	1.0.0	
kotlin-metadata-jvm	Stable	2.0.0	
kotlin-reflect (JVM)	Beta	1.0.0	
kotlinx-datetime	Alpha	0.2.0	

Component	Status	Status since version	Comments
kotlinx-io	Alpha	0.2.0	
kotlinx-collections-immutable	Alpha	0.3.5	

Kotlin Multiplatform

Component	Status	Status since version	Comments
Kotlin Multiplatform	Stable	1.9.20	
Kotlin Multiplatform plugin for Android Studio	Beta	0.8.0	Versioned separately from the language

Kotlin/Native

Component	Status	Status since version	Comments
Kotlin/Native Runtime	Stable	1.9.20	
Kotlin/Native interop with C and Objective-C	Beta	1.3.0	Stability of C and Objective-C library import
klib binaries	Stable	1.9.20	Not including cinterop klibs, see below
cinterop klib binaries	Beta	1.3.0	Stability of C and Objective-C library import
CocoaPods integration	Stable	1.9.20	

For more information about the level of support for different targets, see Kotlin/Native supported targets and hosts.

Build tools

Component	Status	Status since version	Comments
Kotlin Gradle plugin	Stable	1.1.1	The Kotlin Multiplatform Gradle plugin is stable from Kotlin 1.9.20.
Kotlin Maven plugin	Stable	1.0.0	

Build tools API (BTA)

Component	Status	Status since version	Comments
BTA for JVM	Beta	2.4.20	
BTA for JS	Alpha	2.4.20	
BTA for Wasm	Alpha	2.4.20	

The Build tools API currently doesn't support Kotlin/Native.

Language tools

Component	Status	Status since version	Comments
Scripting syntax and semantics	Alpha	1.2.0	
Scripting embedding and extension API	Beta	1.5.0	
Scripting IDE support	Beta		Available since IntelliJ IDEA 2023.1 and later
CLI scripting	Alpha	1.2.0	

Language features and design proposals

For language features and new design proposals, see Kotlin language features and proposals.

Kotlin release process

This page explains the Kotlin release cycle and the different types of releases we ship. It also includes details about past and upcoming Kotlin releases, along with instructions on how to update to a specific release.

Since Kotlin 2.0.0, we ship the following types of releases:

- *Language releases* (2. *x*. *0*) that bring major changes in the language and include tooling updates. Released once in 6 months.
- *Tooling releases* (2. *x*. *20*) that are shipped between language releases and include updates in the tooling, performance improvements, and bug fixes. Released in 3 months after corresponding *language release*.
- *Bug fix releases* (2. *x*. *yz*) that include bug fixes for *tooling releases*. There is no exact release schedule for these releases.

For example, for the language release 2.2.0, there was only one tooling release 2.2.20 and one bug fix release 2.2.21.

For each language and tooling release, we also ship several preview (*EAP*) versions for you to try new features before they are released. For more information, see Participate in the Kotlin Early Access Preview.

If you want to be notified about new Kotlin releases, subscribe to the Kotlin newsletter, follow Kotlin on X, or enable the **Watch | Custom | Releases** option on the Kotlin GitHub repository.

Upcoming Kotlin releases

Here is the approximate schedule for upcoming stable Kotlin releases:

- **2.5.0**: Planned for December 2026
- **2.5.20**: Planned for March 2027

Update to a new Kotlin version

To upgrade your project to a new release, update the Kotlin version in your build system.

Gradle

To update to Kotlin 2.4.20, change the version of the Kotlin Gradle plugin in your `build.gradle(.kts)` file:

Kotlin

```
plugins {
    // Replace `<...>` with the plugin name appropriate for your target environment
    kotlin("<...>") version "2.4.20"
    // For example, if your target environment is JVM:
    // kotlin("jvm") version "2.4.20"
    // If your target is Kotlin Multiplatform:
    // kotlin("multiplatform") version "2.4.20"
}
```

Groovy

```
plugins {
    // Replace `<...>` with the plugin name appropriate for your target environment
    id 'org.jetbrains.kotlin.<...>' version '2.4.20'
    // For example, if your target environment is JVM:
    // id 'org.jetbrains.kotlin.jvm' version '2.4.20'
    // If your target is Kotlin Multiplatform:
    // id 'org.jetbrains.kotlin.multiplatform' version '2.4.20'
}
```

If you have projects created with earlier Kotlin versions, check if you also need to update the version of any kotlinx libraries.

If you are migrating to a new language release, the Kotlin plugin's migration tools will help you with the process.

To learn more about how to work with Gradle in your project, see Configure a Gradle project.

Maven

To update to Kotlin 2.4.20, change the version in your `pom.xml` file:

```
<properties>
    <kotlin.version>2.4.20</kotlin.version>
</properties>
```

Alternatively, you can change the version of the `kotlin-maven-plugin` in your `pom.xml` file:

```
<plugins>
    <plugin>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-maven-plugin</artifactId>
        <version>2.4.20</version>
    </plugin>
</plugins>
```

If you have projects created with earlier Kotlin versions, check if you also need to update the version of any kotlinx libraries.

To learn more about how to work with Maven in your project, see Maven.

IDE support

Kotlin has full out-of-the-box support in IntelliJ IDEA and Android Studio.

Standard library security support

Starting with Kotlin 2.4.0, the Kotlin standard library for the JVM has an 18–month support window for each release line. Language releases (2. *x*. *0*) and the following tooling releases (2. *x*. *20*) belong to the same release line (2. *x*).

If we discover a security vulnerability that affects the Kotlin standard library for the JVM, we ship the following simultaneously:

- A bug fix release based on the latest Kotlin release in the release line that includes the security fix.
- Bug fix releases for every release line within its active support window.

For example, if we discover a security vulnerability and the latest Kotlin release is Kotlin 2.4.20, we release a bug fix version for Kotlin 2.4.20 only. We don't release a bug fix version for Kotlin 2.4.0.

The following table lists all Kotlin releases and their support windows:

Kotlin release line	Release date	End of support	Latest bug fix release
2.4	June 3, 2026	December 3, 2027	2.4.10

We always want to hear about any security issues you find. To report problems that you discover in Kotlin, post a message directly to our issue tracker or send us an email.

Release history

The following table lists details of previous Kotlin releases:

You can also try Early Access Preview (EAP) versions of Kotlin.

Build info	Build highlights
2.4.20 Released: September 7, 2026 Release on GitHub	A tooling release with new experimental and stable features, performance improvements, bug fixes, and tooling updates. For more details, please refer to the changelog.
2.4.10 Released: July 14, 2026 Release on GitHub	A bug fix release for Kotlin 2.4.0. For more details, refer to the changelog.
2.4.0 Released: June 3, 2026 Release on GitHub	A language release including both new and stable language features, tooling updates, performance improvements for different platforms, and important fixes. Learn more about Kotlin 2.4.0 in What's new in Kotlin 2.4.0.
2.3.21 Released: April 23, 2026 Release on GitHub	A bug fix release for Kotlin 2.3.20. For more details, please refer to the changelog.
2.3.20 Released: March 16, 2026 Release on GitHub	A tooling release with performance improvements, bug fixes, and tooling updates. For more details, please refer to the changelog.
2.3.10 Released: February 5, 2026	A bug fix release for Kotlin 2.3.0 that includes performance improvements and an important fix for a rare race condition with `kotlinx.serialization`. For more details, please refer to the changelog.

Build info	Build highlights
Release on GitHub	
2.3.0 Released: December 16, 2025 Release on GitHub	A language release including both new and stable language features, tooling updates, performance improvements for different platforms, and important fixes. Learn more about Kotlin 2.3.0 in What's new in Kotlin 2.3.0.
2.2.21 Released: October 23, 2025 Release on GitHub	A bug fix release containing support for Xcode 26, along with other improvements and bug fixes. For more details, please refer to the changelog.
2.2.20 Released: September 10, 2025 Release on GitHub	A tooling release for Kotlin 2.2.0 that contains important changes for web development and other improvements. Learn more about Kotlin 2.2.20 in What's new in Kotlin 2.2.20.
2.2.10 Released: August 14, 2025 Release on GitHub	A bug fix release for Kotlin 2.2.0. For more details, please refer to the changelog.
2.2.0 Released: June 23, 2025 Release on GitHub	A language release including both new and stable language features, tooling updates, performance improvements for different platforms, and important fixes. Learn more about Kotlin 2.2.0 in What's new in Kotlin 2.2.0.
2.1.21 Released: May 13, 2025 Release on GitHub	A bug fix release for Kotlin 2.1.20. For more details, please refer to the changelog.
2.1.20 Released: March 20, 2025 Release on GitHub	A tooling release for Kotlin 2.1.0 containing new experimental features, performance improvements, and bug fixes. Learn more about Kotlin 2.1.20 in What's new in Kotlin 2.1.20.

Build info	Build highlights
2.1.10 Released: January 27, 2025 Release on GitHub	A bug fix release for Kotlin 2.1.0 For more details, please refer to the changelog.
2.1.0 Released: November 27, 2024 Release on GitHub	A language release introducing new language features. Learn more about Kotlin 2.1.0 in What's new in Kotlin 2.1.0.
2.0.21 Released: October 10, 2024 Release on GitHub	A bug fix release for Kotlin 2.0.20 For more details, please refer to the changelog.
2.0.20 Released: August 22, 2024 Release on GitHub	A tooling release for Kotlin 2.0.0 containing performance improvements and bug fixes. Features also include concurrent marking in Kotlin/Native's garbage collector, UUID support in Kotlin common standard library, Compose compiler updates, and support up to Gradle 8.8. Learn more about Kotlin 2.0.20 in What's new in Kotlin 2.0.20.
2.0.10 Released: August 6, 2024 Release on GitHub	A bug fix release for Kotlin 2.0.0. Learn more about Kotlin 2.0.0 in What's new in Kotlin 2.0.0.
2.0.0 Released: May 21, 2024 Release on GitHub	A language release with the Stable Kotlin K2 compiler. Learn more about Kotlin 2.0.0 in What's new in Kotlin 2.0.0.
1.9.25 Released: July 19, 2024 Release on GitHub	A bug fix release for Kotlin 1.9.20, 1.9.21, 1.9.22, 1.9.23, and 1.9.24. Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.
1.9.24	A bug fix release for Kotlin 1.9.20, 1.9.21, 1.9.22, and 1.9.23.

Build info	Build highlights
Released: May 7, 2024 Release on GitHub	Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.
1.9.23 Released: March 7, 2024 Release on GitHub	A bug fix release for Kotlin 1.9.20, 1.9.21, and 1.9.22. Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.
1.9.22 Released: December 21, 2023 Release on GitHub	A bug fix release for Kotlin 1.9.20 and 1.9.21. Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.
1.9.21 Released: November 23, 2023 Release on GitHub	A bug fix release for Kotlin 1.9.20. Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.
1.9.20 Released: November 1, 2023 Release on GitHub	A feature release with Kotlin K2 compiler in Beta and Stable Kotlin Multiplatform. Learn more in: • What's new in Kotlin 1.9.20
1.9.10 Released: August 23, 2023 Release on GitHub	A bug fix release for Kotlin 1.9.0. Learn more about Kotlin 1.9.0 in What's new in Kotlin 1.9.0. For Android Studio Giraffe and Hedgehog, the Kotlin plugin 1.9.10 will be delivered with upcoming Android Studios updates.
1.9.0 Released: July 6, 2023 Release on GitHub	A feature release with Kotlin K2 compiler updates, new enum class values function, new operator for open-ended ranges, preview of Gradle configuration cache in Kotlin Multiplatform, changes to Android target support in Kotlin Multiplatform, preview of custom memory allocator in Kotlin/Native. Learn more in: • What's new in Kotlin 1.9.0 •

Build info	Build highlights
	What's new in Kotlin YouTube video
1.8.22 Released: June 8, 2023 Release on GitHub	A bug fix release for Kotlin 1.8.20. Learn more about Kotlin 1.8.20 in What's new in Kotlin 1.8.20.
1.8.21 Released: April 25, 2023 Release on GitHub	A bug fix release for Kotlin 1.8.20. Learn more about Kotlin 1.8.20 in What's new in Kotlin 1.8.20. For Android Studio Flamingo and Giraffe, the Kotlin plugin 1.8.21 will be delivered with upcoming Android Studios updates.
1.8.20 Released: April 3, 2023 Release on GitHub	A feature release with Kotlin K2 compiler updates, AutoCloseable interface, and Base64 encoding in stdlib, new JVM incremental compilation enabled by default, new Kotlin/Wasm compiler backend. Learn more in: • What's new in Kotlin 1.8.20 • What's new in Kotlin YouTube video
1.8.10 Released: February 2, 2023 Release on GitHub	A bug fix release for Kotlin 1.8.0. Learn more about Kotlin 1.8.0. For Android Studio Electric Eel and Flamingo, the Kotlin plugin 1.8.10 will be delivered with upcoming Android Studios updates.
1.8.0 Released: December 28, 2022 Release on GitHub	A feature release with improved kotlin-reflect performance, new recursively copy or delete directory content experimental functions for JVM, improved Objective-C/Swift interoperability. Learn more in: • What's new in Kotlin 1.8.0 • Compatibility guide for Kotlin 1.8.0

Build info	Build highlights
1.7.21 Released: November 9, 2022 Release on GitHub	A bug fix release for Kotlin 1.7.20. Learn more about Kotlin 1.7.20 in What's new in Kotlin 1.7.20. For Android Studio Dolphin, Electric Eel, and Flamingo, the Kotlin plugin 1.7.21 will be delivered with upcoming Android Studios updates.
1.7.20 Released: September 29, 2022 Release on GitHub	An incremental release with new language features, the support for several compiler plugins in the Kotlin K2 compiler, the new Kotlin/Native memory manager enabled by default, and the support for Gradle 7.1. Learn more in: • What's new in Kotlin 1.7.20 • What's new in Kotlin YouTube video • Compatibility guide for Kotlin 1.7.20 Learn more about Kotlin 1.7.20.
1.7.10 Released: July 7, 2022 Release on GitHub	A bug fix release for Kotlin 1.7.0. Learn more about Kotlin 1.7.0. For Android Studio Dolphin (213) and Android Studio Electric Eel (221), the Kotlin plugin 1.7.10 will be delivered with upcoming Android Studios updates.
1.7.0 Released: June 9, 2022 Release on GitHub	A feature release with Kotlin K2 compiler in Alpha for JVM, stabilized language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs. Learn more in: • What's new in Kotlin 1.7.0 • What's new in Kotlin YouTube video • Compatibility guide for Kotlin 1.7.0
1.6.21 Released: April 20, 2022 Release on GitHub	A bug fix release for Kotlin 1.6.20. Learn more about Kotlin 1.6.20.
1.6.20 Released: April 4, 2022 Release on GitHub	An incremental release with various improvements such as: • Prototype of context receivers • Callable references to functional interface constructors •

Build info	Build highlights
	Kotlin/Native: performance improvements for the new memory manager • Multiplatform: hierarchical project structure by default • Kotlin/JS: IR compiler improvements • Gradle: compiler execution strategies Learn more about Kotlin 1.6.20.
1.6.10 Released: December 14, 2021 Release on GitHub	A bug fix release for Kotlin 1.6.0. Learn more about Kotlin 1.6.0.
1.6.0 Released: November 16, 2021 Release on GitHub	A feature release with new language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs. Learn more in: • Release blog post • What's new in Kotlin 1.6.0 • Compatibility guide
1.5.32 Released: November 29, 2021 Release on GitHub	A bug fix release for Kotlin 1.5.31. Learn more about Kotlin 1.5.30.
1.5.31 Released: September 20, 2021 Release on GitHub	A bug fix release for Kotlin 1.5.30. Learn more about Kotlin 1.5.30.
1.5.30 Released: August 23, 2021 Release on GitHub	An incremental release with various improvements such as: • Instantiation of annotation classes on JVM • Improved opt-in requirement mechanism and type inference • Kotlin/JS IR backend in Beta • Support for Apple Silicon targets • Improved CocoaPods support •

Build info	Build highlights
	Gradle: Java toolchain support and improved daemon configuration Learn more in: • Release blog post • What's new in Kotlin 1.5.30
1.5.21 Released: July 13, 2021 Release on GitHub	A bug fix release for Kotlin 1.5.20. Learn more about Kotlin 1.5.20.
1.5.20 Released: June 24, 2021 Release on GitHub	An incremental release with various improvements such as: • String concatenation via `invokedynamic` on JVM by default • Improved support for Lombok and support for JSpecify • Kotlin/Native: KDoc export to Objective-C headers and faster `Array.copyInto()` inside one array • Gradle: caching of annotation processors' classloaders and support for the `--parallel` Gradle property • Aligned behavior of stdlib functions across platforms Learn more in: • Release blog post • What's new in Kotlin 1.5.20
1.5.10 Released: May 24, 2021 Release on GitHub	A bug fix release for Kotlin 1.5.0. Learn more about Kotlin 1.5.0.
1.5.0 Released: May 5, 2021 Release on GitHub	A feature release with new language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs. Learn more in: • Release blog post • What's new in Kotlin 1.5.0 • Compatibility guide
1.4.32 Released: March 22, 2021	A bug fix release for Kotlin 1.4.30. Learn more about Kotlin 1.4.30.

Build info	Build highlights
Release on GitHub	
1.4.31 Released: February 25, 2021 Release on GitHub	A bug fix release for Kotlin 1.4.30 Learn more about Kotlin 1.4.30.
1.4.30 Released: February 3, 2021 Release on GitHub	An incremental release with various improvements such as: • New JVM backend, now in Beta • Preview of new language features • Improved Kotlin/Native performance • Standard library API improvements Learn more in: • Release blog post • What's new in Kotlin 1.4.30
1.4.21 Released: December 7, 2020 Release on GitHub	A bug fix release for Kotlin 1.4.20 Learn more about Kotlin 1.4.20.
1.4.20 Released: November 23, 2020 Release on GitHub	An incremental release with various improvements such as: • Supporting new JVM features, like string concatenation via `invokedynamic` • Improved performance and exception handling for Kotlin Multiplatform Mobile projects • Extensions for JDK Path: `Path("dir") / "file.txt"` Learn more in: • Release blog post • What's new in Kotlin 1.4.20
1.4.10 Released: September 7, 2020 Release on GitHub	A bug fix release for Kotlin 1.4.0. Learn more about Kotlin 1.4.0.

Build info	Build highlights
1.4.0 Released: August 17, 2020 Release on GitHub	A feature release with many features and improvements that mostly focus on quality and performance. Learn more in: • Release blog post • What's new in Kotlin 1.4.0 • Compatibility guide • Migrating to Kotlin 1.4.0
1.3.72 Released: April 15, 2020 Release on GitHub	A bug fix release for Kotlin 1.3.70. Learn more about Kotlin 1.3.70.

What's new in Kotlin 2.3.20

Released: March 16, 2026

The Kotlin 2.3.20 release is out! Here are the main highlights:

- **Gradle**: Compatibility with Gradle 9.3.0 and Kotlin/JVM compilation uses BTA by default
- **Maven**: Simplified setup for Kotlin projects
- **Kotlin compiler plugins**: Lombok is in Alpha and improved JPA support in the `kotlin.plugin.jpa` plugin
- **Language**: Support for name-based destructuring declarations
- **Standard library**: New API for creating immutable copies of `Map.Entry`
- **Kotlin/Native**: New interoperability mode for C and Objective-C libraries

For information about the Kotlin release cycle, see Kotlin release process.

Update to Kotlin 2.3.20

The latest version of Kotlin is included in the latest versions of IntelliJ IDEA and Android Studio.

To update to the new Kotlin version, make sure your IDE is updated to the latest version and change the Kotlin version to 2.3.20 in your build scripts.

Stable

New features

The following feature is Stable in this release:

Simplified setup for Kotlin projects

Kotlin 2.3.20 makes it easier to set up Kotlin in Maven projects. Now Kotlin supports the automatic configuration of source roots and Kotlin's standard library.

With the new configuration, when you create a new Kotlin project with the Maven build system or introduce Kotlin to your existing Java Maven project, you don't need to manually specify source root paths or add the `kotlin-stdlib` dependency in your POM build file.

How to enable

In your `pom.xml` file, add `<extensions>true</extensions>` to the `<build><plugins>` section of the Kotlin Maven plugin:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>2.4.20</version>
            <extensions>true</extensions> <!-- Add this extension  -->
        </plugin>
    </plugins>
</build>
```

The new features of the `<extensions>` option here are:

- Registering `src/main/kotlin` and `src/test/kotlin` directories as source roots in case they already exist but are not specified in the plugin configuration.
- Adding the `kotlin-stdlib` dependency in case it's not explicitly defined already.

You can also opt out of the automatic addition of Kotlin's standard library. For that, add the following to the `<properties>` section:

```
<project>
    <properties>
        <!-- Disable smart defaults via property -->
        <kotlin.smart.defaults.enabled>false</kotlin.smart.defaults.enabled>
    </properties>
</project>
```

Note that this property disables not only automatic addition of the standard library, but also the registration of source root paths. Other `<extensions>` features are not affected.

For more information on configuring Kotlin Maven projects, see Configure a Maven project.

New features

The following pre-stable features are available in this release. This includes features with Beta, Alpha, and Experimental status:

- Compiler: Lombok is now in Alpha
- Language: Name-based destructuring
- Standard library: New API for creating immutable copies of `Map.Entry`
- Kotlin/Native: New interoperability mode for C or Objective-C libraries

Lombok is now in Alpha

Kotlin 1.5.20 introduced the experimental Lombok compiler plugin, which lets you generate and use Java's Lombok declarations in modules that mix Kotlin and Java code.

In 2.3.20, the Lombok compiler plugin has been promoted to Alpha because we plan to make this functionality production-ready, but it's still under development.

Experimental

Name-based destructuring

Kotlin 2.3.20 introduces *name-based destructuring declarations*, which match variables to property names instead of relying on position-based `componentN()` functions.

Previously, destructuring declarations used position-based destructuring:

```
data class User(val username: String, val email: String)

fun main() {
    val user = User("alice", "alice@example.com")

    val (email, username) = user

    println(email)
    // alice

    println(username)
    // alice@example.com
}
```

In this example, because destructuring relies on the order of `componentN()` functions, `email` receives the value of `username`, and `username` receives the value of `email`.

Starting with Kotlin 2.3.20, you can use name-based destructuring where each variable refers to a property by name:

```
fun main() {
    val user = User("alice", "alice@example.com")

    // Uses name-based destructuring with explicit form
    (val mail = email, val name = username) = user

    println(name)
    // alice

    println(mail)
    // alice@example.com
}
```

Name-based destructuring is Experimental. You can control how the compiler interprets destructuring declarations with the `-Xname-based-destructuring` compiler option.

It has the following modes:

- `only-syntax` enables the explicit form of name-based destructuring without changing the behavior of existing destructuring declarations.
- `name-mismatch` reports warnings when position-based destructuring in data classes uses variable names that don't match the property names.

- `complete` enables short-form name-based destructuring with parentheses and continues supporting position-based destructuring with square bracket syntax.

If you use `complete` mode, the short-form destructuring syntax with parentheses matches variables to property names instead of relying on position:

```
val (email, username) = user
```

How to enable

To use named-based destructuring in your project, add the compiler option to your build configuration file:

Gradle

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xname-based-destructuring=only-syntax")
    }
}
```

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-Xname-based-destructuring=only-syntax</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Opting in to name-based destructuring also introduces a new syntax for position-based destructuring using square brackets:

```
// Uses explicit position-based destructuring
val [username, email] = user
```

We plan to gradually move toward destructuring declarations using name-based matching by default, while preserving position-based destructuring with the new square bracket syntax.

For more information, see the feature's KEEP.

We would appreciate your feedback in YouTrack.

Experimental

New API for creating immutable copies of `Map.Entry`

Kotlin 2.3.20 introduces the `Map.Entry.copy()` extension function for creating an immutable copy of a `Map.Entry`. This function allows you to reuse entries obtained from `Map.entries` after modifying the map by copying them first.

`Map.Entry.copy()` is Experimental. To opt in, use the `@OptIn(ExperimentalStdlibApi::class)` annotation or the compiler option:

```
-opt-in=kotlin.ExperimentalStdlibApi
```

Here's an example of using `Map.Entry.copy()` to remove entries from a mutable map:

```
@OptIn(ExperimentalStdlibApi::class)
fun main() {
    val map = mutableMapOf(1 to 1, 2 to 2, 3 to 3, 4 to 4)

    val toRemove = map.entries
        .filter { it.key % 2 == 0 }
        .map { it.copy() }

    map.entries.removeAll(toRemove)

    println("map = $map")
    // map = {1=1, 3=3}
}
```

Experimental

New interoperability mode for C or Objective-C libraries

If you use C or Objective-C libraries in your Kotlin Multiplatform (KMP) libraries or applications, we invite you to test the new interoperability mode and share the results.

In general, Kotlin/Native enables importing C and Objective-C libraries into Kotlin. However, for KMP libraries, this functionality is currently affected by the KMP compatibility issues with older compiler versions.

In other words, if you publish a KMP library compiled with one Kotlin version, importing C or Objective-C libraries might make it impossible to use that Kotlin library in projects with an earlier Kotlin version.

To address this and other issues, the Kotlin team has been revising the interoperability mechanism used under the hood. Starting with Kotlin 2.3.20, you can try the new mode through a compiler option.

How to enable

1. In your Gradle build file, check whether you have a `cinterops {}` block or a `pod()` dependency. If these are present, your project uses C or Objective-C libraries.
2. Ensure your project uses `2.3.20` or a later version.
3. In the same build file, add the `-Xccall-mode` compiler option to the cinterop tool invocation:

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget>().configureEach {
        compilations.configureEach {
            cinterops.configureEach {
                extraOpts += listOf("-Xccall-mode", "direct")
            }
        }
    }
}
```

4. Build and test your project as usual by running unit tests, the app, and so on. You can also use the `--continue` option to allow Gradle to continue executing tasks even after failures, helping to find more problems at once.

Do **not** publish libraries compiled with the new interoperability mode yet, as it's still Experimental.

Report your results

The new interoperability mode is supposed to be a drop-in replacement in most cases. We're planning to eventually enable it by default. But to achieve that, we need to ensure it works as well as possible and test it on a wide range of projects, because:

- Some C and Objective-C declarations aren't yet supported in the new mode (mostly because of compatibility issues). We'd like to better understand the real-world impact of this and prioritize future steps accordingly.
- There may be bugs or things we haven't considered. Testing languages with numerous interacting features is challenging, and testing the interaction between languages (each with a unique set of features) is even more so.

Help us examine real-world projects and identify challenging cases. Whether you encounter any issues or not, share your results in the comments to in YouTrack.

Language

Kotlin 2.3.20 adds name-based destructuring declarations that match variables to property names instead of relying on position. It also introduces changes to overload resolution for declarations with context parameters.

Changes to overload resolution for context parameters

Kotlin 2.3.20 introduces changes to overload resolution for declarations with context parameters.

Previously, overload resolution treated declarations with context parameters as more specific than those without them.

Starting with Kotlin 2.3.20, this rule no longer applies, making overload selection more uniform. As a result, calls that previously resolved now become ambiguous, resulting in a compilation error when overloads differ only by context parameters. In such cases, the compiler warns about potential ambiguity.

Here's an example:

```
class Logger {
    fun info(msg: String) = println("INFO: $msg")
}

fun saveUser(id: Int) {
    println("Saving user $id (no logger)")
}

// Reports a warning: Contextual declaration is shadowed
context(logger: Logger)
fun saveUser(id: Int) {
    logger.info("Saving user $id")
}

fun main() {
    val logger = Logger()

    context(logger) {
        // Reports an ambiguity error in 2.3.20
        saveUser(1)
    }
}
```

Additionally, Kotlin 2.3.20 reduces the number of `kotlin.context` overloads from 22 to 6 to reduce excessive overload candidates during resolution and code completion.

Experimental

Name-based destructuring

Kotlin 2.3.20 introduces *name-based destructuring declarations*, which match variables to property names instead of relying on position-based `componentN()` functions.

Previously, destructuring declarations used position-based destructuring:

```
data class User(val username: String, val email: String)

fun main() {
    val user = User("alice", "alice@example.com")

    val (email, username) = user

    println(email)
    // alice

    println(username)
    // alice@example.com
}
```

In this example, because destructuring relies on the order of `componentN()` functions, `email` receives the value of `username`, and `username` receives the value of `email`.

Starting with Kotlin 2.3.20, you can use name-based destructuring where each variable refers to a property by name:

```
fun main() {
    val user = User("alice", "alice@example.com")

    // Uses name-based destructuring with explicit form
    (val mail = email, val name = username) = user

    println(name)
    // alice

    println(mail)
    // alice@example.com
}
```

Name-based destructuring is Experimental. You can control how the compiler interprets destructuring declarations with the `-Xname-based-destructuring` compiler option.

It has the following modes:

- `only-syntax` enables the explicit form of name-based destructuring without changing the behavior of existing destructuring declarations.
- `name-mismatch` reports warnings when position-based destructuring in data classes uses variable names that don't match the property names.
- `complete` enables short-form name-based destructuring with parentheses and continues supporting position-based destructuring with square bracket syntax.

If you use `complete` mode, the short-form destructuring syntax with parentheses matches variables to property names instead of relying on position:

```
val (email, username) = user
```

How to enable

To use named-based destructuring in your project, add the compiler option to your build configuration file:

Gradle

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xname-based-destructuring=only-syntax")
    }
}
```

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-Xname-based-destructuring=only-syntax</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Opting in to name-based destructuring also introduces a new syntax for position-based destructuring using square brackets:

```
// Uses explicit position-based destructuring
val [username, email] = user
```

We plan to gradually move toward destructuring declarations using name-based matching by default, while preserving position-based destructuring with the new square bracket syntax.

For more information, see the feature's KEEP.

We would appreciate your feedback in YouTrack.

Standard library

Kotlin 2.3.20 includes a new experimental feature for the standard library.

Experimental

New API for creating immutable copies of `Map.Entry`

Kotlin 2.3.20 introduces the `Map.Entry.copy()` extension function for creating an immutable copy of a `Map.Entry`. This function allows you to reuse entries obtained from `Map.entries` after modifying the map by copying them first.

`Map.Entry.copy()` is Experimental. To opt in, use the `@OptIn(ExperimentalStdlibApi::class)` annotation or the compiler option:

```
-opt-in=kotlin.ExperimentalStdlibApi
```

Here's an example of using `Map.Entry.copy()` to remove entries from a mutable map:

```kotlin
@OptIn(ExperimentalStdlibApi::class)
fun main() {
    val map = mutableMapOf(1 to 1, 2 to 2, 3 to 3, 4 to 4)

    val toRemove = map.entries
        .filter { it.key % 2 == 0 }
        .map { it.copy() }

    map.entries.removeAll(toRemove)

    println("map = $map")
    // map = {1=1, 3=3}
}
```

Kotlin compiler plugins

Kotlin 2.3.20 brings important updates to the Lombok and `kotlin.plugin.jpa` compiler plugins.

Improved JPA support in the `kotlin.plugin.jpa` plugin

The `kotlin.plugin.jpa` plugin now automatically applies the `all-open` compiler plugin with the newly added built-in JPA preset, in addition to applying the existing `no-arg` compiler plugin.

Previously, using `kotlin("plugin.jpa")` enabled only the `no-arg` plugin with JPA presets.

In this release, we improved the `kotlin.plugin.jpa` preset so the `all-open` plugin is configured automatically. This ensures that lazy associations work as expected instead of causing eager loading and triggering extra queries.

Starting with Kotlin 2.3.20:

- The `all-open` compiler plugin provides a JPA preset.
- The Gradle `org.jetbrains.kotlin.plugin.jpa` plugin automatically applies the `org.jetbrains.kotlin.plugin.all-open` plugin with the JPA preset enabled.
- The Maven JPA setup enables `all-open` with the JPA preset by default. (Support in IntelliJ IDEA is available from 2026.1.)
- The Maven dependency `org.jetbrains.kotlin:kotlin-maven-noarg` now implicitly includes `org.jetbrains.kotlin:kotlin-maven-allopen`, so you no longer need to add it explicitly in the `<plugin> <dependencies>` block.

As a result, JPA entities annotated with the following annotations are automatically treated as `open` and receive no-argument constructors without additional configuration:

- `javax.persistence.Entity`
- `javax.persistence.Embeddable`
- `javax.persistence.MappedSuperclass`
- `jakarta.persistence.Entity`
- `jakarta.persistence.Embeddable`
- `jakarta.persistence.MappedSuperclass`

This change simplifies build configuration and improves the out-of-the-box experience when using Kotlin with JPA frameworks.

The upcoming release of IntelliJ IDEA 2026.1 automatically configures the `kotlin.plugin.jpa` plugin when setting up Kotlin in a project. The IDE offers a quick-fix to add the plugin and remove any redundant no-arg constructor declarations.

Lombok is now in Alpha

Kotlin 1.5.20 introduced the experimental Lombok compiler plugin, which lets you generate and use Java's Lombok declarations in modules that mix Kotlin and Java code.

In 2.3.20, the Lombok compiler plugin has been promoted to Alpha because we plan to make this functionality production-ready, but it's still under development.

Kotlin/JVM

Kotlin 2.3.20 introduces several improvements to Java interoperability. The compiler now recognizes the Vert.x `@Nullable` annotation for nullability checks. This release also adds support for the Java `@Unmodifiable` and `@UnmodifiableView` annotations to treat annotated collections as read-only in Kotlin.

Support for Vert.x `@Nullable` annotation

Kotlin 2.3.20 adds support for the `io.vertx.codegen.annotations.Nullable` annotation. The compiler now recognizes this annotation and reports nullability mismatches as warnings by default.

To enforce strict nullability checks and upgrade these warnings to errors, add the following compiler option to your build file:

Gradle

```
// build.gradle(.kts)
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xnullability-annotations=@io.vertx.codegen.annotations:strict")
    }
}
```

Maven

```
<!-- pom.xml -->
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-Xnullability-annotations=@io.vertx.codegen.annotations:strict</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Support for Java unmodifiable collection annotations

Kotlin 2.3.20 adds support for the `org.jetbrains.annotations.Unmodifiable` and `org.jetbrains.annotations.UnmodifiableView` Java annotations.

Starting from Kotlin 2.3.20, collections returned from Java declarations marked with these annotations are treated as read-only in Kotlin. Assigning them to a mutable collection type results in a type mismatch warning. This warning is scheduled to become an error in Kotlin 2.5.0.

Here's an example:

```java
// Java
public class Java {
    public static @UnmodifiableView List<Object> unmodifiableView() {
        return List.of();
    }

    public static @Unmodifiable List<Object> unmodifiable() {
        return List.of();
    }
}
```

```kotlin
// Kotlin

fun main() {
    // Reports a warning: Java type mismatch
    val mutableView: MutableList<Any> = Java.unmodifiableView()
    val mutableCopy: MutableList<Any> = Java.unmodifiable()
}
```

Kotlin/Native

Kotlin 2.3.20 introduces a new experimental interoperability mode for C and Objective-C libraries, a cross-compilation checker, and a new DSL for disabling the compilation cache in Kotlin/Native projects.

Cross-compilation checker

Kotlin 2.3.20 introduces a way to determine if cross-compilation is supported for the given target. This can be useful for third-party plugins that follow the status of compilation tasks.

Generally, Kotlin/Native allows cross-compilation, meaning that any supported host can produce `.klib` artifacts for supported targets. However, artifact production for Apple targets is still limited if your project uses cinterop dependencies.

The new `crossCompilationSupported` API now checks whether cross-compilation is supported: the target should be enabled by the host manager, and none of the target's compilations should involve cinterop dependencies. The checker is enabled by default.

For more information on supported targets and hosts, see the Kotlin/Native documentation.

New DSL for disabling compilation cache

Kotlin 2.3.20 comes with a new DSL for disabling the compilation cache in Kotlin/Native projects. It's intended to make the decision of disabling the cache more considered and explicit.

Since disabling the cache makes Kotlin/Native builds significantly slower, it should be used only temporarily and in exceptional cases. That's why disabling the cache is now tied to a specific Kotlin version and must include a reason, which acts as documentation.

If you do need to disable the compilation cache in your project, update the `binaries {}` block in your Gradle build file as follows:

```
kotlin {
    listOf(
        iosX64(),
        iosArm64(),
        iosSimulatorArm64()
    ).forEach {
        // Specify your binary kind
        it.binaries.framework {
            baseName = "CacheKind"
            isStatic = true

            // Disable cache with the new DSL
            disableNativeCache(
                 version = DisableCacheInKotlinVersion.2_3_0,
                 reason = "Cache bug",
                 issue = URI("https://youtrack.com/YY-1111")
            )
        }
    }
}
```

- `version` – the Kotlin version for which the compilation cache is disabled.
- `reason` (mandatory) – the reason why the compilation cache is disabled.
- `issue` (optional) – a URL to the corresponding issue in your bug tracker.

The new DSL replaces the deprecated `kotlin.native.cacheKind` Gradle property. You can safely remove it from your `gradle.properties` file.

For more tips on improving compilation times, see the Kotlin/Native documentation.

Experimental

New interoperability mode for C or Objective-C libraries

If you use C or Objective-C libraries in your Kotlin Multiplatform (KMP) libraries or applications, we invite you to test the new interoperability mode and share the results.

In general, Kotlin/Native enables importing C and Objective-C libraries into Kotlin. However, for KMP libraries, this functionality is currently affected by the KMP compatibility issues with older compiler versions.

In other words, if you publish a KMP library compiled with one Kotlin version, importing C or Objective-C libraries might make it impossible to use that Kotlin library in projects with an earlier Kotlin version.

To address this and other issues, the Kotlin team has been revising the interoperability mechanism used under the hood. Starting with Kotlin 2.3.20, you can try the new mode through a compiler option.

How to enable

1. In your Gradle build file, check whether you have a `cinterops {}` block or a `pod()` dependency. If these are present, your project uses C or Objective-C libraries.
2. Ensure your project uses `2.3.20` or a later version.

3. In the same build file, add the `-Xccall-mode` compiler option to the cinterop tool invocation:

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget>().configureEach {
        compilations.configureEach {
            cinterops.configureEach {
                extraOpts += listOf("-Xccall-mode", "direct")
            }
        }
    }
}
```

4. Build and test your project as usual by running unit tests, the app, and so on. You can also use the `--continue` option to allow Gradle to continue executing tasks even after failures, helping to find more problems at once.

Do **not** publish libraries compiled with the new interoperability mode yet, as it's still Experimental.

Report your results

The new interoperability mode is supposed to be a drop-in replacement in most cases. We're planning to eventually enable it by default. But to achieve that, we need to ensure it works as well as possible and test it on a wide range of projects, because:

- Some C and Objective-C declarations aren't yet supported in the new mode (mostly because of compatibility issues). We'd like to better understand the real-world impact of this and prioritize future steps accordingly.
- There may be bugs or things we haven't considered. Testing languages with numerous interacting features is challenging, and testing the interaction between languages (each with a unique set of features) is even more so.

Help us examine real-world projects and identify challenging cases. Whether you encounter any issues or not, share your results in the comments to in YouTrack.

Kotlin/Wasm

Kotlin 2.3.20 improves the performance of string operations, compilation time, and memory usage. It also adds support for the experimental `@nativeInvoke` annotation, which lets you call Kotlin objects or classes like JavaScript functions.

Improved string performance

Kotlin/Wasm now uses JS String builtins for operations on `kotlin.String` values. This allows Kotlin/Wasm to benefit from JavaScript engine string optimizations in browsers and Wasm runtimes supporting the proposal. The optimization applies to operations such as concatenation, interpolation, `StringBuilder.append()`, and number-to-string conversion.

It results in:

- Up to 4.6 times faster string interpolation in targeted benchmarks.
- Approximately 5% smaller Wasm binaries in the KotlinConf application build.
- Around 1% median improvement across all Wasm benchmarks.
- At least 20% faster `StringBuilder.append()` and concatenation of `kotlin.String` instances in append-heavy workloads.

Improved compilation time and memory optimizations

Kotlin 2.3.20 adds compiler optimizations that significantly reduce memory consumption during compilation, especially in large projects. These optimizations also improve incremental build performance.

In our testing, we observed a 65% improvement in clean build times and a 21% improvement in incremental build times.

Experimental

Support for `@nativeInvoke` annotation

Kotlin 2.3.20 introduces support for the `@nativeInvoke` annotation for the `wasmJs` target. This annotation allows you to treat a Kotlin object or a class as if it were a function in JavaScript. It's designed to mark a member function of an `external` declaration (a class or an interface) as the "invoke operator" of a JavaScript object.

When you annotate a function, every call to that function in Kotlin is translated into a direct call of the JavaScript object itself:

```
import kotlin.js.nativeInvoke

@OptIn(ExperimentalWasmJsInterop::class)
external class JsAction {
    @nativeInvoke
    operator fun invoke(data: String)
}

fun main() {
    val action = JsAction()
    action("Run task")
}
```

This is a temporary solution until stable interoperability between Kotlin/Wasm and JavaScript is designed. It may be modified or removed in future releases, and the compiler reports a warning when you use it.

For more information on the Kotlin/Wasm interoperability with JavaScript, see Interoperability with JavaScript.

Kotlin/JS

Kotlin 2.3.20 makes it possible to implement Kotlin interfaces from TypeScript and introduces experimental support for the SWC compilation platform.

Experimental

Implementing Kotlin interfaces from JavaScript/TypeScript

Kotlin 2.3.20 lifts the limitation on implementing Kotlin interfaces on the JavaScript/TypeScript side. Previously, it was only possible to export Kotlin interfaces to TypeScript as TypeScript interfaces; implementing them from TypeScript was forbidden.

Now you can implement any Kotlin interface in the following way:

```kotlin
// Kotlin
@JsExport
interface DataProcessor {
    suspend fun process(): String
}


@JsExport
fun registerProcessor(processor: DataProcessor) { ... }
```

```typescript
// TypeScript
import { DataProcessor, registerProcessor } from "my-kmp-library"


class JsonProcessor implements DataProcessor {
    readonly [DataProcessor.Symbol] = true


    async process(): Promise<string> {
        return "processed JSON data"
    }
}

registerProcessor(new JsonProcessor())
```

It's also possible to reuse Kotlin default implementations from TypeScript. Even though TypeScript has no concept of default implementations in interfaces, you can work around that by delegating to the `DefaultImpls` object:

```kotlin
// Kotlin
@JsExport
interface Logger {
    fun log(): String = "[INFO] Default log entry"
    val prefix: String get() = "LOG"
}
```

```typescript
// TypeScript
import { Logger, acceptLogger } from "my-kmp-library"

class ConsoleLogger implements Logger {
    readonly [Logger.Symbol] = true


    // Delegates to the default method implementation
    log(): string {
        return Logger.DefaultImpls.log(this);
    }

    // Delegates to the default property implementation
    get prefix(): string {
        return Logger.DefaultImpls.prefix.get(this);
    }
}

acceptLogger(new ConsoleLogger())
```

How to enable

In your build file, add the new compiler option:

```
kotlin {
    js {
        // ...
        generateTypeScriptDefinitions()
        compilerOptions {
            freeCompilerArgs.add("-Xenable-implementing-interfaces-from-typescript")
        }
    }
}
```

For more information, see `@JsExport` annotation.

Experimental

Support for SWC compilation platform

Starting with Kotlin 2.3.20, Kotlin/JS supports the SWC compilation platform. It helps transpile newer versions of JavaScript/TypeScript code into older and more compatible JavaScript code.

Delegating code conversion to an external tool allows us to reduce the number of variants produced by the Kotlin/JS compiler and speed up compiler modernization, only focusing on supporting the latest JavaScript features. Currently, the latest supported ECMAScript version is still `es2015`.

Additionally, delegating the transpilation work lets us improve the inlined JavaScript feature. Currently, it only supports ES5 syntax (this will change in 2.4.0). Supporting newer syntax while targeting lower versions would be challenging, as it would require the compiler to transpile the JS code within the inlined JS block itself. With SWC, we'll be able to add modern JS syntax, and the tool will transpile the code to the required syntax for the end-user version.

Migrating to SWC also gives you an opportunity to implement a browserlist-based DSL inside the Kotlin Gradle plugin. This allows you to declare target browsers or environments instead of specific JS versions.

How to enable

In your `gradle.properties` file, add the following option:

```
kotlin.js.delegated.transpilation=true
```

We're planning to stabilize transpilation through SWC in future Kotlin releases. After it becomes the default, the functionality of compiling multiple JS targets will be fully delegated from the Kotlin/JS compiler to the transpiler.

For more information on the SWC platform, see the official documentation.

Gradle

Kotlin 2.3.20 is compatible with new versions of Gradle and includes changes to Kotlin/JVM compilation in the Kotlin Gradle plugin.

Compatibility with Gradle 9.3.0

Kotlin 2.3.20 is fully compatible with Gradle 7.6.3 through 9.3.0. You can also use Gradle versions up to the latest Gradle release. However, be aware that doing so may result in deprecation warnings, and some new Gradle features might not work.

Improvements to binary compatibility validation in KGP

Kotlin 2.2.0 brought support for binary compatibility validation in the Kotlin Gradle plugin for the first time. Kotlin 2.3.20 adds two improvements.

First, the binary compatibility validation Gradle tasks no longer include "Legacy" in their names. We made this change because the old naming convention confused Kotlin developers:

Old name	New name
`checkLegacyAbi`	`checkKotlinAbi`
`updateLegacyAbi`	`updateKotlinAbi`
`dumpLegacyAbi`	`internalDumpKotlinAbi`

The old task names still exist in Kotlin 2.3.20 to ease the transition to the new names.

Second, if you enable binary compatibility validation in your project, Gradle now runs the `checkKotlinAbi` task automatically when you run the `check` task. Previously, Gradle didn't run the `checkKotlinAbi` task, even though the `check` task is supposed to run all verification tasks. This led to inconsistent behavior in Gradle projects.

Experimental

Kotlin/JVM compilation uses Build tools API by default

In Kotlin 2.3.20, Kotlin/JVM compilation in the Kotlin Gradle plugin uses the Build tools API (BTA) by default. This change in the internal compilation infrastructure enables faster development of build tool support for the Kotlin compiler.

If you notice any issues, share your feedback in our issue tracker.

Maven

Kotlin 2.3.20 brings an important change to make it easier to set up your Maven projects.

Simplified setup for Kotlin projects

Kotlin 2.3.20 makes it easier to set up Kotlin in Maven projects. Now Kotlin supports the automatic configuration of source roots and Kotlin's standard library.

With the new configuration, when you create a new Kotlin project with the Maven build system or introduce Kotlin to your existing Java Maven project, you don't need to manually specify source root paths or add the `kotlin-stdlib` dependency in your POM build file.

How to enable

In your `pom.xml` file, add `<extensions>true</extensions>` to the `<build><plugins>` section of the Kotlin Maven plugin:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>2.4.20</version>
            <extensions>true</extensions> <!-- Add this extension -->
        </plugin>
    </plugins>
</build>
```

The new features of the `<extensions>` option here are:

- Registering `src/main/kotlin` and `src/test/kotlin` directories as source roots in case they already exist but are not specified in the plugin configuration.
- Adding the `kotlin-stdlib` dependency in case it's not explicitly defined already.

You can also opt out of the automatic addition of Kotlin's standard library. For that, add the following to the `<properties>` section:

```
<project>
    <properties>
        <!-- Disable smart defaults via property -->
        <kotlin.smart.defaults.enabled>false</kotlin.smart.defaults.enabled>
    </properties>
</project>
```

Note that this property disables not only automatic addition of the standard library, but also the registration of source root paths. Other `<extensions>` features are not affected.

For more information on configuring Kotlin Maven projects, see Configure a Maven project.

Build tools API

Kotlin 2.3.20 introduces more changes for developers who want to integrate their build systems with the Kotlin compiler using the build tools API (BTA).

Improvements to build operations

In this release, the BTA improves how build tools manage build operations. Build operations let build tools interact with the Kotlin compiler. Each build operation is an implementation of the `BuildOperation` interface.

You can now cancel build operations that implement the `CancellableBuildOperation` interface with the `cancel()` function.

The `cancel()` function works on a "best effort" basis. This means the operation isn't guaranteed to be canceled.

For example:

```
val operation = toolchains.jvm.jvmCompilationOperationBuilder(sources, destination) {}

toolchains.createBuildSession().use {
    try {
        it.executeOperation(operation.build())
    } catch (e: OperationCancelledException) {
        println("Build operation has been cancelled.")
    }
}

// ...

// From another thread:
operation.cancel()
```

In addition, build operations are now more robust because you can create them so that they can't be changed once they start. To do this, a build tool must use the builder pattern:

1. Configure the object using a mutable builder.
2. Call the `build()` function to create an immutable instance of the object.

For example:

```
fun prepareBuildOperation(toolchains: KotlinToolchains, sources: List<Path>, destination: Path):
JvmCompilationOperation {
    val builder = toolchains.jvm.jvmCompilationOperationBuilder(sources, destination)

    // Configure the operation using the builder
    builder.compilerArguments[CommonToolArguments.VERBOSE] = true
    builder[COMPILER_ARGUMENTS_LOG_LEVEL] = CompilerArgumentsLogLevel.ERROR

    // Return an immutable operation
    return builder.build()
}
```

Consistent metric collection across build tools

Before Kotlin 2.3.20, the build metrics infrastructure was centered around Gradle, which influenced parts of the infrastructure, like metric names. In addition, not all metrics were available for different compiler execution strategies.

In Kotlin 2.3.20, the BTA provides build-tool-agnostic metric collection for the JVM. The BTA also introduces a consistent set of metrics, regardless of the compiler execution strategy. Metrics that are specific to a particular compilation approach or compiler execution strategy are reported only when applicable. For example, incremental compilation metrics are only available for incremental builds, and daemon-specific metrics are only available when using the Kotlin daemon.

Build tools can now configure a `BuildMetricsCollector` object for a build operation to capture build metrics that provide the user with insight into build performance:

```
val operation =
    kotlinToolchains.jvm.jvmCompilationOperationBuilder(sources, outputDirectory)
operation[BuildOperation.METRICS_COLLECTOR] = object : BuildMetricsCollector {
    override fun collectMetric(
        name: String,
        type: BuildMetricsCollector.ValueType,
        value: Long
    ) {
        // ...
    }
}
```

Easier configuration of compiler plugins by build tools

In Kotlin 2.3.20, the BTA provides a new and simpler way for build tools to configure compiler plugins. This approach allows build tools to propagate the configuration directly to their users.

Instead of configuring compiler plugins through the command line with experimental compiler options, build tools can use the `kotlin.buildtools.api.arguments.CommonCompilerArguments.COMPILER_PLUGINS` option to configure a list of objects that represent compiler plugin configurations:

```
import org.jetbrains.kotlin.buildtools.api.KotlinToolchains
import org.jetbrains.kotlin.buildtools.api.arguments.CompilerPlugin
import org.jetbrains.kotlin.buildtools.api.arguments.CompilerPluginOption
import
org.jetbrains.kotlin.buildtools.api.arguments.CommonCompilerArguments.Companion.COMPILER_PLUGINS
import org.jetbrains.kotlin.buildtools.api.arguments.CompilerPlugin
import org.jetbrains.kotlin.buildtools.api.arguments.CompilerPluginOption
import org.jetbrains.kotlin.buildtools.api.jvm.JvmPlatformToolchain
import org.jetbrains.kotlin.buildtools.api.jvm.JvmPlatformToolchain.Companion.jvm
import org.jetbrains.kotlin.buildtools.api.jvm.operations.JvmCompilationOperation
import java.nio.file.Path

...

val toolchains: KotlinToolchains = ...
val jvmToolchain: JvmPlatformToolchain = toolchains.jvm
val operation: JvmCompilationOperation.Builder = jvmToolchain.jvmCompilationOperationBuilder(...)
val noArgPluginClasspath: List<Path> = ...
operation.compilerArguments[COMPILER_PLUGINS] = listOf(
    CompilerPlugin(
        pluginId = "org.jetbrains.kotlin.noarg",
        classpath = noArgPluginClasspath,
        rawArguments = listOf(CompilerPluginOption("annotation", "GenerateNoArgsConstructor")),
        orderingRequirements = emptySet(),
    )
)
```

Breaking changes and deprecations

This section highlights important breaking changes and deprecations. For more information about deprecations in Kotlin 2.3.0 and 2.3.20, see the Compatibility guide.

- In Kotlin 2.3.20, Kotlin/Wasm performs module initialization as part of the Wasm module's instantiation, instead of relying on external JavaScript to call an `_initialize()` function afterward. This change makes Kotlin/Wasm more independent and prepares it for the ES module integration proposal.

If you use the `@EagerInitialization` annotation, related code may fail if it runs before module initialization completes. We recommend avoiding using the `@EagerInitialization` annotation unless you truly need it.

- Experimental context receivers are no longer supported and are replaced by context parameters.
- This release takes the next step in the deprecation cycle for Intel chip-based Apple targets. Starting with Kotlin 2.3.20, we deprecate the `macosX64`, `tvosX64`, and `watchosX64` targets. We plan to completely remove support for these targets in the next Kotlin release.

  Because many third-party libraries still rely on the `iosX64` target, we'll keep it in support tier 3 for now. This means we don't guarantee CI testing, and we may not provide source and binary compatibility between different compiler releases. For more information about support tiers, see Kotlin/Native target support.
- In Kotlin 2.3.20, stricter dependency matching in Kotlin Multiplatform can cause metadata compilation failures when dependency resolution differs between common and platform source sets. See the issue in YouTrack for details and a workaround.

Documentation updates

We made the following documentation changes in the Kotlin ecosystem:

- Kotlin roadmap – See the updated list of Kotlin's priorities on language and ecosystem evolution.
- Upgrading to AGP 9 – Explore our advice for migrating your multiplatform projects with Android apps to AGP 9.
- Configure CI for a KMP app – Follow a tutorial to configure GitHub Actions for continuous integration in a multiplatform project.
- Compose UI previews – Learn how to preview composables in the IDE without running an emulator.
- Handling web resources – Find information about how to handle web resources in Compose Multiplatform.
- Setting up the viewport – Learn how to use the `ComposeViewport()` function to render your UI on an HTML canvas with Compose Multiplatform for web.
- Custom compiler plugins – Learn how compiler plugins work and what you can do if you can't find one that fits your use case.
- Application structure – Choose the best application structure for your Ktor Server app.
- HTTP request lifecycle – Learn how to cancel request processing in Ktor when a client disconnects using the HTTP request lifecycle.
- Dependency injection – Learn how to configure dependency injection in Ktor Server, with updated guidance and practical examples.
- Exposed's Spring Boot integration – Learn how to use Exposed with Spring Boot 3 and 4.

What's new in Kotlin 2.3.0

Released: December 16, 2025

The Kotlin 2.3.0 release is out! Here are the main highlights:

- **Language**: More stable and default features, unused return value checker, explicit backing fields, and changes to context-sensitive resolution.
- **Kotlin/JVM**: Support for Java 25.

- **Kotlin/Native**: Improved interop through Swift export, faster build time for release tasks, C and Objective-C library import in Beta.
- **Kotlin/Wasm**: Fully qualified names and new exception handling proposal enabled by default, as well as new compact storage for Latin-1 characters.
- **Kotlin/JS**: New experimental suspend function export, `LongArray` representation, unified companion object access, and more.
- **Gradle**: Compatibility with Gradle 9.0 and a new API for registering generated sources.
- **Compose compiler**: Stack traces for minified Android applications.
- **Standard library**: Stable time tracking functionality and improved UUID generation and parsing.

You can also find an overview of the updates in this video:

Watch video online.

For information about the Kotlin release cycle, see Kotlin release process.

IDE support

The Kotlin plugins that support 2.3.0 are bundled in the latest versions of IntelliJ IDEA and Android Studio. You don't need to update the Kotlin plugin in your IDE. All you need to do is change the Kotlin version to 2.3.0 in your build scripts.

See Update to a new release for details.

Language

Kotlin 2.3.0 focuses on feature stabilization, introduces a new mechanism for detecting unused return values, and improves context-sensitive resolution.

Stable features

In previous Kotlin releases, several new language features were introduced as Experimental and Beta. The following features have now graduated to Stable in Kotlin 2.3.0:

- Support for nested type aliases
- Data-flow-based exhaustiveness checks for `when` expressions

Features enabled by default

In Kotlin 2.3.0, support for `return` statements in expression bodies with explicit return types is now enabled by default.

See the full list of Kotlin language features and proposals.

Experimental

Unused return value checker

Kotlin 2.3.0 introduces the unused return value checker to help prevent ignored results. It warns you whenever an expression returns a value other than `Unit` or `Nothing` and isn't passed to a function, checked in a condition, or otherwise used.

The checker helps catch bugs where a function call produces a meaningful result that's silently dropped, which can lead to unexpected behavior or hard-to-trace issues.

> The checker ignores values returned from increment operations such as `++` and `--`.

Consider the following example:

```
fun formatGreeting(name: String): String {
    if (name.isBlank()) return "Hello, anonymous user!"
    if (!name.contains(' ')) {
        // The checker reports a warning that this result is ignored
        "Hello, " + name.replaceFirstChar(Char::titlecase) + "!"
    }
    val (first, last) = name.split(' ')
    return "Hello, $first! Or should I call you Dr. $last?"
}
```

In this example, a string is created but never used, so the checker reports it as an ignored result.

This feature is Experimental. To opt in, add the following compiler option to your build file:

Gradle

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xreturn-value-checker=check")
    }
}
```

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-Xreturn-value-checker=check</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

With this option, the checker only reports ignored results from expressions that are marked, such as most functions in the Kotlin standard library.

To mark your functions, use the `@MustUseReturnValues` annotation to mark the scope on which you want the checker to report ignored return values.

For example, you can mark an entire file:

```
// Marks all functions and classes in this file so the checker reports unused return values
@file:MustUseReturnValues

package my.project

fun someFunction(): String
```

Alternatively, you can mark a specific class:

```
// Marks all functions in this class so the checker reports unused return values
@MustUseReturnValues
class Greeter {
    fun greet(name: String): String = "Hello, $name"
}

fun someFunction(): Int = ...
```

You can also mark your entire project by adding the following compiler option to your build file:

Gradle

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xreturn-value-checker=full")
    }
}
```

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-Xreturn-value-checker=full</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

With this setting, Kotlin automatically treats your compiled files as if they are annotated with `@MustUseReturnValues`, and the checker reports all return values for your project's functions.

You can suppress warnings on specific functions by marking them with the `@IgnorableReturnValue` annotation. Annotate functions where ignoring the return value is common and expected, such as `MutableList.add`:

```
@IgnorableReturnValue
fun <T> MutableList<T>.addAndIgnoreResult(element: T): Boolean {
    return add(element)
}
```

You can suppress a warning without marking the function itself as ignorable. To do this, assign the result to a special unnamed variable with an underscore (`_`):

```
// Non-ignorable function
fun computeValue(): Int = 42

fun main() {
    // Reports a warning: result is ignored
    computeValue()

    // Suppresses the warning only at this call site with a special unused variable
    val _ = computeValue()
}
```

For more information, see the feature's KEEP.

We would appreciate your feedback in YouTrack.

Experimental

Explicit backing fields

Kotlin 2.3.0 introduces explicit backing fields – a new syntax for explicitly declaring the underlying field that holds a property's value, in contrast to the existing implicit backing fields.

You can find an overview of this feature in this video:

Watch video online.

The new explicit syntax simplifies the common backing properties pattern where a property's internal type is different from its exposed API type. For example, you might use an `ArrayList` while exposing it as a read-only `List` or a `MutableList`. Previously, this required an additional private property.

With explicit backing fields, the implementation type of the `field` is directly defined within the property's scope. This eliminates the need for a separate private property and allows the compiler to automatically perform smart casting to the backing field's type within the same private scope.

Before:

```
private val _city = MutableStateFlow<String>("")
val city: StateFlow<String> get() = _city

fun updateCity(newCity: String) {
    _city.value = newCity
}
```

After:

```
val city: StateFlow<String>
    field = MutableStateFlow("")

fun updateCity(newCity: String) {
    // Smart casting works automatically
    city.value = newCity
}
```

This feature is Experimental. To opt in, add the following compiler option to your build file:

Gradle

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xexplicit-backing-fields")
    }
}
```

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-Xexplicit-backing-fields</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

For more information, see the feature's KEEP.

We would appreciate your feedback in YouTrack.

Experimental

Changes to context-sensitive resolution

Context-sensitive resolution is still Experimental, but we are continually improving the feature based on user feedback:

- The sealed and enclosing supertypes of the current type are now considered part of the contextual scope of the search. No other supertype scopes are considered. See the KT-77823 YouTrack issue for motivation and examples.
- When type operators and equalities are involved, the compiler now reports a warning if using context-sensitive resolution makes the resolution ambiguous. This can happen, for example, when a clashing declaration of a class is imported. See the KT-77821 YouTrack issue for motivation and examples.

See the full text of the current proposal in KEEP.

Kotlin/JVM: Support for Java 25

Starting with Kotlin 2.3.0, the compiler can generate classes containing Java 25 bytecode.

Kotlin/Native

Kotlin 2.3.0 introduces improvements to the Swift export support and import of C and Objective-C libraries, as well as enhanced build times for release tasks.

Experimental

Improved interop through Swift export

Kotlin 2.3.0 further improves Kotlin's interoperability with Swift through Swift export, adding support for native enum classes and variadic function parameters.

Previously, Kotlin enums were exported as ordinary Swift classes. Now the mapping is direct, and you can use regular native Swift enums. For example:

```
// Kotlin
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}

val color = Color.RED
```

```
// Swift
public enum Color: Swift.CaseIterable, Swift.LosslessStringConvertible, Swift.RawRepresentable {
    case RED, GREEN, BLUE

    var rgb: Int { get }
}
```

In addition, Kotlin's `vararg` functions are now directly mapped to Swift's variadic function parameters.

Such functions let you pass a variable number of arguments. This is useful when you don't know the number of arguments in advance or when you want to create or pass a collection without specifying its type. For example:

```
// Kotlin
fun log(vararg messages: String)
```

```
// Swift
public func log(messages: Swift.String...)
```

Generic types in variadic function parameters are not yet supported.

Beta

C and Objective-C library import is in Beta

Support for importing C and Objective-C libraries to Kotlin/Native projects is in Beta.

Full compatibility with different versions of Kotlin, dependencies, and Xcode is still not guaranteed, but the compiler now emits better diagnostics in the event of binary compatibility issues.

The import is not stable yet, and the `@ExperimentalForeignApi` opt-in annotation is still necessary when using C and Objective-C libraries in your project for certain things related to C and Objective-C interoperability, including:

- Some APIs in the `kotlinx.cinterop.*` package, required when working with native libraries or memory.
- All declarations in native libraries, except for platform libraries.

For compatibility and to prevent you from having to change your source code, the new stability status is not reflected in the annotation name.

For more information, see Stability of C and Objective-C library import.

Default explicit names in block types for Objective-C headers

Explicit parameter names in Kotlin's function types, introduced in Kotlin 2.2.20, are now the default for Objective-C headers exported from Kotlin/Native projects. These parameter names improve autocompletion suggestions in Xcode and help avoid Clang warnings.

Consider the following Kotlin code:

```
// Kotlin:
fun greetUser(block: (name: String) -> Unit) = block("John")
```

Kotlin forwards the parameter names from Kotlin function types to Objective-C block types, allowing Xcode to use them in suggestions:

```
// Objective-C:
greetUserBlock:^(NSString *name) {
    // ...
};
```

If you run into issues, you can disable explicit parameter names. To do that, add the following binary option to your `gradle.properties` file:

```
kotlin.native.binary.objcExportBlockExplicitParameterNames=false
```

Please report any problems in YouTrack.

Faster build time for release tasks

Kotlin/Native has received several performance improvements in 2.3.0. They resulted in faster build times for release tasks like `linkRelease*`, for example `linkReleaseFrameworkIosArm64`.

According to our benchmarks, release builds can be up to 40% faster, depending on the project size. These improvements are most noticeable in Kotlin Multiplatform projects targeting iOS.

For more tips on improving project compilation times, see the documentation.

Changes to Apple target support

Kotlin 2.3.0 raises the minimum supported versions of Apple targets:

- For iOS and tvOS, from 12.0 to 14.0.

- For watchOS, from 5.0 to 7.0.

According to public data, usage of older versions is already very limited. This change simplifies overall Apple target maintenance for us and opens an opportunity to support Mac Catalyst in Kotlin/Native.

If you have to keep older versions in your project, add the following lines to your build file:

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget>().configureEach {
        binaries.configureEach {
            freeCompilerArgs += "-Xoverride-konan-properties=minVersion.ios=12.0"
            freeCompilerArgs += "-Xoverride-konan-properties=minVersion.tvos=12.0"
        }
    }
}
```

Note that such a setup is not guaranteed to successfully compile and can break your app during building or at runtime.

This release also takes the next step in the deprecation cycle for Intel chip-based Apple targets.

Starting with Kotlin 2.3.0, the `macosX64`, `iosX64`, `tvosX64`, and `watchosX64` targets are demoted to support-tier 3. This means they are not guaranteed to be tested on CI, and source and binary compatibility between different compiler releases might not be provided. We plan to eventually remove support for the `x86_64` Apple targets in Kotlin 2.4.0.

For more information, see Kotlin/Native target support.

Kotlin/Wasm

Kotlin 2.3.0 enables by default fully qualified names for the Kotlin/Wasm targets, the new exception handling proposal for the `wasmWasi` target, and introduces compact storage for Latin-1 characters.

Fully qualified names enabled by default

On Kotlin/Wasm targets, fully qualified names (FQNs) were not enabled by default at runtime. You had to manually enable support for the `KClass.qualifiedName` property to use FQNs.

Only the class name (without the package) was accessible, which caused issues for code ported from the JVM to Wasm targets or for libraries that expected fully qualified names at runtime.

In Kotlin 2.3.0, the `KClass.qualifiedName` property is enabled by default on Kotlin/Wasm targets. This means that FQNs are available at runtime without any additional configuration.

Enabling FQNs by default improves code portability and makes runtime errors more informative by displaying the fully qualified name.

This change does not increase the size of the compiled Wasm binary, thanks to compiler optimizations that reduce metadata by using compact storage for Latin-1 string literals.

Compact storage for Latin-1 characters

Previously, Kotlin/Wasm stored string literal data as is, meaning every character was encoded in UTF-16. This was not optimal for text containing only, or predominantly, Latin-1 characters.

Starting with Kotlin 2.3.0, the Kotlin/Wasm compiler stores string literals containing only Latin-1 characters in UTF-8 format.

This optimization significantly reduces metadata, as experiments on JetBrains' KotlinConf application have shown. It results in:

- Up to 13% smaller Wasm binaries compared to builds without the optimization.
- Up to 8% smaller Wasm binaries even when fully qualified names are enabled, compared to earlier versions that did not store them.

This compact storage is important for web environments where download and startup times matter. Additionally, this optimization removes the size barrier that previously prevented storing fully qualified names for classes and enabling `KClass.qualifiedName` by default.

This change is enabled by default, and no further action is required.

New exception handling proposal enabled by default for `wasmWasi`

Previously, Kotlin/Wasm used the legacy exception handling proposal for all targets, including `wasmWasi`. However, most standalone WebAssembly virtual machines (VMs) are aligning with the new version of the exception handling proposal.

Starting with Kotlin 2.3.0, the new WebAssembly exception handling proposal is enabled by default for the `wasmWasi` target, ensuring better compatibility with modern WebAssembly runtimes.

For the `wasmWasi` target, the change is safe to introduce early because applications targeting it usually run in a less diverse runtime environment (often running on a single specific VM), which is typically controlled by the user, reducing the risk of compatibility issues.

The new exception handling proposal remains off by default for the `wasmJs` target. You can enable it manually by using the `-Xwasm-use-new-exception-proposal` compiler option.

Kotlin/JS

Kotlin 2.3.0 brings experimental support for exporting suspend functions to JavaScript and the `BigInt64Array` type to represent Kotlin's `LongArray` type.

With this release, you can now access companion objects inside interfaces in a unified way, use the `@JsStatic` annotation in interfaces with companion objects, the `@JsQualifier` annotation in individual functions and classes, and default exports through a new annotation, `@JsExport.Default`.

Experimental

New export of suspend function with `JsExport`

Previously, the `@JsExport` annotation did not allow exporting suspend functions (or classes and interfaces containing such functions) to JavaScript. You had to manually wrap each suspend function, which was cumbersome and prone to errors.

Starting with Kotlin 2.3.0, suspend functions can be exported directly to JavaScript using the `@JsExport` annotation.

Enabling suspend function exports reduces boilerplate and improves interoperability between Kotlin/JS and JavaScript/TypeScript (JS/TS). Kotlin's async functions can now be called directly from JS/TS without extra code.

To enable this feature, add the following compiler option to your `build.gradle.kts` file:

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xenable-suspend-function-exporting")
    }
}
```

Once enabled, classes and functions marked with the `@JsExport` annotation can include suspend functions without additional wrappers.

They can be consumed as regular JavaScript async functions and can be overridden as an async function, as well:

```
@JsExport
open class Foo {
    suspend fun foo() = "Foo"
}
```

```
class Bar extends Foo {
    override async foo(): Promise<string> {
        return "Bar"
    }
}
```

This feature is Experimental. We would appreciate your feedback in our issue tracker, YouTrack.

Experimental

Usage of the `BigInt64Array` type to represent Kotlin's `LongArray` type

Previously, Kotlin/JS represented its `LongArray` as a JavaScript `Array<bigint>`. This approach worked but was not ideal for interoperability with JavaScript APIs that expect typed arrays.

Starting with this release, Kotlin/JS now uses JavaScript's built-in `BigInt64Array` type to represent Kotlin's `LongArray` values when compiling to JavaScript.

Using `BigInt64Array` simplifies interop with JavaScript APIs that use typed arrays. It also allows APIs that accept or return `LongArray` to be exported more naturally from Kotlin to JavaScript.

To enable this feature, add the following compiler option to your `build.gradle.kts` file:

```
kotlin {
    js {
        // ...
        compilerOptions {
            freeCompilerArgs.add("-Xes-long-as-bigint")
        }
    }
}
```

This feature is Experimental. We would appreciate your feedback in our issue tracker, YouTrack.

Unified companion object access across JS module systems

Previously, when you exported a Kotlin interface with a companion object to JavaScript/TypeScript using the `@JsExport` annotation, consuming the interface in TypeScript worked differently for ES modules compared to other module systems.

As a result, you had to adjust the consumption of the output on the TypeScript side, depending on the module system.

Consider this Kotlin code:

```
@JsExport
interface Foo {
    companion object {
        fun bar() = "OK"
    }
}
```

You had to call it differently depending on the module system:

```
// Worked for CommonJS, AMD, UMD, and no modules
Foo.bar()

// Worked for ES modules
Foo.getInstance().bar()
```

In this release, Kotlin unifies companion-object exports across all JavaScript module systems.

Now, for every module system (ES modules, CommonJS, AMD, UMD, no modules), companion objects inside interfaces are always accessed the same way (just like companions in classes):

```
// Works for all module systems
Foo.Companion.bar()
```

This improvement also fixes collection interoperability. Before, collection factory functions had to be accessed differently depending on the module system:

```
// Worked for CommonJS, AMD, UMD, and no modules
KtList.fromJsArray([1, 2, 3])

// Worked for ES modules
KtList.getInstance().fromJsArray([1, 2, 3])
```

Now, accessing collection factory functions is similar across all the module systems:

```
// Works for all module systems
KtList.fromJsArray([1, 2, 3])
```

This change reduces inconsistent behavior between module systems and avoids bugs and interoperability issues.

This feature is enabled by default.

Support for `@JsStatic` annotations in interfaces with companion objects

Previously, the `@JsStatic` annotation was not allowed inside exported interfaces with companion objects.

For example, the following code would produce an error because only members of class companion objects can be annotated with `@JsStatic`:

```
@JsExport
interface Foo {
    companion object {
        @JsStatic // Error
        fun bar() = "OK"
    }
}
```

In this case, you had to drop the `@JsStatic` annotation and access the companion from JavaScript (JS) in the following way:

```
// For all module systems
Foo.Companion.bar()
```

Now, the `@JsStatic` annotation is supported in interfaces with companion objects. You can use this annotation on such companions and call the function directly from JS, just as you would for classes:

```
// For all module systems
Foo.bar()
```

This change simplifies API consumption in JS, allows static factory methods on interfaces, and removes inconsistencies between classes and interfaces.

This feature is enabled by default.

`@JsQualifier` annotation allowed in individual functions and classes

Previously, you could only apply the `@JsQualifier` annotation at the file level, requiring all external JavaScript (JS) declarations to be placed in separate files.

Starting from Kotlin 2.3.0, you can apply the `@JsQualifier` annotation directly to individual functions and classes, just like the `@JsModule` and `@JsNonModule` annotations.

For example, you can now write the following external function code next to regular Kotlin declarations in the same file:

```
@JsQualifier("jsPackage")
private external fun jsFun()
```

This change simplifies Kotlin/JS interop, keeps your project structure cleaner, and aligns Kotlin/JS with how other platforms handle external declarations.

This feature is enabled by default.

Support for JavaScript default exports

Previously, Kotlin/JS could not generate JavaScript's default exports from Kotlin code. Instead, Kotlin/JS only generated named exports, for example:

```
export { SomeDeclaration };
```

If you required a default export, you had to use workarounds inside the compiler, such as placing the `@JsName` annotation with `default` and a space as an argument:

```
@JsExport
@JsName("default ")
class SomeDeclaration
```

Kotlin/JS now supports default exports directly through a new annotation:

```
@JsExport.Default
```

When you apply this annotation to a Kotlin declaration (class, object, function, or property), the generated JavaScript automatically includes an `export default` statement for ES modules:

```
export default HelloWorker;
```

> For module systems different from ES modules, the new `@JsExport.Default` annotation works similarly to the regular `@JsExport` annotation.

This change enables Kotlin code to conform to JavaScript conventions and is particularly important for platforms like Cloudflare Workers, or frameworks like `React.lazy`.

This feature is enabled by default. You only need to use the `@JsExport.Default` annotation.

Gradle

Kotlin 2.3.0 is fully compatible with Gradle 7.6.3 through 9.0.0. You can also use Gradle versions up to the latest Gradle release. However, be aware that doing so may result in deprecation warnings, and some new Gradle features might not work.

In addition, the minimum supported Android Gradle plugin version is now 8.2.2, and the maximum supported version is 8.13.0.

Kotlin 2.3.0 also introduces a new API for registering generated sources in your Gradle projects.

Experimental

New API for registering generated sources in Gradle projects

Kotlin 2.3.0 introduces a new Experimental API in the `KotlinSourceSet` interface, which you can use to register generated sources in your Gradle projects.

This new API is a quality-of-life improvement that helps IDEs distinguish between generated code and regular source files. The API allows IDEs to highlight the generated code differently in the UI and to trigger generation tasks when the project is imported. We are currently working on adding this support in IntelliJ IDEA. The API is also especially useful for third-party plugins or tools that generate code, such as KSP (Kotlin Symbol Processing).

For more information, see Register generated sources.

Standard library

Kotlin 2.3.0 stabilizes the new time tracking functionality, `kotlin.time.Clock` and `kotlin.time.Instant`, and adds several improvements to the Experimental UUID API.

Experimental

Improved UUID generation and parsing

Kotlin 2.3.0 introduces several improvements for the UUID API, including:

- Support for returning `null` when parsing invalid UUIDs
- New functions to generate v4 and v7 UUIDs
- Support for generating v7 UUIDs for specific timestamps

UUID support in the standard library is Experimental but planned to be stabilized in the future. To opt in, use the `@OptIn(ExperimentalUuidApi::class)` annotation or add the following compiler option to your build file:

Gradle

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-opt-in=kotlin.uuid.ExperimentalUuidApi")
    }
}
```

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <configuration>
                <args>
                    <arg>-opt-in=kotlin.uuid.ExperimentalUuidApi</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

We would appreciate your feedback in YouTrack or in the related Slack channel.

Support for returning `null` when parsing invalid UUIDs

Kotlin 2.3.0 introduces new functions to create a `Uuid` instance from a string, which return `null` instead of throwing an exception if the string isn't a valid UUID.

These functions include the following:

- `Uuid.parseOrNull()` – parses UUIDs in either hex-and-dash or hexadecimal format.
- `Uuid.parseHexDashOrNull()` – parses UUIDs only in hex-and-dash format, and returns `null` otherwise.
- `Uuid.parseHexOrNull()` – parses UUIDs only in plain hexadecimal format, and returns `null` otherwise.

Here's an example:

```
import kotlin.uuid.ExperimentalUuidApi
import kotlin.uuid.Uuid

@OptIn(ExperimentalUuidApi::class)
fun main() {
    val valid = Uuid.parseOrNull("550e8400-e29b-41d4-a716-446655440000")
    println(valid)
    // 550e8400-e29b-41d4-a716-446655440000

    val invalid = Uuid.parseOrNull("not-a-uuid")
    println(invalid)
    // null


    val hexDashValid = Uuid.parseHexDashOrNull("550e8400-e29b-41d4-a716-446655440000")
    println(hexDashValid)
    // 550e8400-e29b-41d4-a716-446655440000


    val hexDashInvalid = Uuid.parseHexDashOrNull("550e8400e29b41d4a716446655440000")
    println(hexDashInvalid)
    // null
}
```

New functions to generate v4 and v7 UUIDs

Kotlin 2.3.0 introduces two new functions for generating UUIDs: `Uuid.generateV4()` and `Uuid.generateV7()`.

Use the `Uuid.generateV4()` function to generate version 4 UUIDs, or the `Uuid.generateV7()` function to generate version 7 UUIDs.

> The `Uuid.random()` function remains unchanged and still generates version 4 UUIDs, just like `Uuid.generateV4()`.

Here's an example:

```
import kotlin.uuid.ExperimentalUuidApi
import kotlin.uuid.Uuid

@OptIn(ExperimentalUuidApi::class)
fun main() {
    // Generates a v4 UUID
    val v4 = Uuid.generateV4()
    println(v4)

    // Generates a v7 UUID
    val v7 = Uuid.generateV7()
    println(v7)

    // Generates a v4 UUID
    val random = Uuid.random()
    println(random)
}
```

Support for generating v7 UUIDs for specific timestamps

Kotlin 2.3.0 introduces the new `Uuid.generateV7NonMonotonicAt()` function, which you can use to generate a version 7 UUID for a specific moment in time.

> Unlike `Uuid.generateV7()`, `Uuid.generateV7NonMonotonicAt()` doesn't guarantee monotonic ordering, so multiple UUIDs created for the same timestamp might not be sequential.

Use this function when you need identifiers tied to a known timestamp, such as when recreating event IDs or generating database entries that reflect when something originally occurred.

For example, to create a version 7 UUID for a particular instant, use the following code:

```
import kotlin.uuid.ExperimentalUuidApi
import kotlin.uuid.Uuid
import kotlin.time.ExperimentalTime
import kotlin.time.Instant

@OptIn(ExperimentalUuidApi::class, ExperimentalTime::class)
fun main() {
    val timestamp = Instant.fromEpochMilliseconds(1577836800000) // 2020-01-01T00:00:00Z

    // Generates a v7 UUID for the specified timestamp (without monotonicity guarantees)
    val v7AtTimestamp = Uuid.generateV7NonMonotonicAt(timestamp)
    println(v7AtTimestamp)
}
```

Compose compiler: Stack traces for minified Android applications

Starting from Kotlin 2.3.0, the compiler outputs ProGuard mappings for Compose stack traces when applications are minified by R8. This expands the experimental stack traces feature that was previously only available in debuggable variants.

The release variant of stack traces contains group keys that can be used to identify composable functions in minified applications without the overhead of recording source information at runtime. The group key stack traces require your application to be built with the Compose runtime 1.10 or newer.

To enable group key stack traces, add the following line before initializing any `@Composable` content:

```
Composer.setDiagnosticStackTraceMode(ComposeStackTraceMode.GroupKeys)
```

With these stack traces enabled, the Compose runtime will append its own stack trace after a crash is captured during composition, measure, or draw passes, even when the app is minified:

```
java.lang.IllegalStateException: <message>
        at <original trace>
    Suppressed: androidx.compose.runtime.DiagnosticComposeException: Composition stack when thrown:
        at $$compose.m$123(SourceFile:1)
        at $$compose.m$234(SourceFile:1)
        ...
```

Stack traces produced by Jetpack Compose 1.10 in this mode only contain group keys that still have to be deobfuscated. This is addressed in the Kotlin 2.3.0 release with the Compose Compiler Gradle plugin, which now appends group key entries to the ProGuard mapping files produced by R8. If you see new warnings in cases when the compiler fails to create mappings for some functions, please report them to the Google IssueTracker.

The Compose Compiler Gradle plugin only creates deobfuscation mappings for the group key stack traces when R8 is enabled for the build due to dependencies on R8 mapping files.

By default, the mapping file Gradle tasks run regardless of whether you enable the traces. If they cause problems in your build, you can disable the feature entirely. Add the following property in the `composeCompiler {}` block of your Gradle configuration:

```
composeCompiler {
    includeComposeMappingFile.set(false)
}
```

There is a known issue with some code not showing up in stack traces for project files supplied by the Android Gradle plugin: KT-83099.

Please report any problems encountered to the Google IssueTracker.

Breaking changes and deprecations

This section highlights important breaking changes and deprecations. For a complete overview, see our Compatibility guide.

- Starting with Kotlin 2.3.0, the compiler no longer supports `-language-version=1.8`. There's also no support for `-language-version=1.9` on non-JVM platforms.
- Language feature sets older than 2.0 (excluding 1.9 for the JVM platform) aren't supported, but the language itself remains fully backward-compatible with Kotlin 1.0.

  If you use both the `kotlin-dsl` **and** the `kotlin("jvm")` plugin in your Gradle project, you may see a Gradle warning about an unsupported Kotlin plugin version. See our compatibility guide for guidance on migration steps.
- In Kotlin Multiplatform, support for the Android target is now available through Google's `com.android.kotlin.multiplatform.library` plugin. Migrate your projects with Android targets to the new plugin and rename your `androidTarget` blocks to `android`.
- If you continue using the Kotlin Multiplatform Gradle plugin for Android targets with Android Gradle plugin (AGP) 9.0.0 or later, you see a configuration error when using the `androidTarget` block, along with diagnostic messages that provide guidance on how to migrate. You can avoid this error by using AGP 8.x and updating to Kotlin 2.3.10, or by migrating to Google's plugin for Android targets.
- AGP 9.0.0 includes built-in support for Kotlin. Starting with Kotlin 2.3.0, you see a configuration error if you use this version of AGP with the `kotlin-android` plugin, because the plugin is no longer necessary. New diagnostic messages are available to help you migrate. If you use older AGP versions, you see a deprecation warning.
- Support for the Ant build system is no longer available.

Documentation updates

Kotlin Multiplatform documentation has moved to kotlinlang.org. Now you can switch between Kotlin and KMP docs in one place. We've also refreshed the table of contents for the language guide and introduced new navigation.

Other notable changes since the last Kotlin release:

- KMP overview – explore the Kotlin Multiplatform ecosystem on a single page.
- Kotlin Multiplatform quickstart – learn how to set up an environment with the KMP IDE plugin.
- What's new in Compose Multiplatform 1.9.3 – learn about the highlights from the latest release.
- Get started with Kotlin/JS – create a web application for the browser using Kotlin/JavaScript.
- Classes – learn the basics and best practices of using classes in Kotlin.
- Extensions – learn how you can extend classes and interfaces in Kotlin.
- Coroutines basics – explore key coroutine concepts and learn how to create your first coroutines.
- Cancellation and timeouts – learn how coroutine cancellation works and how to make coroutines respond to cancellation.
- Kotlin/Native libraries – see how to produce `klib` library artifacts.
- Kotlin Notebook overview – create interactive notebook documents with the Kotlin Notebook plugin.
- Add Kotlin to a Java project – configure a Java project to use both Kotlin and Java.
- Test Java code with Kotlin – test a mixed Java-Kotlin project with JUnit.
- New case studies page – discover how different companies apply Kotlin.

How to update to Kotlin 2.3.0

The Kotlin plugin is distributed as a bundled plugin in IntelliJ IDEA and Android Studio.

To update to the new Kotlin version, change the Kotlin version to 2.3.0 in your build scripts.

Compatibility guide for Kotlin 2.3.x

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 2.2 to Kotlin 2.3. This document also includes information about tool-related changes.

Basic terms

In this document, we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Drop support in `-language-version` for 1.8 and 1.9

Issue: KT-76343, KT-76344.

Component: Compiler

Incompatible change type: source

Short summary: Starting with Kotlin 2.3, the compiler no longer supports `-language-version=1.8`. Support for `-language-version=1.9` is also removed for non-JVM platforms.

Deprecation cycle:

- 2.2.0: report a warning when using `-language-version` with versions 1.8 and 1.9
- 2.3.0: raise the warning to an error for `-language-version` with version 1.8 on all platforms and for version 1.9 on non-JVM platforms

Report upper-bound constraint violation errors for inferred types with typealiases

Issue: KTLC-287

Component: Core language

Incompatible change type: source

Short summary: Previously, the compiler never reported errors about upper-bound violation constraints for the inferred types. This has been fixed in Kotlin 2.3.0 so that the error is reported consistently across all type parameters.

Deprecation cycle:

- 2.2.20: report deprecation warnings for bounds violations by implicit type arguments
- 2.3.0: raise the warning to an error for `UPPER_BOUND_VIOLATED` on implicit type arguments

Prohibit `@JvmSerializableLambda` annotation on `inline` and `crossinline` lambdas

Issue: KTLC-9

Component: Core language

Incompatible change type: source

Short summary: You can no longer apply the `@JvmSerializableLambda` annotation to `inline` or `crossinline` lambdas. These lambdas aren't serializable, so applying `@JvmSerializableLambda` had no effect.

Deprecation cycle:

- 2.1.20: report a warning when the `@JvmSerializableLambda` is applied to `inline` and `crossinline` lambdas
- 2.3.0: raise the warning to an error; this change can be enabled in progressive mode

Prohibit delegating a Kotlin interface to a Java class when the generic signatures don't match

Issue: KTLC-267

Component: Core language

Incompatible change type: source

Short summary: Kotlin 2.3.0 forbids delegation to a Java class that implements a generic interface method with a non-generic override. Previously, allowing this behavior led to type mismatches and `ClassCastException` reported at runtime. This change shifts the error from runtime to compile time.

Deprecation cycle:

- 2.1.20: report a warning
- 2.3.0: raise the warning to an error

Deprecate use of `return` in expression-bodied functions without explicit return type

Issue: KTLC-288

Component: Core language

Incompatible change type: source

Short summary: Kotlin now deprecates using `return` inside expression bodies when the function's return type isn't explicitly declared.

Deprecation cycle:

- 2.3.0: report a warning
- 2.4.0: raise the warning to an error

Prohibit inheritance from nullable supertypes introduced via typealias

Issue: KTLC-279

Component: Core language

Incompatible change type: source

Short summary: Kotlin now reports an error when attempting to inherit from a nullable typealias, which is consistent with how it already handles direct nullable supertypes.

Deprecation cycle:

- 2.2.0: report a warning
- 2.3.0: raise the warning to an error

Unify generic signature generation for top-level lambdas and call arguments

Issue: KTLC-277

Component: Reflection

Incompatible change type: behavioral

Short summary: Kotlin 2.3.0 uses the same type-checking logic for top-level lambdas as it does for lambdas passed as call arguments, ensuring consistent generic signature generation across both cases.

Deprecation cycle:

- 2.3.0: introduce the new behavior; not applicable in progressive mode

Prohibit reified type parameters from being inferred as intersection types

Issue: KTLC-13

Component: Core language

Incompatible change type: source

Short summary: Kotlin 2.3.0 prohibits situations where a reified type parameter is inferred to an intersection type, as this could lead to incorrect runtime behavior.

Deprecation cycle:

- 2.1.0: report a warning when a reified type parameter is inferred as an intersection type
- 2.3.0: raise the warning to an error

Prohibit exposing less-visible types through type parameter bounds

Issue: KTLC-275

Component: Core language

Incompatible change type: source

Short summary: Kotlin 2.3.0 forbids using type parameter bounds that expose types with more restrictive visibility than the function or declaration itself, aligning the rules for functions with those already applied to classes.

Deprecation cycle:

- 2.1.0: report a warning on the problematic type parameter bound
- 2.3.0: raise the warning to an error

Standard library

Deprecate Char-to-number conversions and introduce explicit digit and code APIs

Issue: KTLC-321

Component: kotlin-stdlib

Incompatible change type: source

Short summary: Kotlin 2.3.0 deprecates `Char.toX()` and `X.toChar()` conversions for numeric types and introduces new, explicit APIs for accessing a character's code and digit value.

Deprecation cycle:

- 1.4.30: introduce new functions as Experimental
- 1.5.0: promote the new functions to Stable; report warnings for old functions with suggestions for replacements
- 2.3.0: raise the warnings to errors

Deprecate `Number.toChar()` function

Issue: KT-56822

Component: kotlin-stdlib

Incompatible change type: source

Short summary: The `Number.toChar()` function is deprecated. Use `toInt().toChar()` or the `Char` constructor instead.

Deprecation cycle:

- 1.9.0: report a warning when using the `Number.toChar()` function
- 2.3.0: raise the warning to an error

Deprecate `String.subSequence(start, end)` function

Issue: KTLC-282

Component: kotlin-stdlib

Incompatible change type: source

Short summary: The `String.subSequence(start, end)` function is deprecated. Use the `String.subSequence(startIndex, endIndex)` function instead.

Deprecation cycle:

- 1.0: report a warning when using `String.subSequence(start, end)`
- 2.3.0: raise the warning to an error

Deprecate `kotlin.io.createTempDirectory()` and `kotlin.io.createTempFile()` functions

Issue: KTLC-281

Component: kotlin-stdlib

Incompatible change type: source

Short summary: The `kotlin.io.createTempDirectory()` and `kotlin.io.createTempFile()` functions are deprecated. Use the `kotlin.io.path.createTempDirectory()` and `kotlin.io.path.createTempFile()` functions instead.

Deprecation cycle:

- 1.4.20: report a warning when using `kotlin.io.createTempDirectory()` and `kotlin.io.createTempFile()` functions
- 2.3.0: raise the warning to an error

Hide `InputStream.readBytes(Int)` function

Issue: KTLC-280

Component: kotlin-stdlib

Incompatible change type: source

Short summary: After being deprecated for a long time, the `InputStream.readBytes(estimatedSize: Int = DEFAULT_BUFFER_SIZE): ByteArray` function is now hidden.

Deprecation cycle:

- 1.3.0: report a warning
- 1.5.0: raise the warning to an error
- 2.3.0: hide the function

Unify Kotlin/Native stack trace printing with other platforms

Issue: KT-81431

Component: Kotlin/Native

Incompatible change type: behavioral

Short summary: When formatting an exception stack trace, additional causes aren't printed if the same exception cause has already been printed.

Deprecation cycle:

- 2.3.20: Unify Kotlin/Native exception stack trace formatting with other Kotlin platforms

Correct `Iterable<T>.intersect()` and `Iterable<T>.subtract()` behavior

Issue: KTLC-268

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: The `Iterable<T>.intersect()` and `Iterable<T>.subtract()` functions now test membership for each receiver element before adding it to the result set. The result set compares elements using `Any::equals`, ensuring correct results even when the argument collection uses referential equality (for example, `IdentityHashMap.keys`).

Deprecation cycle:

- 2.3.0: enable the new behavior

Tools

Unsupported KGP version warning when using `kotlin-dsl` and `kotlin("jvm")` plugins

Issue: KT-79851

Component: Gradle

Incompatible change type: behavioral

Short summary: In Kotlin 2.3, if you use both the `kotlin-dsl` **and** the `kotlin("jvm")` plugin in your Gradle project, you may see a Gradle warning about an unsupported Kotlin Gradle plugin (KGP) version.

Migration steps:

In general, we don't recommend using both the `kotlin-dsl` and the `kotlin("jvm")` plugins in the same Gradle project. This setup isn't supported.

For convention plugins, precompiled script plugins or any other form of unpublished build logic, you have three options:

1. Don't apply the `kotlin("jvm")` plugin explicitly. Instead, let the `kotlin-dsl` plugin automatically provide a compatible KGP version.
2. If you want to apply the `kotlin("jvm")` plugin explicitly, use the `embeddedKotlinVersion` constant to specify the embedded Kotlin version.

   To upgrade the embedded Kotlin and language versions, update your Gradle version. You can find compatible Gradle versions in Gradle's Compatibility Notes for Kotlin.
3. Don't use the `kotlin-dsl` plugin. This may be more appropriate for binary plugins that aren't tied to a specific Gradle version.

As a last resort, you can configure your project to use language version 2.1 or higher, which overrides the conflicting behavior of the `kotlin-dsl` plugin. However, we strongly recommend not doing so.

If you experience difficulties during migration, reach out in the #gradle channel in our Slack for support.

Deprecation cycle:

- 2.3.0: introduce a diagnostic that detects when the `kotlin-dsl` plugin is used with an incompatible language or API version of the compiler

Deprecate `kotlin-android` plugin for AGP versions 9.0.0 and later

Issue: KT-81199

Component: Gradle

Incompatible change type: source

Short summary: In Kotlin 2.3.0, the `org.jetbrains.kotlin.android` plugin is deprecated when using Android Gradle plugin (AGP) versions 9.0.0 or later. Starting with AGP 9.0.0, AGP provides built-in support for Kotlin, so the `kotlin-android` plugin is no longer required.

Deprecation cycle:

- 2.3.0: report a warning when the `kotlin-android` plugin is used with AGP versions 9.0.0 or later, and both the `android.builtInKotlin` and `android.newDsl=false` Gradle properties are set to `false`

Deprecate `testApi` configuration

Issue: KT-63285

Component: Gradle

Incompatible change type: source

Short summary: Kotlin 2.3.0 deprecates the `testApi` configuration. This configuration exposed test dependencies and sources to other modules, but Gradle doesn't support this behavior.

Migration options: Replace any instances of `testApi()` with `testImplementation()`, and do the same for other variants. For example, replace `kotlin.sourceSets.commonTest.dependencies.api()` with `kotlin.sourceSets.commonTest.dependencies.implementation()`.

For Kotlin/JVM projects, consider using Gradle's test fixtures instead. If you'd like to see support for test fixtures in multiplatform projects, share your use case in YouTrack.

Deprecation cycle:

- 2.3.0: report a warning

Deprecate `createTestExecutionSpec()` function

Issue: KT-75449

Component: Gradle

Incompatible change type: source

Short summary: Kotlin 2.3.0 deprecates the `createTestExecutionSpec()` function in the `KotlinJsTestFramework` interface since it is no longer used.

Deprecation cycle:

- 2.2.20: report a warning
- 2.3.0: raise the warning to an error
- 2.4.0: remove the function

Remove `closureTo()`, `createResultSet()`, and `KotlinToolingVersionOrNull()` functions

Issue: KT-64273

Component: Gradle

Incompatible change type: source

Short summary: Kotlin 2.3.0 removes the `closureTo()`, `createResultSet()` functions from the `closure` DSL since they are no longer used. In addition, the `KotlinToolingVersionOrNull()` function is removed. Use the `KotlinToolingVersion()` function instead.

Deprecation cycle:

- 1.7.20: report an error
- 2.3.0: remove the functions

Deprecate the `ExtrasProperty` API

Issue: KT-74915

Component: Gradle

Incompatible change type: source

Short summary: The `ExtrasProperty` API, which has been deprecated since Kotlin 2.0.0, is now internalized in Kotlin 2.3.0. Use Gradle's `ExtraPropertiesExtension` API as an alternative.

Deprecation cycle:

- 2.0.0: report a warning
- 2.1.0: raise the warning to an error
- 2.3.0: make the API internal

Deprecate `HasKotlinDependencies` in `KotlinCompilation`

Issue: KT-67290

Component: Gradle

Incompatible change type: source

Short summary: Kotlin 2.3.0 deprecates the `HasKotlinDependencies` interface in `KotlinCompilation`. Dependency-related APIs are now exposed through the `KotlinSourceSet` interface instead.

Deprecation cycle:

- 2.3.0: report a warning

Deprecate npm and Yarn package manager internal functions and properties

Issue: KT-81009

Component: Gradle

Incompatible change type: source

Short summary: The following functions and properties related to the npm and Yarn package managers are deprecated:

- `CompositeDependency.dependencyName`, `CompositeDependency.dependencyVersion`, `CompositeDependency.includedBuildDir`.
- `KotlinNpmInstallTask.Companion.NAME`.
- `LockCopyTask.Companion.STORE_PACKAGE_LOCK_NAME`, `LockCopyTask.Companion.RESTORE_PACKAGE_LOCK_NAME`, `LockCopyTask.Companion.UPGRADE_PACKAGE_LOCK`.
- `Npm.npmExec()`.
- `NpmProject.require()`, `NpmProject.useTool()`.
- `PublicPackageJsonTask.jsIrCompilation`.
- `YarnBasics.yarnExec()`.
- `YarnPlugin.Companion.STORE_YARN_LOCK_NAME`, `YarnPlugin.Companion.RESTORE_YARN_LOCK_NAME`, `YarnPlugin.Companion.UPGRADE_YARN_LOCK`.
- `YarnSetupTask.Companion.NAME`.

Deprecation cycle:

- 2.2.0 and 2.2.20: report warnings when using these functions or properties
- 2.3.0: raise the warnings to errors
- 2.4.0: remove the functions and properties

Deprecate support for PhantomJS

Issue: KT-76019

Component: Gradle

Incompatible change type: source

Short summary: Since PhantomJS is no longer maintained, Kotlin 2.3.0 deprecates the `karmaPhantomjsLauncher` property in the `NpmVersions` API.

Deprecation cycle:

- 2.3.0: report a warning

Prohibit subclassing of classes that set up test runs or JavaScript runtime

Issue: KT-75869, KT-81007

Component: Gradle

Incompatible change type: source

Short summary: Kotlin 2.3.0 prohibits subclassing the following classes:

- `KotlinTest`
- `KotlinNativeTest`
- `KotlinJsTest`
- `KotlinJsIrTarget`
- `KotlinNodeJsIr`
- `KotlinD8Ir`
- `KotlinKarma`
- `KotlinMocha`
- `KotlinWebpack`
- `TypeScriptValidationTask`
- `YarnRootExtension`

These classes were never intended to be subclassed. All use cases for subclassing should now be covered by the configuration blocks provided by the Kotlin Gradle plugin DSL. If the existing APIs for these tasks don't meet your needs for setting up test runs or the JavaScript runtime, share your feedback in YouTrack.

Deprecation cycle:

- 2.2.0: report a warning for code that creates subclasses from these classes
- 2.3.0: raise the warnings to errors
- 2.4.0: remove the APIs

Deprecate `ExperimentalWasmDsl` annotation class

Issue: KT-81005

Component: Gradle

Incompatible change type: source

Short summary: The `ExperimentalWasmDsl` annotation class is deprecated since the functionality has moved to the `kotlin-plugin-annotations` module.

Deprecation cycle:

- 2.0.20: report a warning
- 2.3.0: raise the warning to an error
- 2.4.0: remove the annotation class

Deprecate `ExperimentalDceDsl` annotation class

Issue: KT-81008

Component: Gradle

Incompatible change type: source

Short summary: The `ExperimentalDceDsl` annotation class isn't used anymore, so it's been deprecated.

Deprecation cycle:

- 2.2.0: report a warning
- 2.3.0: raise the warning to an error
- 2.4.0: remove the annotation class

Deprecate JavaScript utilities

Issue: KT-81010

Component: Gradle

Incompatible change type: source

Short summary: The following functions and properties are only used internally, so they've been deprecated:

- `JsIrBinary.generateTs`
- `KotlinJsIrLink.mode`
- `NodeJsSetupTask.Companion.NAME`
- `Appendable.appendConfigsFromDir()`
- `ByteArray.toHex()`
- `FileHasher.calculateDirHash()`
- `String.jsQuoted()`

Deprecation cycle:

- 2.2.0: report a warning when the `KotlinJsIrLink.mode` property is used
- 2.2.0: report a warning when the `NodeJsSetupTask.Companion.NAME` property and functions are used
- 2.2.20: report a warning when the `JsIrBinary.generateTs` property is used
- 2.3.0: raise the warnings to errors
- 2.4.0: remove the APIs

Deprecate migrated D8 and Binaryen properties

Issue: KT-81006

Component: Gradle

Incompatible change type: source

Short summary: The following properties are deprecated because they've been migrated from the `org.jetbrains.kotlin.gradle.targets.js` package to the `org.jetbrains.kotlin.gradle.targets.wasm` package:

- `binaryen.BinaryenEnvSpec`
- `binaryen.BinaryenExtension`
- `binaryen.BinaryenPlugin`
- `binaryen.BinaryenRootPlugin`
- `BinaryenSetupTask.Companion.NAME`
- `d8.D8EnvSpec`
- `d8.D8Plugin`
- `D8SetupTask.Companion.NAME`

Deprecation cycle:

- 2.2.0: report a warning
- 2.3.0: raise the warning to an error
- 2.4.0: remove the properties

Deprecate `create()` function in `NodeJsExec` DSL

Issue: KT-81004

Component: Gradle

Incompatible change type: source

Short summary: The `create()` function in the companion object of the `NodeJsExec` DSL is deprecated. Use the `register()` function instead.

Deprecation cycle:

- 2.1.20: report a warning
- 2.3.0: raise the warning to an error
- 2.4.0: remove the function

Deprecate properties in `kotlinOptions` DSL

Issue: KT-76720

Component: Gradle

Incompatible change type: source

Short summary: The ability to configure compiler options through the `kotlinOptions` DSL and the related `KotlinCompile<KotlinOptions>` task interface has been deprecated in favor of the new `compilerOptions` DSL since Kotlin 2.2.0. Kotlin 2.3.0 continues the deprecation cycle for all properties in the `kotlinOptions` interface. To migrate, use the `compilerOptions` DSL to configure compiler options. For guidance on the migration, see Migrate from `kotlinOptions {}` to `compilerOptions {}`.

Deprecation cycle:

- 2.0.0: report a warning for `kotlinOptions` DSL
- 2.2.0: raise the warning to an error and deprecate all properties in `kotlinOptions`
- 2.3.0: raise the warning to an error for all properties in `kotlinOptions`

Deprecate `kotlinArtifacts` API

Issue: KT-77066

Component: Gradle

Incompatible change type: source

Short summary: The experimental `kotlinArtifacts` API is deprecated. Use the current DSL available in the Kotlin Gradle plugin to build final native binaries. If it's not sufficient for migration, leave a comment in this YouTrack issue.

Deprecation cycle:

- 2.2.0: report a warning when the `kotlinArtifacts` API is used
- 2.3.0: raise this warning to an error
- 2.4.0: remove the API

Remove `kotlin.mpp.resourcesResolutionStrategy` Gradle property

Issue: KT-74955

Component: Gradle

Incompatible change type: source

Short summary: Previously the `kotlin.mpp.resourcesResolutionStrategy` Gradle property was deprecated because it wasn't used. In Kotlin 2.3.0, the Gradle property is removed completely.

Deprecation cycle:

- 2.2.0: report a configuration-time diagnostic
- 2.3.0: remove the Gradle property

Deprecated old mode of multiplatform IDE import

Issue: KT-61127

Component: Gradle

Incompatible change type: source

Short summary: Before Kotlin 2.3.0, we supported multiple modes of multiplatform IDE import. Now, the older mode is deprecated, leaving only one mode available. Previously, the old mode was enabled using the `kotlin.mpp.import.enableKgpDependencyResolution=false` Gradle property. Using this property now triggers a deprecation warning.

Deprecation cycle:

- 2.3.0: report a warning when the `kotlin.mpp.import.enableKgpDependencyResolution=false` Gradle property is used

Remove properties to disable precise compilation backup

Issue: KT-81038

Component: Gradle

Incompatible change type: source

Short summary: Kotlin 1.9.0 introduced an experimental optimization for incremental compilation called precise compilation backup. After successful testing, this optimization was enabled by default in Kotlin 2.0.0. Kotlin 2.3.0 removes the `kotlin.compiler.preciseCompilationResultsBackup` and `kotlin.compiler.keepIncrementalCompilationCachesInMemory` Gradle properties that opt out of this optimization.

Deprecation cycle:

- 2.1.20: report a warning
- 2.3.0: remove the properties

Deprecate `destinationDir` in `CInteropProcess`

Issue: KT-74910

Component: Gradle

Incompatible change type: source

Short summary: The `destinationDir` property in the `CInteropProcess` task is deprecated. Use the `CInteropProcess.destinationDirectory.set()` function instead.

Deprecation cycle:

- 2.1.0: report a warning when the `destinationDir` property is used
- 2.2.0: raise this warning to an error
- 2.3.0: hide the `destinationDir` property

Deprecate `konanVersion` in `CInteropProcess`

Issue: KT-74911

Component: Gradle

Incompatible change type: source

Short summary: The `konanVersion` property in the `CInteropProcess` task is deprecated. Use `CInteropProcess.kotlinNativeVersion` instead.

Deprecation cycle:

- 2.1.0: report a warning when the `konanVersion` property is used
- 2.2.0: raise this warning to an error
- 2.3.0: hide the `konanVersion` property

Remove `KotlinCompile.classpathSnapshotProperties` properties

Issue: KT-76177

Component: Gradle

Incompatible change type: source

Short summary: The `kotlin.incremental.useClasspathSnapshot` Gradle property was removed in Kotlin 2.2.0. In Kotlin 2.3.0, the following properties are also removed:

- `KotlinCompile.classpathSnapshotProperties.useClasspathSnapshot`
- `KotlinCompile.classpathSnapshotProperties.classpath`

Deprecation cycle:

- 2.0.20: deprecate the `kotlin.incremental.useClasspathSnapshot` property with a warning
- 2.2.0: remove the `kotlin.incremental.useClasspathSnapshot` property
- 2.3.0: remove the `KotlinCompile.classpathSnapshotProperties.useClasspathSnapshot` and `KotlinCompile.classpathSnapshotProperties.classpath` properties

Deprecate `getPluginArtifactForNative()` function

Issue: KT-78870

Component: Gradle

Incompatible change type: source

Short summary: In Kotlin 2.2.20, the `getPluginArtifactForNative()` function was deprecated. Use the `getPluginArtifact()` function instead.

Deprecation cycle:

- 2.2.20: report a warning
- 2.3.0: raise the warning to an error
- 2.4.0: remove the function

Change the approach to registering all generated sources

Issue: KT-45161

Component: Gradle

Incompatible change type: source

Short summary: Kotlin 2.3.0 introduces a new Experimental API in the `KotlinSourceSet` interface that lets you register generated sources in a Gradle project. Previously, you could use the `kotlin` property to access all generated sources. Starting with Kotlin 2.3.0, if your plugin or build logic needs access to all generated sources, use the `allKotlinSources` property instead.

Migration advice:

- To register generated sources, use the `generatedKotlin` property.
- To access all sources, including non-generated sources, use the `allKotlinSources` property.

Deprecate `kotlin.publishJvmEnvironmentAttribute` property

Issue: KT-83678

Component: Gradle

Incompatible change type: source

Short summary: In Kotlin 2.3.20, the `kotlin.publishJvmEnvironmentAttribute` property is deprecated. This property allowed disabling the publication of the `org.gradle.jvm.environment` attribute for multiplatform libraries. Starting with Kotlin 2.0.20, `org.gradle.jvm.environment` is published by default to ensure conventional dependency resolution.

Deprecation cycle:

- 2.3.20: report a warning
- 2.4.0: remove the property

Deprecate `CleanableStore` interface and `CleanDataTask` class

Issue: KT-78104

Component: Gradle

Incompatible change type: source

Short summary: The `CleanableStore` interface and `CleanDataTask` class are deprecated since they are no longer used.

Deprecation cycle:

- 2.3.20: report a warning
- 2.4.20: raise the warning to an error

Deprecate `kotlin.kmp.isolated-projects.support` Gradle property

Issue: KT-79257

Component: Gradle

Incompatible change type: source

Short summary: Since multiplatform projects are compatible with isolated projects by default and there are no other options, the `kotlin.kmp.isolated-projects.support` Gradle property is deprecated.

Deprecation cycle:

- 2.3.20: report a warning

Deprecate `kotlin.mpp.enableKotlinToolingMetadataArtifact` Gradle property

Issue: KT-79924

Component: Gradle

Incompatible change type: source

Short summary: Since the `kotlin-tooling-metadata.json` artifact is now always generated for multiplatform projects, the `kotlin.mpp.enableKotlinToolingMetadataArtifact` Gradle property is deprecated.

Deprecation cycle:

- 2.3.20: report a warning
- 2.4.0: remove support

Deprecate `LanguageSettings.enableLanguageFeature` DSL

Issue: KT-82323, KT-82847

Component: Gradle

Incompatible change type: source

Short summary: The `LanguageSettings.enableLanguageFeature` DSL exposed an internal compiler configuration only intended for Kotlin compiler tests. Therefore, the DSL is deprecated.

Deprecation cycle:

- 2.3.20: report a warning when using `LanguageSettings.enableLanguageFeature`
- 2.4.0: raise the warning to an error

Deprecate "out of process" compiler execution strategy

Issue: KT-83125

Component: Gradle

Incompatible change type: source

Short summary: The "out of process" compiler execution strategy is not supported by the Build tools API and is the slowest strategy available. In Kotlin 2.3.20, the strategy is deprecated in favor of "daemon" and "in process" compiler execution strategies.

Deprecation cycle:

- 2.3.20: report a warning
- 2.4.0: remove the "out of process" compiler execution strategy

Build tool removal

Remove support for Ant

Issue: KT-75875

Component: Ant

Short summary: Kotlin 2.3.0 removes support for Ant as a build tool. Use Gradle or Maven instead.

Deprecation cycle:

- 2.2.0: report a warning
- 2.3.0: remove support

What's new in Kotlin 2.2.20

Released: September 10, 2025

The Kotlin 2.2.20 release is out, delivering important changes for web development. Kotlin/Wasm is now Beta, with improvements to exception handling in JavaScript interop, npm dependency management, built-in browser debugging support, and a new shared source set for `js` and `wasmJs` targets.

Additionally, here are some main highlights:

- **Kotlin Multiplatform**: Swift export available by default, stable cross-platform compilation for Kotlin libraries, and a new approach for declaring common dependencies.
- **Language**: Improved overload resolution when passing lambdas to overloads with suspend function types.
- **Kotlin/Native**: Support for Xcode 26, stack canaries, and smaller binary size for release binaries.
- **Kotlin/JS**: `Long` values compiled into JavaScript `BigInt`.

Compose Multiplatform for web goes Beta. Learn more in our blog post.

You can also find a short overview of the updates in this video:

Watch video online.

For information about the Kotlin release cycle, see Kotlin release process.

IDE support

The Kotlin plugin that supports Kotlin 2.2.20 is bundled in the latest versions of IntelliJ IDEA and Android Studio. To update, all you need to do is change the Kotlin version to 2.2.20 in your build scripts.

See Update to a new release for details.

Language

In Kotlin 2.2.20, you can try out upcoming language features planned for Kotlin 2.3.0, including improved overload resolution when passing lambdas to overloads with `suspend` function types and support for `return` statements in expression bodies with explicit return types. This release also includes improvements to exhaustiveness checks for `when` expressions, reified `Throwable` catches, and Kotlin contracts.

Improved overload resolution for lambdas with `suspend` function types

Previously, overloading a function with both a regular function type and a `suspend` function type caused an ambiguity error when passing a lambda. You could work around this error with an explicit type cast, but the compiler incorrectly reported a `No cast needed` warning:

```
// Defines two overloads
fun transform(block: () -> Int) {}
fun transform(block: suspend () -> Int) {}

fun test() {
    // Fails with overload resolution ambiguity
    transform({ 42 })

    // Uses an explicit cast, but the compiler incorrectly reports
    // a "No cast needed" warning
    transform({ 42 } as () -> Int)
}
```

With this change, when you define both a regular and a `suspend` function type overload, a lambda without a cast resolves to the regular overload. Use the `suspend` keyword to resolve to the suspend overload explicitly:

```
// Resolves to transform(() -> Int)
transform({ 42 })

// Resolves to transform(suspend () -> Int)
transform(suspend { 42 })
```

This behavior will be enabled by default in Kotlin 2.3.0. To test it now, set your language version to `2.3` using the following compiler option:

```
-language-version 2.3
```

Or configure it in your `build.gradle(.kts)` file:

```
kotlin {
    compilerOptions {
        languageVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_3)
    }
}
```

We would appreciate your feedback in our issue tracker, YouTrack.

Support for `return` statements in expression bodies with explicit return types

Previously, using `return` in an expression body caused a compiler error because it could cause the function's return type to be inferred as `Nothing`.

```
fun example() = return 42
// Error: Returns are prohibited for functions with an expression body
```

With this change, you can now use `return` in expression bodies as long as the return type is written explicitly:

```
// Specifies the return type explicitly
fun getDisplayNameOrDefault(userId: String?): String = getDisplayName(userId ?: return "default")

// Fails because it doesn't specify the return type explicitly
fun getDisplayNameOrDefault(userId: String?) = getDisplayName(userId ?: return "default")
```

Similarly, `return` statements inside lambdas and nested expressions in functions with expression bodies used to compile unintentionally. Kotlin now supports these cases as long as the return type is specified explicitly. Cases without an explicit return type will be deprecated in Kotlin 2.3.0:

```
// Return type isn't explicitly specified, and the return statement is inside a lambda
// which will be deprecated
fun returnInsideLambda() = run { return 42 }

// Return type isn't explicitly specified, and the return statement is inside the initializer
// of a local variable, which will be deprecated
fun returnInsideIf() = when {
    else -> {
        val result = if (someCondition()) return "" else "value"
        result
    }
}
```

This behavior will be enabled by default in Kotlin 2.3.0. To test it now, set your language version to `2.3` using the following compiler option:

```
-language-version 2.3
```

Or configure it in your `build.gradle(.kts)` file:

```
kotlin {
    compilerOptions {
        languageVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_3)
    }
}
```

We would appreciate your feedback in our issue tracker, YouTrack.

Experimental

Data-flow-based exhaustiveness checks for `when` expressions

Kotlin 2.2.20 introduces **data-flow-based** exhaustiveness checks for `when` expressions. Previously, the compiler's checks were limited to the `when` expression itself, often forcing you to add a redundant `else` branch. With this update, the compiler now tracks prior condition checks and early returns, so you can remove redundant `else` branches.

For example, the compiler now recognizes that the function returns when the `if` condition is met, so the `when` expression only needs to handle the remaining cases:

```
enum class UserRole { ADMIN, MEMBER, GUEST }

fun getPermissionLevel(role: UserRole): Int {
    // Covers the Admin case outside of the when expression
    if (role == UserRole.ADMIN) return 99

    return when (role) {
        UserRole.MEMBER -> 10
        UserRole.GUEST -> 1
        // You no longer have to include this else branch
        // else -> throw IllegalStateException()
    }
}
```

This feature is Experimental. To enable it, add the following compiler option to your `build.gradle(.kts)` file:

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xdata-flow-based-exhaustiveness")
    }
}
```

Experimental

Support for reified types in `catch` clauses

In Kotlin 2.2.20, the compiler now allows using reified generic type parameters in `catch` clauses of `inline` functions.

Here's an example:

```
inline fun <reified ExceptionType : Throwable> handleException(block: () -> Unit) {
    try {
        block()
        // This is now allowed after the change
    } catch (e: ExceptionType) {
        println("Caught specific exception: ${e::class.simpleName}")
    }
}

fun main() {
    // Tries to perform an action that might throw an IOException
    handleException<java.io.IOException> {
        throw java.io.IOException("File not found")
    }
    // Caught specific exception: IOException
}
```

Previously, attempting to catch a reified `Throwable` type in an `inline` function would result in an error.

This behavior will be enabled by default in Kotlin 2.4.0. To use it now, add the following compiler option to your `build.gradle(.kts)` file:

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xallow-reified-type-in-catch")
    }
}
```

The Kotlin team is grateful to external contributor Iven Krall for their contribution.

Experimental

Improved Kotlin contracts

Kotlin 2.2.20 introduces several improvements to Kotlin contracts, including:

- Support for generics in contract type assertions.
- Support for contracts inside property accessors and specific operator functions.
- Support for the `returnsNotNull()` function in contracts as a way to ensure a non-null return value when a condition is met.
- New `holdsIn` keyword, allowing you to assume conditions are true when passed inside lambdas.

These improvements are Experimental. To opt in, you still need to use the `@OptIn(ExperimentalContracts::class)` annotation when declaring contracts. The `holdsIn` keyword and the `returnsNotNull()` function also need the `@OptIn(ExperimentalExtendedContracts::class)` annotation.

To use these improvements, you also need to add the compiler options described in each section below.

We would appreciate your feedback in our issue tracker.

Support for generics in contract type assertions

You can now write contracts that perform type assertions on generic types:

```
import kotlin.contracts.*

sealed class Failure {
    class HttpError(val code: Int) : Failure()
    // Insert other failure types here
}

sealed class Result<out T, out F : Failure> {
    class Success<T>(val data: T) : Result<T, Nothing>()
    class Failed<F : Failure>(val failure: F) : Result<Nothing, F>()
}

@OptIn(ExperimentalContracts::class)
// Uses a contract to assert a generic type
fun <T, F : Failure> Result<T, F>.isHttpError(): Boolean {
    contract {
        returns(true) implies (this@isHttpError is Result.Failed<Failure.HttpError>)
    }
    return this is Result.Failed && this.failure is Failure.HttpError
}
```

In this example, the contract performs a type assertion on the `Result` object, allowing the compiler to safely smart cast it to the asserted generic type.

This feature is Experimental. To opt in, add the following compiler option to your `build.gradle(.kts)` file:

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xallow-contracts-on-more-functions")
    }
}
```

Support for contracts inside property accessors and specific operator functions

You can now define contracts inside property accessors and specific operator functions. This lets you use contracts on more types of declarations, making them more flexible.

For example, you can use a contract inside a getter to enable smart casting for a receiver object:

```
import kotlin.contracts.*

val Any.isHelloString: Boolean
    get() {
        @OptIn(ExperimentalContracts::class)
        // Enables smart casting the receiver to String when the getter returns true
        contract { returns(true) implies (this@isHelloString is String) }
        return "hello" == this
    }

fun printIfHelloString(x: Any) {
    if (x.isHelloString) {
        // Prints the length after the smart cast of the receiver to String
        println(x.length)
        // 5
    }
}
```

Additionally, you can use contracts in the following operator functions:

- `invoke`
- `contains`
- `rangeTo`, `rangeUntil`
- `componentN`
- `iterator`
- `unaryPlus`, `unaryMinus`, `not`
- `inc`, `dec`

Here's an example of using a contract in an operator function to ensure the initialization of a variable inside a lambda:

```
import kotlin.contracts.*

class Runner {
    @OptIn(ExperimentalContracts::class)
    // Enables initialization of variables assigned inside the lambda
    operator fun invoke(block: () -> Unit) {
        contract {
            callsInPlace(block, InvocationKind.EXACTLY_ONCE)
        }
        block()
    }
}

fun testOperator(runner: Runner) {
    val number: Int
    runner {
        number = 1
    }
    // Prints the value after definite initialization guaranteed by the contract
    println(number)
    // 1
}
```

This feature is Experimental. To opt in, add the following compiler option to your `build.gradle(.kts)` file:

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xallow-contracts-on-more-functions")
    }
}
```

Support for the `returnsNotNull()` function in contracts

Kotlin 2.2.20 introduces the `returnsNotNull()` function for contracts. You can use this function to ensure that a function returns a non-null value when a specific condition is met. This simplifies your code by replacing separate nullable and non-nullable function overloads with a single, concise function:

```
import kotlin.contracts.*

@OptIn(ExperimentalContracts::class, ExperimentalExtendedContracts::class)
fun decode(encoded: String?): String? {
    contract {
        // Guarantees a non-null return value when the input is non-null
        (encoded != null) implies (returnsNotNull())
    }
    if (encoded == null) return null
    return java.net.URLDecoder.decode(encoded, "UTF-8")
}

fun useDecodedValue(s: String?) {
    // Uses a safe call since the return value may be null
    decode(s)?.length
    if (s != null) {
        // Treats the return value as non-null after the smart cast
        decode(s).length
    }
}
```

In this example, the contract in the `decode()` function allows the compiler to smart-cast its return value when the input is non-null, removing the need for extra null checks or multiple overloads.

This feature is Experimental. To opt in, add the following compiler option to your `build.gradle(.kts)` file:

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xallow-condition-implies-returns-contracts")
    }
}
```

New `holdsIn` keyword

Kotlin 2.2.20 introduces the new `holdsIn` keyword for contracts. You can use it to ensure that a boolean condition is assumed to be `true` inside a specific lambda. This lets you build DSLs with conditional smart casts using contracts.

Here's an example:

```
import kotlin.contracts.*

@OptIn(ExperimentalContracts::class, ExperimentalExtendedContracts::class)
fun <T> T.alsoIf(condition: Boolean, block: (T) -> Unit): T {
    contract {
        // Declares that the lambda runs at most once
        callsInPlace(block, InvocationKind.AT_MOST_ONCE)
        // Declares that the condition is assumed to be true inside the lambda
        condition holdsIn block
    }
    if (condition) block(this)
    return this
}

fun useApplyIf(input: Any) {
    val result = listOf(1, 2, 3)
        .first()
        .alsoIf(input is Int) {
            // The input parameter is smart cast to Int inside the lambda
            // Prints the sum of input and first list element
            println(input + it)
            // 2
        }
        .toString()
}
```

This feature is Experimental. To opt in, add the following compiler option to your `build.gradle(.kts)` file:

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xallow-holdsin-contract")
    }
}
```

Experimental

Kotlin/JVM: Support `invokedynamic` with `when` expressions

In Kotlin 2.2.20, you can now compile `when` expressions with `invokedynamic`. Previously, `when` expressions with multiple type checks compiled to a long chain of `instanceof` checks in the bytecode.

Now you can use `invokedynamic` with `when` expressions to generate smaller bytecode, similar to the bytecode produced by Java `switch` statements, when the following conditions are met:

- All conditions except for `else` are `is` or `null` checks.
- The expression doesn't contain guard conditions (`if`).
- The conditions don't include types that can't be type-checked directly, such as mutable Kotlin collections (`MutableList`) or function types (`kotlin.Function1` , `kotlin.Function2` , and so on).
- There are at least two conditions besides `else` .
- All branches check the same subject of the `when` expression.

For example:

```
open class Example

class A : Example()
class B : Example()
class C : Example()

fun test(e: Example) = when (e) {
    // Uses invokedynamic with SwitchBootstraps.typeSwitch
    is A → 1
    is B → 2
    is C → 3
    else → 0
}
```

With the new feature enabled, the `when` expression in this example compiles to a single `invokedynamic` type switch instead of multiple `instanceof` checks.

To enable this feature, compile your Kotlin code with JVM target 21 or above and add the following compiler option:

```
-Xwhen-expressions=indy
```

Or add it to the `compilerOptions {}` block of your `build.gradle(.kts)` file:

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xwhen-expressions=indy")
    }
}
```

This feature is Experimental. We would appreciate your feedback in our issue tracker, YouTrack.

Kotlin Multiplatform

Kotlin 2.2.20 introduces significant changes for Kotlin Multiplatform: Swift export is available by default, there's a new shared source set, and you can try a new approach to managing common dependencies.

Experimental

Swift export available by default

Kotlin 2.2.20 introduces experimental support for Swift export. It allows you to export Kotlin sources directly and call Kotlin code from Swift idiomatically, eliminating the need for Objective-C headers.

This should significantly improve multiplatform development for Apple targets. For example, if you have a Kotlin module with top-level functions, Swift export enables clean, module-specific imports, removing the confusing Objective-C underscores and mangled names.

The key features are:

- **Multi-module support**. Each Kotlin module is exported as a separate Swift module, simplifying function calls.
- **Package support**. Kotlin packages are explicitly preserved during export, avoiding naming conflicts in the generated Swift code.
- **Type aliases**. Kotlin type aliases are exported and preserved in Swift, improving readability.
- **Enhanced nullability for primitives**. Unlike Objective-C interop, which required boxing types like `Int?` into wrapper classes like `KotlinInt` to preserve nullability, Swift export converts nullability information directly.
- **Overloads**. You can call Kotlin's overloaded functions in Swift without ambiguity.
- **Flattened package structure**. You can translate Kotlin packages into Swift enums, removing the package prefix from generated Swift code.
- **Module name customization**. You can customize the resulting Swift module names in the Gradle configuration of your Kotlin project.

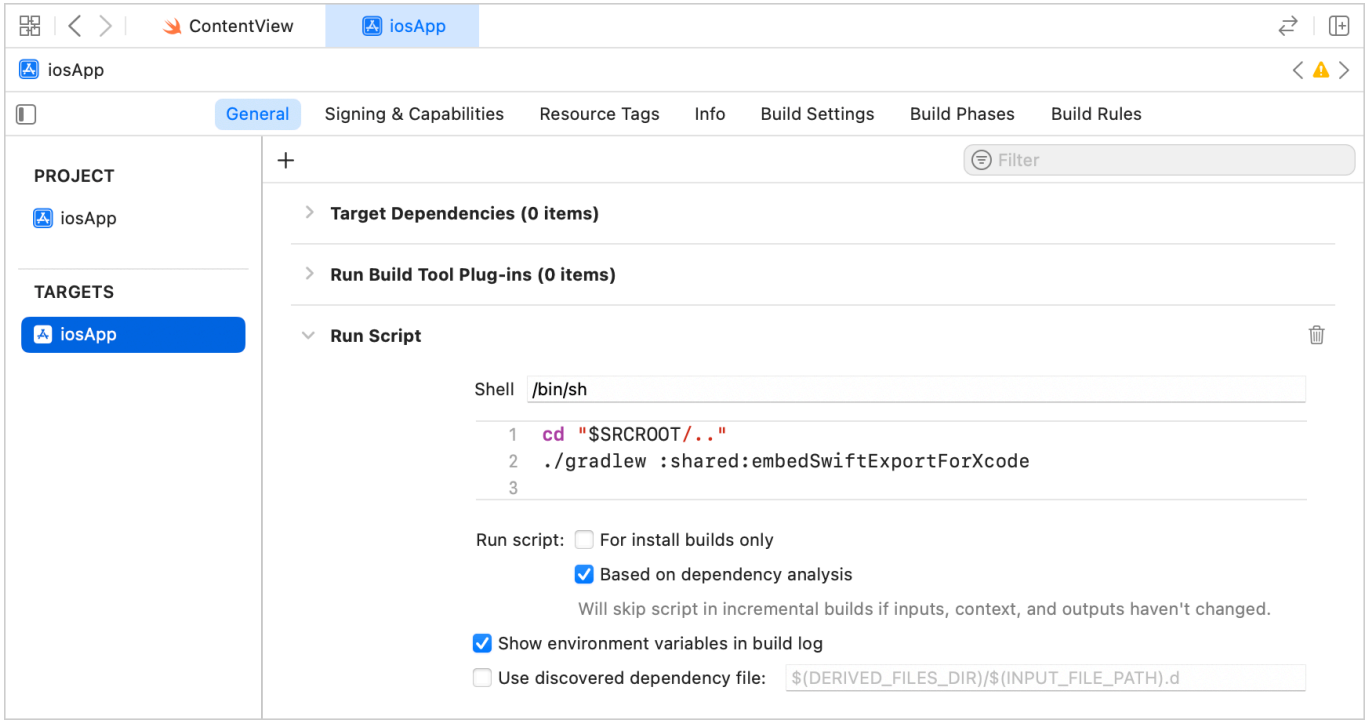
How to enable Swift export

The feature is currently Experimental and works only in projects that use direct integration to connect the iOS framework to the Xcode project. This is a standard configuration for multiplatform projects created with the Kotlin Multiplatform plugin in IntelliJ IDEA or through the web wizard.

To try out Swift export, configure your Xcode project:

1. In Xcode, open the project settings.
2. On the **Build Phases** tab, locate the **Run Script** phase with the `embedAndSignAppleFrameworkForXcode` task.
3. Adjust the script to feature the `embedSwiftExportForXcode` task instead in the run script phase:

```
./gradlew :<Shared module name>:embedSwiftExportForXcode
```



Add the Swift export script

4. Build the project. Swift modules are generated in the build output directory.

The feature is available by default. If you have already enabled it in previous releases, you can now remove `kotlin.experimental.swift-export.enabled` from your `gradle.properties` file.

> To save time, clone our public sample with Swift export already set up.

For more information about Swift export, see our documentation.

Leave feedback

We're planning to expand and gradually stabilize Swift export support in future Kotlin releases. After Kotlin 2.2.20, we'll focus on improving interoperability between Kotlin and Swift, particularly around coroutines and flows.

Support for Swift export is a significant change for Kotlin Multiplatform. We would appreciate your feedback:

- Contact the development team directly in Kotlin Slack – get an invite and join the #swift-export channel.
- Report any problems you face with Swift export in YouTrack.

Shared source set for `js` and `wasmJs` targets

Previously, Kotlin Multiplatform didn't include a shared source set for JavaScript (`js`) and WebAssembly (`wasmJs`) web targets by default. To share code between `js` and `wasmJs` , you had to manually configure a custom source set or write code in two places, one version for `js` and another for `wasmJs` . For example:

```
// commonMain
expect suspend fun readCopiedText(): String

// jsMain
external interface Navigator { val clipboard: Clipboard }
// Different interop in JS and Wasm
external interface Clipboard { fun readText(): Promise<String> }
external val navigator: Navigator

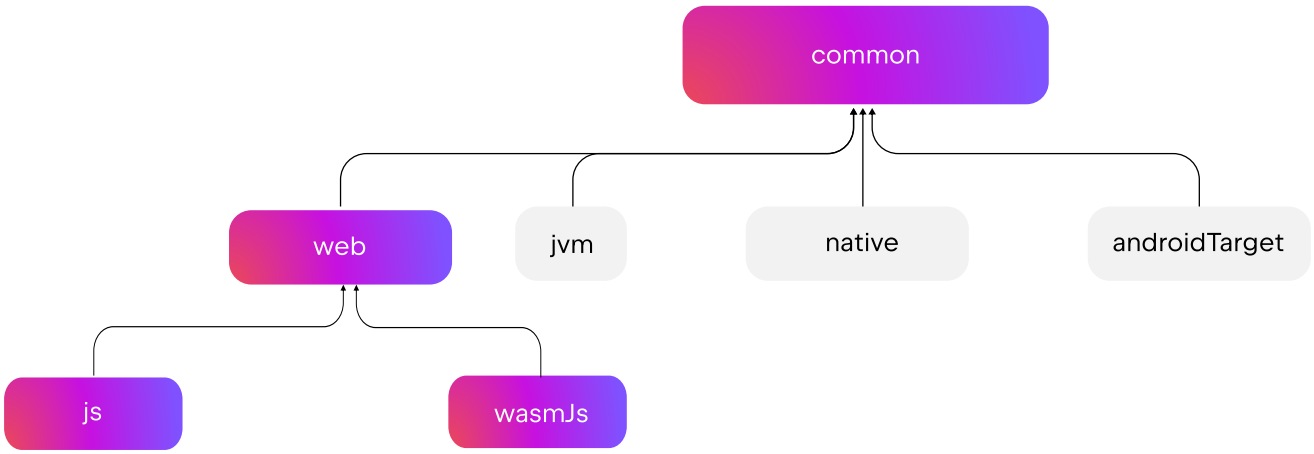
suspend fun readCopiedText(): String {
    // Different interop in JS and Wasm
    return navigator.clipboard.readText().await()
}

// wasmJsMain
external interface Navigator { val clipboard: Clipboard }
external interface Clipboard { fun readText(): Promise<JsString> }
external val navigator: Navigator

suspend fun readCopiedText(): String {
    return navigator.clipboard.readText().await().toString()
}
```

Starting with this release, the Kotlin Gradle plugin adds a new shared source set for web (comprising `webMain` and `webTest`) when you use the default hierarchy template.

With this change, the `web` source set becomes a parent of both `js` and `wasmJs` source sets. The updated source set hierarchy looks like this:



An example of using the default hierarchy template with web

The new source set allows you to write one piece of code for both the `js` and `wasmJs` targets. You can put your shared code in `webMain` and have it automatically work for both:

```
// commonMain
expect suspend fun readCopiedText(): String

// webMain
@OptIn(ExperimentalWasmJsInterop::class)
private suspend fun <R : JsAny?> Promise<R>.await(): R = suspendCancellableCoroutine { continuation ->
    this.then(
        onFulfilled = { continuation.resumeWith(Result.success(it)); null },
        onRejected = { continuation.resumeWithException(it.asJsException()); null }
    )
}

external interface Navigator { val clipboard: Clipboard }
external interface Clipboard { fun readText(): Promise<JsString> }
external val navigator: Navigator

actual suspend fun readCopiedText(): String {
    return navigator.clipboard.readText().await().toString()
}
```

This update simplifies code sharing between the `js` and `wasmJs` targets. It is particularly useful in two cases:

- If you're a library author, and you want to add support for both the `js` and `wasmJs` targets, without duplicating code.
- If you're developing Compose Multiplatform applications that target the web, enabling cross-compilation for both the `js` and `wasmJs` targets for wider browser compatibility. Given this fallback mode, when you create a website, it works on all browsers out of the box, as modern browsers use `wasmJs` and older ones use `js`.

To try this feature, use the default hierarchy template in the `kotlin {}` block of your `build.gradle(.kts)` file:

```
kotlin {
    js()
    wasmJs()

    // Enables the default source set hierarchy, including webMain and webTest
    applyDefaultHierarchyTemplate()
}
```

Before using the default hierarchy, consider carefully any potential conflicts if you have projects with a custom shared source set or if you've renamed the `js("web")` target. To resolve these conflicts, rename the conflicting source set or target, or don't use the default hierarchy.

Stable cross-platform compilation for Kotlin libraries

Kotlin 2.2.20 completes an important roadmap item, stabilizing cross-platform compilation for Kotlin libraries.

You can now use any supported host to produce `.klib` artifacts for publishing Kotlin libraries. This significantly streamlines the publishing process, particularly for Apple targets that previously required a Mac machine.

The feature is available by default. If you have already enabled cross-compilation with `kotlin.native.enableKlibsCrossCompilation=true`, you can now remove it from your `gradle.properties` file.

Unfortunately, a few limitations are still present. You still need to use a Mac machine if:

- Your library or any dependent modules have cinterop dependencies.
- You have the CocoaPods integration set up in your project.

- You need to build or test final binaries for Apple targets.

For more information about the publication of multiplatform libraries, see our documentation.

Experimental

New approach for declaring common dependencies

To simplify setting up multiplatform projects with Gradle, Kotlin 2.2.20 now lets you declare common dependencies in the `kotlin {}` block by using a top-level `dependencies {}` block when your project uses Gradle 8.8 or higher. These dependencies behave as if they were declared in the `commonMain` source set. This feature works similarly to the dependencies block that you use for Kotlin/JVM and Android-only projects, and it's now Experimental in Kotlin Multiplatform.

Declaring common dependencies at the project level reduces repetitive configuration across source sets and helps streamline your build setup. You can still add platform-specific dependencies in each source set as needed.

To try this feature, opt in by adding the `@OptIn(ExperimentalKotlinGradlePluginApi::class)` annotation before the top-level `dependencies {}` block. For example:

```
kotlin {
    @OptIn(ExperimentalKotlinGradlePluginApi::class)
    dependencies {
        implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0")
    }
}
```

We would appreciate your feedback on this feature in YouTrack.

New diagnostic for target support in dependencies

Before Kotlin 2.2.20, if a dependency in your build script didn't support all the targets required by the source set, the error messages produced by Gradle made it hard to understand the problem.

Kotlin 2.2.20 introduces a new diagnostic that clearly shows which targets each dependency supports and which it doesn't.

This diagnostic is enabled by default. If, for some reason, you need to disable it, let us know in a comment in this YouTrack issue. You can use the following Gradle properties to disable the diagnostic in your `gradle.properties` file:

Property	Description
`kotlin.kmp.eagerUnresolvedDependenciesDiagnostic=false`	Runs the diagnostic only for metadata compilations and imports
`kotlin.kmp.unresolvedDependenciesDiagnostic=false`	Disables the diagnostic completely

Kotlin/Native

This release brings support for Xcode 26, improvements to interoperability with Objective-C/Swift, debugging, and new binary options.

Support for Xcode 26

Starting with Kotlin 2.2.21, the Kotlin/Native compiler supports Xcode 26 – the latest stable version of Xcode. You can now update your Xcode and get access to the latest APIs to continue working on your Kotlin projects for Apple operating systems.

Support for stack canaries in binaries

Starting with Kotlin 2.2.20, Kotlin adds support for stack canaries in the resulting Kotlin/Native binaries. As part of stack protection, this security feature protects against stack smashing, mitigating some common application vulnerabilities. Already available in Swift and Objective-C, it's now supported in Kotlin as well.

The implementation of stack protection in Kotlin/Native follows the behavior of the stack protector in Clang.

To enable stack canaries, add the following binary option to your `gradle.properties` file:

```
kotlin.native.binary.stackProtector=yes
```

The property enables the feature for all the Kotlin functions that are vulnerable to stack smashing. Alternative modes are:

- `kotlin.native.binary.stackProtector=strong`, which uses a stronger heuristic for the functions vulnerable to stack smashing.
- `kotlin.native.binary.stackProtector=all`, which enables stack protectors for all functions.

Note that in some cases, stack protection might come with a performance cost.

Experimental

Smaller binary size for release binaries

Kotlin 2.2.20 introduces the `smallBinary` option that can help you decrease the binary size for release binaries. The new option effectively sets `-Oz` as the default optimization argument for the compiler during the LLVM compilation phase.

With the `smallBinary` option enabled, you can make release binaries smaller and improve build time. However, it might affect runtime performance in some cases.

The new feature is currently Experimental. To try it out in your project, add the following binary option to your `gradle.properties` file:

```
kotlin.native.binary.smallBinary=true
```

The Kotlin team is grateful to Troels Lund for his help in implementing this feature.

Improved debugger object summaries

Kotlin/Native now generates clearer object summaries for debugger tools like LLDB and GDB. This improves the readability of the produced debug information and streamlines your debugging experience.

Consider the following object, for example:

```
class Point(val x: Int, val y: Int)
val point = Point(1, 2)
```

Previously, the inspection would show you only limited information, including a pointer to the object's memory address:

```
(lldb) v point
(ObjHeader *) point = [x: ..., y: ...]
(lldb) v point→x
(int32_t *) x = 0x0000000100274048
```

With Kotlin 2.2.20, the debugger now shows richer details, including the actual values:

```
(lldb) v point
(ObjHeader *) point = Point(x=1, y=2)
(lldb) v point→x
(int32_t) point→x = 1
```

The Kotlin team is grateful to Nikita Nazarov for his help in implementing this feature.

For more information on debugging in Kotlin/Native, see the documentation.

Explicit names in block types for Objective-C headers

Kotlin 2.2.20 introduces an option to add explicit parameter names to Kotlin's function types for Objective-C headers exported from Kotlin/Native projects. Parameter names improve autocomplete suggestions in Xcode and help avoid Clang warnings.

Previously, parameter names in block types were omitted in the generated Objective-C headers. In such cases, Xcode's autocompletion would suggest calling such functions without parameter names in the Objective-C block. The generated block would trigger Clang warnings.

For example, for the following Kotlin code:

```
// Kotlin:
fun greetUser(block: (name: String) → Unit) = block("John")
```

The generated Objective-C header had no parameter name:

```
// Objective-C:
+ (void)greetUserBlock:(void (^)(NSString *))block __attribute__((swift_name("greetUser(block:)")));
```

So when calling the `greetUserBlock()` function from Objective-C in Xcode, the IDE suggested:

```
// Objective-C:
greetUserBlock:^(NSString *) {
    // ...
};
```

The missing parameter name `(NSString *)` in the suggestion caused Clang warnings.

With the new option, Kotlin forwards the parameter names from Kotlin function types to Objective-C block types, so Xcode uses them in suggestions:

```
// Objective-C:
greetUserBlock:^(NSString *name) {
    // ...
};
```

To enable explicit parameter names, add the following binary option to your `gradle.properties` file:

```
kotlin.native.binary.objcExportBlockExplicitParameterNames=true
```

The Kotlin team is grateful to Yijie Jiang for implementing this feature.

Reduced size of Kotlin/Native distribution

The Kotlin/Native distribution used to contain two JAR files with compiler code:

- `konan/lib/kotlin-native.jar`
- `konan/lib/kotlin-native-compiler-embeddable.jar`.

Starting with Kotlin 2.2.20, `kotlin-native.jar` is no longer published.

The removed JAR file is the legacy version of the embeddable compiler, which is no longer needed. This change significantly reduces the size of the distribution.

As a consequence, the following options are now deprecated and removed:

- The `kotlin.native.useEmbeddableCompilerJar=false` Gradle property. Instead, the embeddable compiler JAR file is always used for Kotlin/Native projects.
- The `KotlinCompilerPluginSupportPlugin.getPluginArtifactForNative()` function. Instead, the `getPluginArtifact()` function is always used.

For more information, see the YouTrack issue.

Exporting KDocs to Objective-C headers by default

KDoc comments are now exported by default when generating Objective-C headers during compilation of Kotlin/Native final binaries.

Previously, you needed to add the `-Xexport-kdoc` option manually to your build file. Now, it's automatically passed to compilation tasks.

This option embeds KDoc comments into klibs and extracts comments from klibs when producing Apple frameworks. As a result, comments on classes and methods appear during autocompletion, for example, in Xcode.

You can disable the export of KDoc comments from klibs to the produced Apple frameworks in the `binaries {}` block of your `build.gradle(.kts)` file:

```
import org.jetbrains.kotlin.gradle.ExperimentalKotlinGradlePluginApi

kotlin {
    iosArm64 {
        binaries {
            framework {
                baseName = "sdk"
                @OptIn(ExperimentalKotlinGradlePluginApi::class)
                exportKdoc.set(false)
            }
        }
    }
}
```

For more information, see our documentation.

Deprecation of `x86_64` Apple targets

Apple stopped producing devices with Intel chips a couple of years ago and recently announced that macOS Tahoe 26 will be the last OS version to support Intel-based architecture.

This makes it increasingly difficult for us to properly test these targets on our build agents, especially in future Kotlin releases where we'll update the supported Xcode version that comes with macOS 26.

Starting with Kotlin 2.2.20, the `macosX64` and `iosX64` targets are demoted to support tier 2. This means the target is regularly tested on CI to ensure it compiles, but it might not be automatically tested to ensure it runs.

We plan to gradually deprecate all `x86_64` Apple targets and eventually remove support for them during the Kotlin 2.2.20-2.4.0 release cycle. This includes the following targets:

- `macosX64`
- `iosX64`
- `tvosX64`
- `watchosX64`

For more information on support tiers, see Kotlin/Native target support.

Kotlin/Wasm

Kotlin/Wasm is now Beta, offering greater stability along with improvements such as separated npm dependencies, refined exception handling for JavaScript interop, built-in browser debugging support, and more.

Separated npm dependencies

Previously, in your Kotlin/Wasm projects, all npm dependencies were installed together in your project folder, including both Kotlin tooling dependencies and your own. They were also recorded together in your project's lock files (`package-lock.json` or `yarn.lock`).

As a result, whenever Kotlin tooling dependencies were updated, you had to update your lock files even if you didn't add or change anything.

Starting from Kotlin 2.2.20, the Kotlin tooling npm dependencies are installed outside your project. Now, the tooling and your (user) dependencies have separate directories:

- **Tooling dependencies' directory:**

  `<kotlin-user-home>/kotlin-npm-tooling/<yarn|npm>/hash/node_modules`

- **User dependencies' directory:**

  `build/wasm/node_modules`

In addition, the lock files inside the project directory contain only user-defined dependencies.

This improvement keeps your lock files focused only on your own dependencies, helps maintain a cleaner project, and reduces unnecessary changes to your files.

This change is enabled by default for the `wasm-js` target. The change is not yet implemented for the `js` target. While there are plans to implement it in future releases, the behavior of the npm dependencies remains the same as before for the `js` target in Kotlin 2.2.20.

Improved exception handling in Kotlin/Wasm and JavaScript interop

Previously, Kotlin had difficulty understanding exceptions (errors) thrown in JavaScript (JS) and crossing over to Kotlin/Wasm code.

In some cases, the issue also occurred in the reverse direction, when an exception was thrown or passed through the Wasm code to JS and wrapped into `WebAssembly.Exception` without any details. These Kotlin exception handling issues made debugging difficult.

Starting from Kotlin 2.2.20, the developer experience with exceptions improves in both directions:

- When exceptions are thrown from JS, you can see more information on Kotlin's side. When such an exception propagates through Kotlin back to JS, it's no longer wrapped into WebAssembly.
- When exceptions are thrown from Kotlin, they can now be caught on the JS side as JS errors.

The new exception handling works automatically in modern browsers that support the `WebAssembly.JSTag` feature:

- Chrome 115+
- Firefox 129+
- Safari 18.4+

In older browsers, the exception handling behavior remains unchanged.

Support for debugging in browsers without configuration

Previously, browsers couldn't automatically access the Kotlin/Wasm project sources required for debugging. To debug Kotlin/Wasm applications in the browser, you had to manually configure your build to serve these sources by adding the following snippet to your `build.gradle(.kts)` file:

```
devServer = (devServer ?: KotlinWebpackConfig.DevServer()).apply {
    static = (static ?: mutableListOf()).apply {
        add(project.rootDir.path)
    }
}
```

Starting with Kotlin 2.2.20, debugging your applications in modern browsers works out of the box. When you run Gradle development tasks (`*DevRun`), Kotlin automatically serves the source files to the browser, allowing you to set breakpoints, inspect variables, and step through Kotlin code without extra setup.

This change simplifies debugging by removing the need for manual configuration. The required configuration is now included in the Kotlin Gradle plugin. If you previously added this configuration to your `build.gradle(.kts)` file, you should remove it to avoid conflicts.

Debugging in browsers is enabled by default for all Gradle `*DevRun` tasks. These tasks serve not only the application but also its source files, so use them only for local development and avoid running them in cloud or production environments where the sources would be publicly exposed.

Handle repeated reloads during debugging

Serving sources by default may cause repeated reloads of the application in the browser before Kotlin compilation and bundling are complete. As a workaround, adjust your webpack configuration to ignore Kotlin source files and disable watching for served static files. Add a `.js` file with the following content into the `webpack.config.d` directory at the root of your project:

```
config.watchOptions = config.watchOptions || {
    ignored: ["**/*.kt", "**/node_modules"]
}

if (config.devServer) {
    config.devServer.static = config.devServer.static.map(file => {
        if (typeof file === "string") {
        return { directory: file,
                 watch: false,
        }
    } else {
        return file
    }
    })
}
```

Elimination of empty `yarn.lock` files

Previously, the Kotlin Gradle plugin (KGP) automatically generated a `yarn.lock` file that included information about npm packages required by the Kotlin toolchain, along with any existing npm dependencies from the project or used libraries.

Now, the KGP manages toolchain dependencies separately, and a project-level `yarn.lock` file is no longer generated unless the project has npm dependencies.

The KGP automatically creates a `yarn.lock` file when npm dependencies are added, and it deletes the `yarn.lock` file when npm dependencies are removed.

This change cleans up project structures and makes it easier to track when actual npm dependencies are introduced.

No additional steps are required to configure this behavior. It's applied by default in Kotlin/Wasm projects starting from Kotlin 2.2.20.

New compiler error in fully qualified class names

On Kotlin/Wasm, the compiler doesn't store fully qualified names (FQNs) of classes in the generated binary by default. This approach avoids increasing the application size.

As a result, in previous Kotlin releases, calling the `KClass::qualifiedName` property returned an empty string instead of the class's qualified name.

Starting with Kotlin 2.2.20, the compiler reports an error when you use the `KClass::qualifiedName` property in Kotlin/Wasm projects, unless you explicitly enable the qualified names feature.

This change prevents unexpected empty strings when calling the `qualifiedName` property and improves developer experience by catching issues at compile time.

The diagnostic is enabled by default, and errors are reported automatically. To disable the diagnostics and allow storing FQNs in Kotlin/Wasm, instruct the compiler to store fully qualified names for all classes by adding the following option to your `build.gradle(.kts)` file:

```
kotlin {
    wasmJs {
        ...
        compilerOptions {
            freeCompilerArgs.add("-Xwasm-kclass-fqn")
        }
    }
}
```

Keep in mind that enabling this option increases the application size.

Kotlin/JS

Kotlin 2.2.20 supports using the `BigInt` type to represent Kotlin's `Long` type, enabling `Long` in exported declarations. Additionally, this release adds a DSL function to clean up Node.js arguments.

Experimental

Usage of the `BigInt` type to represent Kotlin's `Long` type

Before the ES2020 standard, JavaScript (JS) did not support a primitive type for precise integers larger than 53 bits.

For this reason, Kotlin/JS used to represent `Long` values (which are 64-bit wide) as JavaScript objects containing two `number` properties. This custom implementation made interoperability between Kotlin and JavaScript more complex.

Starting with Kotlin 2.2.20, Kotlin/JS now uses JavaScript's built-in `BigInt` type to represent Kotlin's `Long` values when compiling to modern JavaScript (ES2020).

This change enables exporting the `Long` type to JavaScript, a feature also introduced in Kotlin 2.2.20. As a result, this change simplifies the interoperability between Kotlin and JavaScript.

To enable it, you need to add the following compiler option to your `build.gradle(.kts)` file:

```
kotlin {
    js {
        ...
        compilerOptions {
            freeCompilerArgs.add("-Xes-long-as-bigint")
        }
    }
}
```

This feature is Experimental. We would appreciate your feedback in our issue tracker, YouTrack.

Usage of `Long` in exported declarations

Because Kotlin/JS used a custom `Long` representation, it was difficult to provide a straightforward way to interact with Kotlin's `Long` from JavaScript. As a result, you couldn't export Kotlin code that used the `Long` type to JavaScript. This issue affected any code using `Long`, such as function parameters, class properties, or constructors.

Now that Kotlin's `Long` type can be compiled to JavaScript's `BigInt` type, Kotlin/JS supports exporting `Long` values to JavaScript, simplifying the interoperability between Kotlin and JavaScript code.

To enable this feature:

1. Allow exporting `Long` in Kotlin/JS by adding the following compiler option to the `freeCompilerArgs` attribute in your `build.gradle(.kts)` file:

```
kotlin {
    js {
        ...
        compilerOptions {
            freeCompilerArgs.add("-XXLanguage:+JsAllowLongInExportedDeclarations")
        }
    }
}
```

2. Enable the `BigInt` type. See how to enable it in Usage of the `BigInt` type to represent Kotlin's `Long` type.

New DSL function for cleaner arguments

When running a Kotlin/JS application with Node.js, the arguments passed to your program (`args`) used to include:

- The path to the executable `Node` .
- The path to your script.
- The actual command-line arguments you provided.

However, the expected behavior for `args` was to include only the command-line arguments. To achieve this, you had to manually skip the first two arguments using the `drop()` function inside your `build.gradle(.kts)` file or in your Kotlin code:

```
fun main(args: Array<String>) {
    println(args.drop(2).joinToString(", "))
}
```

This workaround was repetitive, error-prone, and didn't work well when sharing code between platforms.

To fix this issue, Kotlin 2.2.20 introduces a new DSL function called `passCliArgumentsToMainFunction()` .

With this function, only the command-line arguments are included, while the `Node` and script paths are excluded:

```
fun main(args: Array<String>) {
    // No need for drop() and only your custom arguments are included
    println(args.joinToString(", "))
}
```

This change reduces boilerplate code, prevents mistakes caused by manually dropping arguments, and improves cross-platform compatibility.

To enable this feature, add the following DSL function inside your `build.gradle(.kts)` file:

```
kotlin {
    js {
        nodejs {
            passCliArgumentsToMainFunction()
        }
    }
}
```

Gradle

Kotlin 2.2.20 adds new compiler performance metrics for Kotlin/Native tasks in Gradle build reports and makes quality-of-life improvements in incremental compilation.

New compiler performance metrics in build reports for Kotlin/Native tasks

In Kotlin 1.7.0, we introduced build reports to help track compiler performance. Since then, we've added more metrics to make these reports even more detailed and useful for investigating performance issues.

In Kotlin 2.2.20, build reports now include compiler performance metrics for Kotlin/Native tasks.

To learn more about build reports and how to configure them, see Enabling build reports.

Experimental

Preview improved incremental compilation for Kotlin/JVM

Kotlin 2.0.0 introduced the new K2 compiler with an optimized frontend. Kotlin 2.2.20 builds on this by using the new frontend to improve performance in certain complex incremental compilation scenarios for Kotlin/JVM.

These improvements are disabled by default while we work on stabilizing the behavior. To enable them, add the following property in your `gradle.properties` file:

```
kotlin.incremental.jvm.fir=true
```

Currently, the `kapt` compiler plugin isn't compatible with this new behavior. We're working on adding support in a future Kotlin release.

We'd appreciate your feedback on this feature in YouTrack.

Incremental compilation detects changes in lambdas of inline functions

Before Kotlin 2.2.20, if you enabled incremental compilation and changed the logic inside a lambda in an inline function, the compiler didn't recompile the call sites of that inline function in other modules. As a result, those call sites used the previous version of the lambda, which could cause unexpected behavior.

In Kotlin 2.2.20, the compiler now detects changes in lambdas of inline functions and automatically recompiles their call sites.

Improvements for library publication

Kotlin 2.2.20 adds new Gradle tasks that make library publication easier. These tasks help you generate key pairs, upload public keys, and run local checks to ensure the verification process succeeds before uploading to the Maven Central repository.

For more information about how to use these tasks as part of the publication process, see Publish your library to Maven Central.

New Gradle tasks for generating and uploading PGP keys

Before Kotlin 2.2.20, if you wanted to publish a multiplatform library to the Maven Central repository, you had to install a third-party program like `gpg` to generate a key pair for signing your publications. Now, the Kotlin Gradle plugin comes with Gradle tasks that let you generate a key pair and upload the public key so you don't have to install another program.

Generate a key pair

The `generatePgpKeys` task generates a key pair. When you run it, you must provide a password for the private keystore and your name in the following format:

```
./gradlew -Psigning.password=example-password generatePgpKeys --name "John Smith <john@example.com>"
```

The task stores the key pair in the `build/pgp` directory.

> Move your key pair to a secure location to prevent accidental deletion or unauthorized access.

Upload the public key

The `uploadPublicPgpKey` task uploads the public key to Ubuntu's key server: `keyserver.ubuntu.com`. When you run it, provide the path to the public key in `.asc` format:

```
./gradlew uploadPublicPgpKey --keyring /path_to/build/pgp/public_KEY_ID.asc
```

New Gradle tasks to test verification locally

Kotlin 2.2.20 also adds Gradle tasks for testing verification locally before uploading your library to the Maven Central repository.

If you're using the Kotlin Gradle plugin along with Gradle's Signing Plugin and Maven Publish Plugin, you can run the `checkSigningConfiguration` and `checkPomFileFor<PUBLICATION_NAME>Publication` tasks to verify that your setup meets Maven Central's requirements. Replace `<PUBLICATION_NAME>` with the name of your publication.

These tasks aren't automatically run as part of the `build` or `check` Gradle tasks, so you need to run them manually. For example, if you have a `KotlinMultiplatform` publication:

```
./gradlew checkSigningConfiguration checkPomFileForKotlinMultiplatformPublication
```

The `checkSigningConfiguration` task checks that:

- The Signing Plugin has keys configured.
- The configured public key has been uploaded to either `keyserver.ubuntu.com` or `keys.openpgp.org` key servers.
- All publications have signing enabled.

If any of these checks fail, the task returns an error with information on how to fix the issue.

The `checkPomFileFor<PUBLICATION_NAME>Publication` task checks whether the `pom.xml` file meets Maven Central's requirements. If it doesn't, the task returns an error with details about which parts of the `pom.xml` file are non-compliant.

Maven: Support for the Kotlin daemon in the `kotlin-maven-plugin`

Kotlin 2.2.20 takes the build tools API introduced in Kotlin 2.2.0 one step further by adding support for the Kotlin daemon in the `kotlin-maven-plugin`. When using the Kotlin daemon, the Kotlin compiler runs in a separate isolated process, which prevents other Maven plugins from overriding system properties. You can see an example in this YouTrack issue.

Starting with Kotlin 2.2.20, the Kotlin daemon is used by default. If you want to revert to the previous behavior, opt out by setting the following property in your `pom.xml` file to `false`:

```
<properties>
    <kotlin.compiler.daemon>false</kotlin.compiler.daemon>
</properties>
```

Kotlin 2.2.20 also introduces a new `jvmArgs` property, which you can use to customize the default JVM arguments for the Kotlin daemon. For example, to override the `-Xmx` and `-Xms` options, add the following to your `pom.xml` file:

```
<properties>
    <kotlin.compiler.daemon.jvmArgs>Xmx1500m,Xms500m</kotlin.compiler.daemon.jvmArgs>
</properties>
```

New common schema for Kotlin compiler options

Kotlin 2.2.20 introduces a common schema for all compiler options published under `org.jetbrains.kotlin:kotlin-compiler-arguments-description`. This artifact includes both a code representation and a JSON equivalent (for non-JVM consumers) of all compiler options, their descriptions, and metadata such as the version in which each option was introduced or stabilized. You can use this schema to generate a custom view of the options or analyze them as needed.

Standard library

This release introduces new experimental features in the standard library: reflection support for identifying interface types in Kotlin/JS, update functions for common atomic types, and `copyOf()` overloads for array resizing.

Experimental

Support for identifying interface types through reflection in Kotlin/JS

Kotlin 2.2.20 adds the Experimental `KClass.isInterface` property to the Kotlin/JS standard library.

With this property, you can now check whether a class reference represents a Kotlin interface. This brings Kotlin/JS closer to parity with Kotlin/JVM, where you can use `KClass.java.isInterface` to check if a class represents an interface.

To opt in, use the `@OptIn(ExperimentalStdlibApi::class)` annotation:

```
@OptIn(ExperimentalStdlibApi::class)
fun inspect(klass: KClass<*>) {
    // Prints true for interfaces
    println(klass.isInterface)
}
```

We would appreciate your feedback in our issue tracker, YouTrack.

Experimental

New update functions for common atomic types

Kotlin 2.2.20 introduces new experimental functions for updating common atomic types and elements of their array counterparts. Each function atomically computes a new value using one of these update functions and replaces the current value, with the return value depending on which function you use:

- `update()` and `updateAt()` set a new value without returning a result.
- `fetchAndUpdate()` and `fetchAndUpdateAt()` set a new value and return the previous value before the change.
- `updateAndFetch()` and `updateAndFetchAt()` set a new value and return the updated value after the change.

You can use these functions to implement atomic transformations that aren't supported out of the box, such as multiplication or bitwise operations. Before this change, incrementing a common atomic type and reading the previous value required a loop with the `compareAndSet()` function.

Like all APIs for common atomic types, these functions are Experimental. To opt in, use the `@OptIn(ExperimentalAtomicApi::class)` annotation.

Here's an example of code that performs different kinds of updates and returns either the previous or updated value:

```
import kotlin.concurrent.atomics.*
import kotlin.random.Random

@OptIn(ExperimentalAtomicApi::class)
fun main() {
    val counter = AtomicLong(Random.nextLong())
    val minSetBitsThreshold = 20

    // Sets a new value without using the result
    counter.update { if (it < 0xDECAF) 0xCACA0 else 0xC0FFEE }

    // Retrieves the current value, then updates it
    val previousValue = counter.fetchAndUpdate { 0x1CEDL.shl(Long.SIZE_BITS -
it.countLeadingZeroBits()) or it }

    // Updates the value, then retrieves the result
    val current = counter.updateAndFetch {
        if (it.countOneBits() < minSetBitsThreshold) it.shl(20) or 0x15BADL else it
    }

    val hexFormat = HexFormat {
        upperCase = true
        number {
            removeLeadingZeros = true
        }
    }
    println("Previous value: ${previousValue.toHexString(hexFormat)}")
    println("Current value: ${current.toHexString(hexFormat)}")
    println("Expected status flag set: ${current and 0xBAD == 0xBADL}")
}
```

We would appreciate your feedback in our issue tracker, YouTrack.

Experimental

Support for `copyOf()` overloads for arrays

Kotlin 2.2.20 introduces an experimental overload for the `copyOf()` function. It's available for arrays of generic type `Array<T>` and all primitive array types.

You can use this function to make an array larger and populate the new elements using values from an initializer lambda. This can help you reduce custom boilerplate code and fixes the common pain point where resizing a generic `Array<T>` produced a nullable result (`Array<T?>`).

Here's an example:

```
@OptIn(ExperimentalStdlibApi::class)
fun main() {
    val row1: Array<String> = arrayOf("one", "two")
    // Resizes the array and populates the new elements using the lambda
    val row2: Array<String> = row1.copyOf(4) { "default" }
    println(row2.contentToString())
    // [one, two, default, default]
}
```

This API is Experimental. To opt in, use the `@OptIn(ExperimentalStdlibApi::class)` annotation.

We would appreciate your feedback in our issue tracker.

Compose compiler

In this release, the Compose compiler brings quality-of-life improvements by adding new warnings and improving the output of build metrics to make them easier to read.

Language version restrictions for default parameters

With this release, the Compose compiler reports an error if the language version specified for compilation is lower than what's required to support default parameters in abstract or open composable functions.

Default parameters are supported in the Compose compiler, starting with Kotlin 2.1.0 for abstract functions and Kotlin 2.2.0 for open functions. When using a newer version of the Compose compiler while targeting older Kotlin language versions, library developers should be aware that default parameters in abstract or open functions may still appear in the public API, even if the language version doesn't support them.

Composable target warnings for the K2 compiler

This release adds warnings about `@ComposableTarget` mismatches when using the K2 compiler.

For example:

```
@Composable fun App() {
  Box { // <-- `Box` is a `@UiComposable`
    Path(...) // <-- `Path` is a `@VectorComposable`
    ^^^^^^^^^
    warning: Calling a Vector composable function where a UI composable was expected
  }
}
```

Fully qualified names in build metrics

Class and function names reported in build metrics are now fully qualified, making it easier to distinguish between declarations with the same name in different packages.

In addition, build metrics no longer include a dump of complex expressions from default parameters, making them easier to read.

Breaking changes and deprecations

This section highlights important breaking changes and deprecations worth noting:

- The kapt compiler plugin now uses the K2 compiler by default. As a result, the `kapt.use.k2` property, which controls whether the plugin uses the K2 compiler, is deprecated. If you set this property to `false` to opt out of using the K2 compiler, Gradle shows a warning.

Documentation updates

The Kotlin documentation has received some notable changes:

- Kotlin roadmap – See the updated list of Kotlin's priorities on language and ecosystem evolution.
- Properties – Learn about the many ways that you can use properties in Kotlin.
- Conditions and loops – Learn how conditions and loops work in Kotlin.

- Kotlin/JavaScript – Explore the use cases for Kotlin/JS.
- Targeting the web – Learn about the different targets that Gradle offers for web development.
- Kotlin daemon – Learn about the Kotlin daemon and how it works with build systems and the Kotlin compiler.
- Coroutines overview page – Learn about coroutine concepts and get started on your learning journey.
- Kotlin/Native binary options – Learn about the binary options for Kotlin/Native and how to configure them.
- Debugging Kotlin/Native – Explore the different ways that you can debug with Kotlin/Native.
- Tips for customizing LLVM backend – Learn how Kotlin/Native uses LLVM and adjust optimization passes.
- Get started with Exposed's DAO API – Learn how to use Exposed's Data Access Object (DAO) API to store and retrieve data in a relational database.
- New pages in Exposed documentation about R2DBC:
  - Working with databases
  - Working with ConnectionFactory
  - Custom type mapping
- HTMX integration – Learn how Ktor provides experimental, first-class support for HTMX.

How to update to Kotlin 2.2.20

The Kotlin plugin is distributed as a bundled plugin in IntelliJ IDEA and Android Studio.

To update to the new Kotlin version, change the Kotlin version to 2.2.20 in your build scripts.

What's new in Kotlin 2.2.0

Released: June 23, 2025

The Kotlin 2.2.0 release is here! Here are the main highlights:

- **Language**: new language features in preview, including context parameters. Several previously experimental features are now Stable, such as guard conditions, non-local break and continue, and multi-dollar interpolation.
- **Kotlin compiler**: Unified management of compiler warnings.
- **Kotlin/JVM**: Changes to default method generation for interface functions.
- **Kotlin/Native**: LLVM 19 and new features for tracking and adjusting memory consumption.
- **Kotlin/Wasm**: Separated Wasm target and the ability to configure Binaryen per project.
- **Kotlin/JS**: Fix for the `copy()` method generated for `@JsPlainObject` interfaces.
- **Gradle**: Binary compatibility validation in the Kotlin Gradle plugin.
- **Standard library**: Stable Base64 and HexFormat APIs.
- **Documentation**: Important improvements have been made to the Kotlin documentation.

You can also watch this video of the Kotlin Language Evolution team discussing new features and answering questions:

Watch video online.

For information about the Kotlin release cycle, see Kotlin release process.

IDE support

The Kotlin plugins that support 2.2.0 are bundled in the latest versions of IntelliJ IDEA and Android Studio. You don't need to update the Kotlin plugin in your IDE. All you need to do is to change the Kotlin version to 2.2.0 in your build scripts.

See Update to a new release for details.

Language

This release promotes guard conditions, non-local `break` and `continue`, and multi-dollar interpolation to Stable. Additionally, several features, such as context parameters and context-sensitive resolution, are introduced in preview.

Experimental

Preview of context parameters

Context parameters allow functions and properties to declare dependencies that are implicitly available in the surrounding context.

With context parameters, you don't need to manually pass around values, such as services or dependencies, that are shared and rarely change across sets of function calls.

Context parameters replace an older experimental feature called context receivers. To migrate from context receivers to context parameters, you can use assisted support in IntelliJ IDEA, as described in the blog post.

The main difference is that context parameters are not introduced as receivers in the body of a function. As a result, you need to use the name of the context parameters to access their members, unlike with context receivers, where the context is implicitly available.

Context parameters in Kotlin represent a significant improvement in managing dependencies through simplified dependency injection, improved DSL design, and scoped operations. For more information, see the feature's KEEP.

How to declare context parameters

You can declare context parameters for properties and functions using the `context` keyword followed by a list of parameters, each of the form `name: Type`. Here is an example with a dependency on the `UserService` interface:

```
// UserService defines the dependency required in the context
interface UserService {
    fun log(message: String)
    fun findUserById(id: Int): String
}

// Declares a function with a context parameter
context(users: UserService)
fun outputMessage(message: String) {
    // Uses log from the context
    users.log("Log: $message")
}

// Declares a property with a context parameter
context(users: UserService)
val firstUser: String
    // Uses findUserById from the context
    get() = users.findUserById(1)
```

You can use `_` as a context parameter name. In this case, the parameter's value is available for resolution but is not accessible by name inside the block:

```
// Uses "_" as context parameter name
context(_: UserService)
fun logWelcome() {
    // Finds the appropriate log function from UserService
    outputMessage("Welcome!")
}
```

How to enable context parameters

To enable context parameters in your project, use the following compiler option in the command line:

```
-Xcontext-parameters
```

Or add it to the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xcontext-parameters")
    }
}
```

Specifying both `-Xcontext-receivers` and `-Xcontext-parameters` compiler options simultaneously leads to an error.

Leave your feedback

This feature is planned to be stabilized and improved in future Kotlin releases. We would appreciate your feedback on our issue tracker, YouTrack.

Experimental

Preview of context-sensitive resolution

Kotlin 2.2.0 introduces an implementation of context-sensitive resolution in preview.

You can find an overview of this feature in this video:

Watch video online.

Previously, you had to write the full name of enum entries or sealed class members, even when the type could be inferred from the context. For example:

```
enum class Problem {
    CONNECTION, AUTHENTICATION, DATABASE, UNKNOWN
}

fun message(problem: Problem): String = when (problem) {
    Problem.CONNECTION -> "connection"
    Problem.AUTHENTICATION -> "authentication"
    Problem.DATABASE -> "database"
    Problem.UNKNOWN -> "unknown"
}
```

Now, with context-sensitive resolution, you can omit the type name in contexts where the expected type is known:

```
enum class Problem {
    CONNECTION, AUTHENTICATION, DATABASE, UNKNOWN
}

// Resolves enum entries based on the known type of problem
fun message(problem: Problem): String = when (problem) {
    CONNECTION -> "connection"
    AUTHENTICATION -> "authentication"
    DATABASE -> "database"
    UNKNOWN -> "unknown"
}
```

The compiler uses this contextual type information to resolve the correct member. This information includes, among other things:

- The subject of a `when` expression
- An explicit return type
- A declared variable type
- Type checks (`is`) and casts (`as`)
- The known type of a sealed class hierarchy
- The declared type of a parameter

Context-sensitive resolution doesn't apply to functions, properties with parameters, or extension properties with receivers.

To try out context-sensitive resolution in your project, use the following compiler option in the command line:

```
-Xcontext-sensitive-resolution
```

Or add it to the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xcontext-sensitive-resolution")
    }
}
```

We plan to stabilize and improve this feature in future Kotlin releases and would appreciate your feedback on our issue tracker YouTrack.

Experimental

Preview of features for annotation use-site targets

Kotlin 2.2.0 introduces a couple of features that make working with annotation use-site targets more convenient.

Experimental

`@all` meta-target for properties

Kotlin allows you to attach annotations to specific parts of a declaration, known as use-site targets. However, annotating each target individually was complex and error-prone:

```
data class User(
    val username: String,

    @param:Email      // Constructor parameter
    @field:Email      // Backing field
    @get:Email        // Getter method
    @property:Email   // Kotlin property reference
    val email: String,
) {
    @field:Email
    @get:Email
    @property:Email
    val secondaryEmail: String? = null
}
```

To simplify this, Kotlin introduces the new `@all` meta-target for properties. This feature tells the compiler to apply the annotation to all relevant parts of the property. When you use it, `@all` attempts to apply the annotation to:

- **`param`** : the constructor parameter, if declared in the primary constructor.
- **`property`** : the Kotlin property itself.
- **`field`** : the backing field, if it exists.
- **`get`** : the getter method.
- **`setparam`** : the parameter of the setter method, if the property is defined as `var` .
- **`RECORD_COMPONENT`** : if the class is a `@JvmRecord` , the annotation applies to the Java record component. This behavior mimics the way Java handles annotations on record components.

The compiler only applies the annotation to the targets for the given property.

In the example below, the `@Email` annotation is applied to all relevant targets of each property:

```
data class User(
    val username: String,

    // Applies @Email to param, property, field,
    // get, and setparam (if var)
    @all:Email val email: String,
) {
    // Applies @Email to property, field, and get
    // (no param since it's not in the constructor)
    @all:Email val secondaryEmail: String? = null
}
```

You can use the `@all` meta-target with any property, both inside and outside the primary constructor. However, you can't use the `@all` meta-target with multiple annotations.

This new feature simplifies the syntax, ensures consistency, and improves interoperability with Java records.

To enable the `@all` meta-target in your project, use the following compiler option in the command line:

```
-Xannotation-target-all
```

Or add it to the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xannotation-target-all")
    }
}
```

This feature is in preview. Please report any problems to our issue tracker, YouTrack. For more information about the `@all` meta-target, read this KEEP proposal.

Experimental

New defaulting rules for use-site annotation targets

Kotlin 2.2.0 introduces new defaulting rules for propagating annotations to parameters, fields, and properties. Where previously an annotation was applied by default only to one of `param`, `property`, or `field`, defaults are now more in line with what is expected of an annotation.

If there are multiple applicable targets, one or more is chosen as follows:

- If the constructor parameter target (`param`) is applicable, it is used.
- If the property target (`property`) is applicable, it is used.
- If the field target (`field`) is applicable while `property` isn't, `field` is used.

If there are multiple targets, and none of `param`, `property`, or `field` are applicable, the annotation results in an error.

To enable this feature, add it to the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xannotation-default-target=param-property")
    }
}
```

Or use the command-line argument for the compiler:

```
-Xannotation-default-target=param-property
```

Whenever you'd like to use the old behavior, you can:

- In a specific case, define the necessary target explicitly, for example, using `@param:Annotation` instead of `@Annotation`.
- For a whole project, use this flag in your Gradle build file:

  ```
  // build.gradle.kts
  kotlin {
      compilerOptions {
          freeCompilerArgs.add("-Xannotation-default-target=first-only")
      }
  }
  ```

This feature is in preview. Please report any problems to our issue tracker, YouTrack. For more information about the new defaulting rules for annotation use-site targets, read this KEEP proposal.

Beta

Support for nested type aliases

Kotlin 2.2.0 adds support for defining type aliases inside other declarations.

You can find an overview of this feature in this video:

Watch video online.

Previously, you could only declare type aliases at the top level of a Kotlin file. This meant that even internal or domain-specific type aliases had to live outside the class where they were used.

Starting from 2.2.0, you can define type aliases inside other declarations, as long as they don't capture type parameters from their outer class:

```
class Dijkstra {
    typealias VisitedNodes = Set<Node>

    private fun step(visited: VisitedNodes, ...) = ...
}
```

Nested type aliases have a few additional constraints, like not being able to mention type parameters. Check the documentation for the entire set of rules.

Nested type aliases allow for cleaner, more maintainable code by improving encapsulation, reducing package-level clutter, and simplifying internal implementations.

How to enable nested type aliases

To enable nested type aliases in your project, use the following compiler option in the command line:

```
-Xnested-type-aliases
```

Or add it to the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xnested-type-aliases")
    }
}
```

Share your feedback

Nested type aliases are currently in Beta. Please report any problems to our issue tracker, YouTrack. For more information about this feature, read this KEEP proposal.

Stable features: Guard conditions, non-local `break` and `continue`, and multi-dollar interpolation

In Kotlin 2.1.0, several new language features were introduced in preview. We're happy to announce that the following language features are now Stable in this release:

- Guard conditions in `when` with a subject
- Non-local `break` and `continue`
- Multi-dollar interpolation: improved handling of `$` in string literals

See the full list of Kotlin language design features and proposals.

Experimental

Kotlin compiler: Unified management of compiler warnings

Kotlin 2.2.0 introduces a new compiler option, `-Xwarning-level`. It's designed to provide a unified way of managing compiler warnings in Kotlin projects.

Previously, you could only apply general module-wide rules, like disabling all warnings with `-nowarn`, turning all warnings to compilation errors with `-Werror`, or enabling additional compiler checks with `-Wextra`. The only option to adjust them for specific warnings was the `-Xsuppress-warning` option.

With the new solution, you can override general rules and exclude specific diagnostics in a consistent way.

How to apply

The new compiler option has the following syntax:

```
-Xwarning-level=DIAGNOSTIC_NAME:(error|warning|disabled)
```

- `error`: raises the specified warning to an error.
- `warning`: emits a warning and is enabled by default.

- `disabled` : completely suppresses the specified warning module-wide.

Keep in mind that you can only configure the severity level of *warnings* with the new compiler option.

Use cases

With the new solution, you can better fine-tune warning reporting in your project by combining general rules with specific ones. Choose your use case:

Suppress warnings

Command	Description
`-nowarn`	Suppresses all warnings during compilation.
`-Xwarning-level=DIAGNOSTIC_NAME:disabled`	Suppresses only specified warnings.
`-nowarn -Xwarning-level=DIAGNOSTIC_NAME:warning`	Suppresses all warnings except for the specified ones.

Raise warnings to errors

Command	Description
`-Werror`	Raises all warnings to compilation errors.
`-Xwarning-level=DIAGNOSTIC_NAME:error`	Raises only specified warnings to errors.
`-Werror -Xwarning-level=DIAGNOSTIC_NAME:warning`	Raises all warnings to errors except for the specified ones.

Enable additional compiler warnings

Command	Description
`-Wextra`	Enables all additional declaration, expression, and type compiler checks that emit warnings if true.
`-Xwarning-level=DIAGNOSTIC_NAME:warning`	Enables only specified additional compiler checks.
`-Wextra -Xwarning-level=DIAGNOSTIC_NAME:disabled`	Enables all additional checks except for the specified ones.

Warning lists

In case you have many warnings you want to exclude from general rules, you can list them in a separate file through `@argfile`.

Leave feedback

The new compiler option is still Experimental. Please report any problems to our issue tracker, YouTrack.

Kotlin/JVM

Kotlin 2.2.0 brings many updates to the JVM. The compiler now supports Java 24 bytecode and introduces changes to default method generation for interface functions. The release also simplifies working with annotations in Kotlin metadata, improves Java interop with inline value classes, and includes better support for annotating JVM records.

Changes to default method generation for interface functions

Starting from Kotlin 2.2.0, functions declared in interfaces are compiled to JVM default methods unless configured otherwise. This change affects how Kotlin's interface functions with implementations are compiled to bytecode.

This behavior is controlled by the new stable compiler option `-jvm-default`, replacing the deprecated `-Xjvm-default` option.

You can control the behavior of the `-jvm-default` option using the following values:

- `enable` (default): generates default implementations in interfaces and includes bridge functions in subclasses and `DefaultImpls` classes. Use this mode to maintain binary compatibility with older Kotlin versions.
- `no-compatibility`: generates only default implementations in interfaces. This mode skips compatibility bridges and `DefaultImpls` classes, making it suitable for new code.
- `disable`: disables default implementations in interfaces. Only bridge functions and `DefaultImpls` classes are generated, matching the behavior before Kotlin 2.2.0.

To configure the `-jvm-default` compiler option, set the `jvmDefault` property in your Gradle Kotlin DSL:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        jvmDefault = JvmDefaultMode.NO_COMPATIBILITY
    }
}
```

Experimental

Support for reading and writing annotations in Kotlin metadata

Previously, you had to read annotations from compiled JVM class files using reflection or bytecode analysis and manually match them to metadata entries based on signatures. This process was error-prone, especially for overloaded functions.

Now, in Kotlin 2.2.0, the Kotlin Metadata JVM library introduces support for reading annotations stored in Kotlin metadata.

To make annotations available in the metadata for your compiled files, add the following compiler option:

```
-Xannotations-in-metadata
```

Alternatively, add it to the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xannotations-in-metadata")
    }
}
```

With this option enabled, the Kotlin compiler writes annotations into metadata alongside the JVM bytecode, making them accessible to the `kotlin-metadata-jvm` library.

The library provides the following APIs for accessing annotations:

- `KmClass.annotations`
- `KmFunction.annotations`
- `KmProperty.annotations`
- `KmConstructor.annotations`
- `KmPropertyAccessorAttributes.annotations`
- `KmValueParameter.annotations`
- `KmFunction.extensionReceiverAnnotations`
- `KmProperty.extensionReceiverAnnotations`
- `KmProperty.backingFieldAnnotations`
- `KmProperty.delegateFieldAnnotations`
- `KmEnumEntry.annotations`

These APIs are Experimental. To opt in, use the `@OptIn(ExperimentalAnnotationsInMetadata::class)` annotation.

Here's an example of reading annotations from Kotlin metadata:

```
@file:OptIn(ExperimentalAnnotationsInMetadata::class)

import kotlin.metadata.ExperimentalAnnotationsInMetadata
import kotlin.metadata.jvm.KotlinClassMetadata

annotation class Label(val value: String)

@Label("Message class")
class Message

fun main() {
    val metadata = Message::class.java.getAnnotation(Metadata::class.java)
    val kmClass = (KotlinClassMetadata.readStrict(metadata) as KotlinClassMetadata.Class).kmClass
    println(kmClass.annotations)
    // [@Label(value = StringValue("Message class"))]
}
```

If you use the `kotlin-metadata-jvm` library in your projects, we recommend testing and updating your code to support annotations. Otherwise, when annotations in metadata become enabled by default in a future Kotlin version, your projects may produce invalid or incomplete metadata.

If you experience any problems, please report them in our issue tracker.

Experimental

Improved Java interop with inline value classes

Kotlin 2.2.0 introduces a new experimental annotation: `@JvmExposeBoxed`. This annotation makes it easier to consume inline value classes from Java.

You can find an overview of this feature in this video:

Watch video online.

By default, Kotlin compiles inline value classes to use **unboxed representations**, which are more performant but often hard or even impossible to use from Java. For example:

```
@JvmInline value class PositiveInt(val number: Int) {
    init { require(number >= 0) }
}
```

In this case, because the class is unboxed, there is no constructor available for Java to call. There's also no way for Java to trigger the `init` block to ensure that `number` is positive.

When you annotate the class with `@JvmExposeBoxed`, Kotlin generates a public constructor that Java can call directly, ensuring that the `init` block also runs.

You can apply the `@JvmExposeBoxed` annotation at the class, constructor, or function level to gain fine-grained control over what's exposed to Java.

For example, in the following code, the extension function `.timesTwoBoxed()` is **not** accessible from Java:

```
@JvmInline
value class MyInt(val value: Int)

fun MyInt.timesTwoBoxed(): MyInt = MyInt(this.value * 2)
```

To make it possible to create an instance of the `MyInt` class and call the `.timesTwoBoxed()` function from Java code, add the `@JvmExposeBoxed` annotation to both the class and the function:

```
@JvmExposeBoxed
@JvmInline
value class MyInt(val value: Int)

@JvmExposeBoxed
fun MyInt.timesTwoBoxed(): MyInt = MyInt(this.value * 2)
```

With these annotations, the Kotlin compiler generates a Java-accessible constructor for the `MyInt` class. It also generates an overload for the extension function that uses the boxed form of the value class. As a result, the following Java code runs successfully:

```
MyInt input = new MyInt(5);
MyInt output = ExampleKt.timesTwoBoxed(input);
```

If you don't want to annotate every part of the inline value classes that you want to expose, you can effectively apply the annotation to a whole module. To apply this behavior to a module, compile it with the `-Xjvm-expose-boxed` option. Compiling with this option has the same effect as if every declaration in the module had the `@JvmExposeBoxed` annotation.

This new annotation does not change how Kotlin compiles or uses value classes internally, and all existing compiled code remains valid. It simply adds new capabilities to improve Java interoperability. The performance of Kotlin code using value classes is not impacted.

The `@JvmExposeBoxed` annotation is useful for library authors who want to expose boxed variants of member functions and receive boxed return types. It eliminates the need to choose between an inline value class (efficient but Kotlin-only) and a data class (Java-compatible but always boxed).

For a more detailed explanation of how the `@JvmExposedBoxed` annotation works and the problems it solves, see this KEEP proposal.

Improved support for annotating JVM records

Kotlin has supported JVM records since Kotlin 1.5.0. Now, Kotlin 2.2.0 improves how Kotlin handles annotations on record components, particularly in relation to Java's `RECORD_COMPONENT` target.

Firstly, if you want to use a `RECORD_COMPONENT` as an annotation target, you need to manually add annotations for Kotlin (`@Target`) and Java. This is because Kotlin's `@Target` annotation doesn't support `RECORD_COMPONENT`. For example:

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.PROPERTY)
@java.lang.annotation.Target(ElementType.CLASS, ElementType.RECORD_COMPONENT)
annotation class exampleClass
```

Maintaining both lists manually can be error-prone, so Kotlin 2.2.0 introduces a compiler warning if the Kotlin and Java targets don't match. For instance, if you omit `ElementType.CLASS` in the Java target list, the compiler reports:

```
Incompatible annotation targets: Java target 'CLASS' missing, corresponding to Kotlin targets 'CLASS'.
```

Secondly, Kotlin's behavior differs from Java when it comes to propagating annotations in records. In Java, annotations on a record component automatically apply to the backing field, getter, and constructor parameter. Kotlin doesn't do this by default, but you can now replicate the behavior using the `@all:` use-site target.

For example:

```
@JvmRecord
data class Person(val name: String, @all:Positive val age: Int)
```

When you use `@JvmRecord` with `@all:`, Kotlin now:

- Propagates the annotation to the property, backing field, constructor parameter, and getter.
- Applies the annotation to the record component, as well, if the annotation supports Java's `RECORD_COMPONENT`.

Kotlin/Native

Starting with 2.2.0, Kotlin/Native uses LLVM 19. This release also brings several experimental features designed to track and adjust memory consumption.

Experimental

Per-object memory allocation

Kotlin/Native's memory allocator can now reserve memory on a per-object basis. In some cases, it may help you satisfy strict memory limitations or reduce memory consumption on the application's startup.

The new feature is designed to replace the `-Xallocator=std` compiler option, which enabled the system memory allocator instead of the default one. Now, you can disable buffering (paging of allocations) without switching memory allocators.

The feature is currently Experimental. To enable it, set the following option in your `gradle.properties` file:

```
kotlin.native.binary.pagedAllocator=false
```

Please report any problems to our issue tracker, YouTrack.

Experimental

Support for Latin-1 encoded strings at runtime

Kotlin now supports Latin-1-encoded strings, similarly to the JVM. This should help reduce the application's binary size and adjust memory consumption.

By default, strings in Kotlin are stored using UTF-16 encoding, where each character is represented by two bytes. In some cases, this leads to strings taking up twice as much space in the binary compared to the source code, and reading data from a simple ASCII file can take up twice as much memory as storing the file on disk.

In turn, Latin-1 (ISO 8859-1) encoding represents each of the first 256 Unicode characters with just one byte. With Latin-1 support enabled, strings are stored in Latin-1 encoding as long as all the characters fall within its range. Otherwise, the default UTF-16 encoding is used.

How to enable Latin-1 support

The feature is currently Experimental. To enable it, set the following option in your `gradle.properties` file:

```
kotlin.native.binary.latin1Strings=true
```

Known issues

As long as the feature is Experimental, the cinterop extension functions `String.pin` , `String.usePinned` , and `String.refTo` become less efficient. Each call to them may trigger automatic string conversion to UTF-16.

The Kotlin team is very grateful to our colleagues at Google and Sonya Valchuk in particular for implementing this feature.

For more information on memory consumption in Kotlin, see the documentation.

Improved tracking of memory consumption on Apple platforms

Starting with Kotlin 2.2.0, memory allocated by Kotlin code is now tagged. This can help you debug memory issues on Apple platforms.

When inspecting your application's high memory usage, you can now identify how much memory is reserved by Kotlin code. Kotlin's share is tagged with an identifier and can be tracked through tools like VM Tracker in Xcode Instruments.

This feature is enabled by default but is available only in the Kotlin/Native default memory allocator when *all* the following conditions are met:

- **Tagging enabled**. The memory should be tagged with a valid identifier. Apple recommends numbers between 240 and 255; the default value is 246.

  If you set up the `kotlin.native.binary.mmapTag=0` Gradle property, tagging is disabled.

- **Allocation with mmap**. The allocator should use the `mmap` system call to map files into memory.

  If you set up the `kotlin.native.binary.disableMmap=true` Gradle property, the default allocator uses `malloc` instead of `mmap`.

- **Paging enabled**. Paging of allocations (buffering) should be enabled.

  If you set up the `kotlin.native.binary.pagedAllocator=false` Gradle property, the memory is reserved on a per-object basis instead.

For more information on memory consumption in Kotlin, see the documentation.

LLVM update from 16 to 19

In Kotlin 2.2.0, we updated LLVM from version 16 to 19. The new version includes performance improvements, bug fixes, and security updates.

This update shouldn't affect your code, but if you encounter any issues, please report them to our issue tracker.

Windows 7 target deprecated

Starting with Kotlin 2.2.0, the minimal supported Windows version has been raised from Windows 7 to Windows 10. Since Microsoft ended support for Windows 7 in January 2025, we also decided to deprecate this legacy target.

For more information, see Kotlin/Native supported targets and hosts.

Kotlin/Wasm

In this release, the build infrastructure for the Wasm target is separated from the JavaScript target. Additionally, now you can configure the Binaryen tool per project or module.

Build infrastructure for Wasm target separated from JavaScript target

Before, the `wasmJs` target shared the same infrastructure as the `js` target. As a result, both targets were hosted in the same directory (`build/js`) and used the same NPM tasks and configurations.

Now, the `wasmJs` target has its own infrastructure separate from the `js` target. This allows Wasm tasks and types to be distinct from JavaScript ones, enabling independent configuration.

Additionally, Wasm-related project files and NPM dependencies are now stored in a separate `build/wasm` directory.

New NPM-related tasks have been introduced for Wasm, while existing JavaScript tasks are now dedicated only to JavaScript:

Wasm tasks	JavaScript tasks
`kotlinWasmNpmInstall`	`kotlinNpmInstall`
`wasmRootPackageJson`	`rootPackageJson`

Similarly, new Wasm-specific declarations have been added:

Wasm declarations	JavaScript declarations
`WasmNodeJsRootPlugin`	`NodeJsRootPlugin`
`WasmNodeJsPlugin`	`NodeJsPlugin`
`WasmYarnPlugin`	`YarnPlugin`
`WasmNodeJsRootExtension`	`NodeJsRootExtension`
`WasmNodeJsEnvSpec`	`NodeJsEnvSpec`
`WasmYarnRootEnvSpec`	`YarnRootEnvSpec`

You can now work with the Wasm target independently of the JavaScript target, which simplifies the configuration process.

This change is enabled by default and requires no additional setup.

Per-project Binaryen configuration

The Binaryen tool, used in Kotlin/Wasm to optimize production builds, was previously configured once in the root project.

Now, you can configure the Binaryen tool per project or module. This change aligns with Gradle's best practices and ensures better support for features like project isolation, improving build performance and reliability in complex builds.

Additionally, you can now configure different versions of Binaryen for different modules, if needed.

This feature is enabled by default. However, if you have a custom configuration of Binaryen, you now need to apply it per project, rather than only in the root project.

Kotlin/JS

This release improves the `copy()` function in `@JsPlainObject` interfaces, type aliases in files with the `@JsModule` annotation, and other Kotlin/JS features.

Fix for `copy()` in `@JsPlainObject` interfaces

Kotlin/JS has an experimental plugin called `js-plain-objects`, which introduced a `copy()` function for interfaces annotated with `@JsPlainObject`. You can use the `copy()` function to manipulate objects.

However, the initial implementation of `copy()` was not compatible with inheritance, and this caused issues when a `@JsPlainObject` interface extended other interfaces.

To avoid limitations on plain objects, the `copy()` function has been moved from the object itself to its companion object:

```
@JsPlainObject
external interface User {
    val name: String
    val age: Int
}

fun main() {
    val user = User(name = "SomeUser", age = 21)
    // This syntax is not valid anymore
    val copy = user.copy(age = 35)
    // This is the correct syntax
    val copy = User.copy(user, age = 35)
}
```

This change resolves conflicts in the inheritance hierarchy and eliminates ambiguity. It is enabled by default starting from Kotlin 2.2.0.

Support for type aliases in files with `@JsModule` annotation

Previously, files annotated with `@JsModule` to import declarations from JavaScript modules were restricted to external declarations only. This meant that you couldn't declare a `typealias` in such files.

Starting with Kotlin 2.2.0, you can declare type aliases inside files marked with `@JsModule`:

```
@file:JsModule("somepackage")
package somepackage
typealias SomeClass = Any
```

This change reduces an aspect of Kotlin/JS interoperability limitations, and more improvements are planned for future releases.

Support for type aliases in files with `@JsModule` is enabled by default.

Support for `@JsExport` in multiplatform `expect` declarations

When working with the `expect/actual` mechanism in Kotlin Multiplatform projects, it was not possible to use the `@JsExport` annotation for `expect` declarations in common code.

Starting with this release, you can apply `@JsExport` directly to `expect` declarations:

```
// commonMain

// Produced error, but now works correctly
@JsExport
expect class WindowManager {
    fun close()
}

@JsExport
fun acceptWindowManager(manager: WindowManager) {
    ...
}

// jsMain

@JsExport
actual class WindowManager {
    fun close() {
        window.close()
    }
}
```

You must also annotate with `@JsExport` the corresponding `actual` implementation in the JavaScript source set, and it has to use only exportable types.

This fix allows shared code defined in `commonMain` to be correctly exported to JavaScript. You can now expose your multiplatform code to JavaScript consumers without having to use manual workarounds.

This change is enabled by default.

Ability to use `@JsExport` with the `Promise<Unit>` type

Previously, when you tried to export a function returning the `Promise<Unit>` type with the `@JsExport` annotation, the Kotlin compiler produced an error.

While return types like `Promise<Int>` worked correctly, using `Promise<Unit>` triggered a "non-exportable type" warning, even though it correctly mapped to `Promise<void>` in TypeScript.

This restriction has been removed. Now, the following code compiles without error:

```
// Worked correctly before
@JsExport
fun fooInt(): Promise<Int> = GlobalScope.promise {
    delay(100)
    return@promise 42
}

// Produced error, but now works correctly
@JsExport
fun fooUnit(): Promise<Unit> = GlobalScope.promise {
    delay(100)
}
```

This change removes an unnecessary restriction in the Kotlin/JS interop model. This fix is enabled by default.

Gradle

Kotlin 2.2.0 is fully compatible with Gradle 7.6.3 through 8.14. You can also use Gradle versions up to the latest Gradle release. However, be aware that doing so may result in deprecation warnings, and some new Gradle features might not work.

In this release, the Kotlin Gradle plugin comes with several improvements to its diagnostics. It also introduces an experimental integration of binary compatibility validation, making it easier to work on libraries.

Experimental

Binary compatibility validation included in Kotlin Gradle plugin

To make it easier to check for binary compatibility between library versions, we're experimenting with moving the functionality of the binary compatibility validator into the Kotlin Gradle plugin (KGP). You can try it out in toy projects, but we don't recommend using it in production yet.

The original binary compatibility validator continues to be maintained during this experimental phase.

Kotlin libraries can use one of two binary formats: JVM class files or `klib`. Since these formats aren't compatible, the KGP works with each of them separately.

To enable the binary compatibility validation feature set, add the following to the `kotlin{}` block in your `build.gradle.kts` file:

```
// build.gradle.kts
kotlin {
    @OptIn(org.jetbrains.kotlin.gradle.dsl.abi.ExperimentalAbiValidation::class)
    abiValidation {
        // Use the set() function to ensure compatibility with older Gradle versions
        enabled.set(true)
    }
}
```

If your project has multiple modules where you want to check for binary compatibility, configure the feature in each module separately. Each module can have its own custom configuration.

Once enabled, run the `checkLegacyAbi` Gradle task to check for binary compatibility issues. You can run the task in IntelliJ IDEA or from the command line in your project directory:

```
./gradlew checkLegacyAbi
```

This task generates an application binary interface (ABI) dump from the current code as a UTF-8 text file. The task then compares the new dump with the one from the previous release. If the task finds any differences, it reports them as errors. After reviewing the errors, if you decide the changes are acceptable, you can update the reference ABI dump by running the `updateLegacyAbi` Gradle task.

Filter classes

The feature lets you filter classes in the ABI dump. You can include or exclude classes explicitly by name or partial name, or by the annotations (or parts of annotation names) that mark them.

For example, this sample excludes all classes in the `com.company` package:

```
// build.gradle.kts
kotlin {
    @OptIn(org.jetbrains.kotlin.gradle.dsl.abi.ExperimentalAbiValidation::class)
    abiValidation {
        filters.excluded.byNames.add("com.company.**")
    }
}
```

Explore the KGP API reference to learn more about configuring the binary compatibility validator.

Multiplatform limitations

In multiplatform projects, if your host doesn't support cross-compilation for all targets, the KGP tries to infer the ABI changes for unsupported targets by checking the ABI dumps from other ones. This approach helps avoid false validation failures if you later switch to a host that **can** compile all targets.

You can change this default behavior so that the KGP doesn't infer ABI changes for unsupported targets by adding the following to your `build.gradle.kts` file:

```
// build.gradle.kts
kotlin {
    @OptIn(org.jetbrains.kotlin.gradle.dsl.abi.ExperimentalAbiValidation::class)
    abiValidation {
        klib {
            keepUnsupportedTargets = false
        }
    }
}
```

However, if you have an unsupported target in your project, running the `checkLegacyAbi` task fails because the task can't create an ABI dump. This behavior may be desirable if it's more important for the check to fail than to miss an incompatible change due to inferred ABI changes from other targets.

Support for rich output in console for Kotlin Gradle plugin

In Kotlin 2.2.0, we support color and other rich output in the console during the Gradle build process, making it easier to read and understand the reported diagnostics.

Rich output is available in supported terminal emulators for Linux and macOS, and we're working on adding support for Windows.

```
Solutions:
 • It is unsupported, please stop using it.
 • History based incremental compilation approach for JVM platform is removed. Kotlin Gradle plugin is now usi

w: ⚠Usage of Internal Kotlin Gradle Plugin Properties Detected
ATTENTION! This build uses the following Kotlin Gradle Plugin properties:

kotlin.internal.suppress.buildToolsApiVersionConsistencyChecks

Internal properties are not recommended for production use.
Stability and future compatibility of the build is not guaranteed.
Solution:
Please consider using the public API instead of internal properties.


> Configure project :kotlin-dom-api-compat
w: 'kotlin-js' Gradle plugin is deprecated and will be removed in the future.
Please use 'kotlin("multiplatform")' plugin with a 'js()' target instead. See the migration guide: https://kot
<=------------> 13% CONFIGURING [50s]
```

Gradle console

This feature is enabled by default, but if you want to override it, add the following Gradle property to your `gradle.properties` file:

```
org.gradle.console=plain
```

For more information about this property and its options, see Gradle's documentation on Customizing log format.

Integration of Problems API within KGP diagnostics

Previously, the Kotlin Gradle Plugin (KGP) was only able to report diagnostics such as warnings and errors as plain text output to the console or logs.

Starting with 2.2.0, the KGP introduces an additional reporting mechanism: it now uses Gradle's Problems API, a standardized way to report rich, structured problem information during the build process.

The KGP diagnostics are now easier to read and more consistently displayed across different interfaces, such as the Gradle CLI and IntelliJ IDEA.

This integration is enabled by default, starting with Gradle 8.6 or later. As the API is still evolving, use the most recent Gradle version to benefit from the latest improvements.

KGP compatibility with `--warning-mode`

The Kotlin Gradle Plugin (KGP) diagnostics reported issues using fixed severity levels, meaning Gradle's `--warning-mode` command-line option had no effect on how the KGP displayed errors.

Now, the KGP diagnostics are compatible with the `--warning-mode` option, providing more flexibility. For example, you can convert all warnings into errors or disable warnings entirely.

With this change, the KGP diagnostics adjust the output based on the selected warning mode:

- When you set `--warning-mode=fail`, diagnostics with `Severity.Warning` are now elevated to `Severity.Error`.
- When you set `--warning-mode=none`, diagnostics with `Severity.Warning` are not logged.

This behavior is enabled by default starting with 2.2.0.

To ignore the `--warning-mode` option, set the following Gradle property to your `gradle.properties` file:

```
kotlin.internal.diagnostics.ignoreWarningMode=true
```

Experimental

New experimental build tools API

You can use Kotlin with various build systems, such as Gradle, Maven, Amper, and others. However, integrating Kotlin into each system to support the full feature set, such as incremental compilation and compatibility with Kotlin compiler plugins, daemons, and Kotlin Multiplatform, requires significant effort.

To simplify this process, Kotlin 2.2.0 introduces a new experimental build tools API (BTA). The BTA is a universal API that acts as an abstraction layer between build systems and the Kotlin compiler ecosystem. With this approach, each build system only needs to support a single BTA entry point.

Currently, the BTA supports Kotlin/JVM only. The Kotlin team at JetBrains already uses it in the Kotlin Gradle plugin (KGP) and the `kotlin-maven-plugin`. You can try the BTA through these plugins, but the API itself isn't ready yet for general use in your own build tool integrations. If you're curious about the BTA proposal or want to share your feedback, see this KEEP proposal.

To try the BTA in:

- The KGP, add the following property to your `gradle.properties` file:

```
kotlin.compiler.runViaBuildToolsApi=true
```

- Maven, you don't need to do anything. It's enabled by default.

The BTA currently has no direct benefits for the Maven plugin, but it lays a solid foundation for the faster delivery of new features, such as support for the Kotlin daemon and the stabilization of incremental compilation.

For the KGP, using the BTA already has the following benefits:

- Improved "in process" compiler execution strategy
- More flexibility to configure different compiler versions from Kotlin

Improved "in process" compiler execution strategy

The KGP supports three Kotlin compiler execution strategies. The "in process" strategy, which runs the compiler inside the Gradle daemon process, previously didn't support incremental compilation.

Now, using the BTA, the "in-process" strategy **does** support incremental compilation. To use it, add the following property to your `gradle.properties` file:

```
kotlin.compiler.execution.strategy=in-process
```

Flexibility to configure different compiler versions from Kotlin

Sometimes you might want to use a newer Kotlin compiler version in your code while keeping the KGP on an older one – for example, to try new language features while still working through build script deprecations. Or you might want to update the version of the KGP but keep an older Kotlin compiler version.

The BTA makes this possible. Here's how you can configure it in your `build.gradle.kts` file:

```kotlin
// build.gradle.kts
import org.jetbrains.kotlin.buildtools.api.ExperimentalBuildToolsApi
import org.jetbrains.kotlin.gradle.ExperimentalKotlinGradlePluginApi

plugins {
    kotlin("jvm") version "2.2.0"
}

group = "org.jetbrains.example"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

kotlin {
    jvmToolchain(8)
    @OptIn(ExperimentalBuildToolsApi::class, ExperimentalKotlinGradlePluginApi::class)
    compilerVersion.set("2.1.21") // Different version than 2.2.0
}
```

The BTA supports configuring the KGP and Kotlin compiler versions with the three previous major versions and one subsequent major version. So in KGP 2.2.0, Kotlin compiler versions 2.1.x, 2.0.x, and 1.9.25 are supported. KGP 2.2.0 is also compatible with future Kotlin compiler versions 2.2.x and 2.3.x.

However, keep in mind that using different compiler versions together with compiler plugins may lead to Kotlin compiler exceptions. The Kotlin team plans to address these kinds of problems in future releases.

Try out the BTA with these plugins and send us your feedback in the dedicated YouTrack tickets for the KGP and the Maven plugin.

Standard library

In Kotlin 2.2.0, the `Base64` API and `HexFormat` API are now Stable.

Stable Base64 encoding and decoding

Kotlin 1.8.20 introduced Experimental support for Base64 encoding and decoding. In Kotlin 2.2.0, the Base64 API is now Stable and includes four encoding schemes, with the new `Base64.Pem` added in this release:

- `Base64.Default` uses the standard Base64 encoding scheme.

  > The `Base64.Default` is the companion object of the `Base64` class. As a result, you can call its functions with `Base64.encode()` and `Base64.decode()` instead of `Base64.Default.encode()` and `Base64.Default.decode()`.

- `Base64.UrlSafe` uses the "URL and Filename safe" encoding scheme.
- `Base64.Mime` uses the MIME encoding scheme, inserting a line separator every 76 characters during encoding and skipping illegal characters during decoding.
- `Base64.Pem` encodes data like `Base64.Mime` but limits the line length to 64 characters.

You can use the Base64 API to encode binary data into a Base64 string and decode it back into bytes.

Here's an example:

```
val foBytes = "fo".map { it.code.toByte() }.toByteArray()
Base64.Default.encode(foBytes) // "Zm8="
// Alternatively:
// Base64.encode(foBytes)

val foobarBytes = "foobar".map { it.code.toByte() }.toByteArray()
Base64.UrlSafe.encode(foobarBytes) // "Zm9vYmFy"

Base64.Default.decode("Zm8=") // foBytes
// Alternatively:
// Base64.decode("Zm8=")

Base64.UrlSafe.decode("Zm9vYmFy") // foobarBytes
```

On the JVM, use the `.encodingWith()` and `.decodingWith()` extension functions to encode and decode Base64 with input and output streams:

```
import kotlin.io.encoding.*
import java.io.ByteArrayOutputStream

fun main() {
    val output = ByteArrayOutputStream()
    val base64Output = output.encodingWith(Base64.Default)

    base64Output.use { stream ->
        stream.write("Hello World!!".encodeToByteArray())
    }

    println(output.toString())
    // SGVsbG8gV29ybGQhIQ==
}
```

Stable Hexadecimal parsing and formatting with the `HexFormat` API

The `HexFormat` API introduced in Kotlin 1.9.0 is now Stable. You can use it to convert between numerical values and hexadecimal strings.

For example:

```
fun main() {
        println(93.toHexString())
    //sampleEnd
}
```

For more information, see New HexFormat class to format and parse hexadecimals.

Compose compiler

In this release, the Compose compiler introduces support for composable function references and changes defaults for several feature flags.

Support for `@Composable` function references

The Compose compiler supports the declaration and usage of composable function references starting from the Kotlin 2.2.0 release:

```
val content: @Composable (String) -> Unit = ::Text

@Composable fun App() {
    content("My App")
}
```

Composable function references behave slightly differently from composable lambda objects at runtime. In particular, composable lambdas allow for finer control over skipping by extending the `ComposableLambda` class. Function references are expected to implement the `KCallable` interface, so the same optimization cannot be applied to them.

`PausableComposition` feature flag enabled by default

The `PausableComposition` feature flag is enabled by default starting from Kotlin 2.2.0. This flag adjusts the Compose compiler output for restartable functions, allowing runtime to force skipping behavior and therefore effectively pause composition by skipping each function. This allows heavy compositions to be split between frames, which will be used by prefetching in a future release.

To disable this feature flag, add the following to your Gradle configuration:

```
// build.gradle.kts
composeCompiler {
    featureFlag = setOf(ComposeFeatureFlag.PausableComposition.disabled())
}
```

`OptimizeNonSkippingGroups` feature flag enabled by default

The `OptimizeNonSkippingGroups` feature flag is enabled by default starting from Kotlin 2.2.0. This optimization improves runtime performance by removing group calls generated for non-skipping composable functions. It should not result in any observable behavior changes at runtime.

If you encounter any issues, you can validate that this change causes the issue by disabling the feature flag. Please report any issues to the Jetpack Compose issue tracker.

To disable the `OptimizeNonSkippingGroups` flag, add the following to your Gradle configuration:

```
composeCompiler {
    featureFlag = setOf(ComposeFeatureFlag.OptimizeNonSkippingGroups.disabled())
}
```

Deprecated feature flags

The `StrongSkipping` and `IntrinsicRemember` feature flags are now deprecated and will be removed in a future release. If you encounter any issues that make you disable these feature flags, please report them to the Jetpack Compose issue tracker.

Breaking changes and deprecations

This section highlights important breaking changes and deprecations worth noting. See our Compatibility guide for a complete overview of all breaking changes and deprecations in this release.

-

Starting with Kotlin 2.2.0, the compiler no longer supports `-language-version=1.6` or `-language-version=1.7`. Language feature sets older than 1.8 aren't supported but the language itself remains fully backward compatible with Kotlin 1.0.

- Support for the Ant build system is deprecated. Kotlin support for Ant hasn't been in active development for a long time, and there are no plans to maintain it further due to its relatively small user base. We plan to remove Ant support in 2.3.0.
- Kotlin 2.2.0 raises the deprecation level of the `kotlinOptions{}` block in Gradle to error. Use the `compilerOptions{}` block instead. For guidance on updating your build scripts, see Migrate from `kotlinOptions{}` to `compilerOptions{}`.
- Kotlin scripting remains an important part of Kotlin's ecosystem, but we're focusing on specific use cases such as custom scripting, as well as `gradle.kts` and `main.kts` scripts, to provide a better experience. To learn more, see our updated blog post. As a result, Kotlin 2.2.0 deprecates support for:
  - REPL: To continue to use REPL via `kotlinc`, opt in with the `-Xrepl` compiler option.
  - JSR-223: Since this JSR is in the **Withdrawn** state, the JSR-223 implementation continues to work with language version 1.9 but won't be migrated to use the K2 compiler in the future.
  - The `KotlinScriptMojo` Maven plugin: We didn't see enough traction with this plugin. You will see compiler warnings if you continue to use it.
- In Kotlin 2.2.0, the `setSource()` function in `KotlinCompileTool` now replaces configured sources instead of adding to them. If you want to add sources without replacing existing ones, use the `source()` function.
- The type of `annotationProcessorOptionProviders` in `BaseKapt` has been changed from `MutableList<Any>` to `MutableList<CommandLineArgumentProvider>`. If your code currently adds a list as a single element, use the `addAll()` function instead of the `add()` function.
- Following the deprecation of the dead code elimination (DCE) tool used in the legacy Kotlin/JS backend, the remaining DSLs related to DCE are now removed from the Kotlin Gradle plugin:
  - The `org.jetbrains.kotlin.gradle.dsl.KotlinJsDce` interface
  - The `org.jetbrains.kotlin.gradle.targets.js.dsl.KotlinJsBrowserDsl.dceTask(body: Action<KotlinJsDce>)` function
  - The `org.jetbrains.kotlin.gradle.dsl.KotlinJsDceCompilerToolOptions` interface
  - The `org.jetbrains.kotlin.gradle.dsl.KotlinJsDceOptions` interface

  The current JS IR compiler supports DCE out of the box, and the `@JsExport` annotation allows specifying which Kotlin functions and classes to retain during DCE.
- The deprecated `kotlin-android-extensions` plugin is removed in Kotlin 2.2.0. Use the `kotlin-parcelize` plugin for the `Parcelable` implementation generator and the Android Jetpack's view bindings for synthetic views instead.
- Experimental `kotlinArtifacts` API is deprecated in Kotlin 2.2.0. Use the current DSL available in the Kotlin Gradle plugin to build final native binaries. If it's not sufficient for migration, leave a comment in this YouTrack issue.
- `KotlinCompilation.source`, deprecated in Kotlin 1.9.0, is now removed from the Kotlin Gradle plugin.
- The parameters for experimental commonization modes are deprecated in Kotlin 2.2.0. Clear the commonization cache to delete invalid compilation artifacts.

- The deprecated `konanVersion` property is now removed from the `CInteropProcess` task. Use `CInteropProcess.kotlinNativeVersion` instead.
- Usage of the deprecated `destinationDir` property will now lead to an error. Use `CInteropProcess.destinationDirectory.set()` instead.

Documentation updates

This release brings notable documentation changes, including the migration of Kotlin Multiplatform documentation to the KMP portal.

Additionally, we created new pages and tutorials, and revamped existing ones.

New and revamped tutorials

- Kotlin intermediate tour – Take your understanding of Kotlin to the next level. Learn when to use extension functions, interfaces, classes, and more.
- Build a Kotlin app that uses Spring AI – Learn how to create a Kotlin app that answers questions using OpenAI and a vector database.
- Create a Spring Boot project with Kotlin – Learn how to create a Spring Boot project with Gradle using IntelliJ IDEA's **New Project** wizard.
- Mapping Kotlin and C tutorial series – Learn how different types and constructs are mapped between Kotlin and C.
- Create an app using C interop and libcurl – Create a simple HTTP client that can run natively using the libcurl C library.
- Create your Kotlin Multiplatform library – Learn how to create and publish a multiplatform library using IntelliJ IDEA.
- Build a full-stack application with Ktor and Kotlin Multiplatform – This tutorial now uses IntelliJ IDEA instead of Fleet, along with Material 3 and the latest versions of Ktor and Kotlin.
- Manage local resource environment in your Compose Multiplatform app – Learn how to manage the application's resource environment, like in-app theme and language.

New and revamped pages

- Kotlin for AI overview – Discover Kotlin's capabilities for building AI-powered applications.
- Dokka migration guide – Learn how to migrate to v2 of the Dokka Gradle plugin.
- Kotlin Metadata JVM library – Explore guidance on reading, modifying, and generating metadata for Kotlin classes compiled for the JVM.
- CocoaPods integration – Learn how to set up the environment, add Pod dependencies, or use a Kotlin project as a CocoaPod dependency through tutorials and sample projects.
- New pages for Compose Multiplatform to support the iOS stable release:
  - Navigation and Deep linking in particular.
  - Implementing layouts in Compose.
  - Localizing strings and other i18n pages like support for RTL languages.

- Compose Hot Reload – Learn how to use Compose Hot Reload with your desktop targets and how to add it to an existing project.
- Exposed migrations – Learn about the tools Exposed provides for managing database schema changes.

How to update to Kotlin 2.2.0

The Kotlin plugin is distributed as a bundled plugin in IntelliJ IDEA and Android Studio.

To update to the new Kotlin version, change the Kotlin version to 2.2.0 in your build scripts.

Compatibility guide for Kotlin 2.2.x

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 2.1 to Kotlin 2.2.

Basic terms

In this document, we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Drop support in `-language-version` for 1.6 and 1.7

Issue: KT-71793

Component: Compiler

Incompatible change type: source

Short summary: Starting with Kotlin 2.2, the compiler no longer supports `-language-version=1.6` or `-language-version=1.7`. This means that language feature sets older than 1.8 are no longer supported. However, the language itself remains fully backward compatible with Kotlin 1.0.

Deprecation cycle:

- 2.1.0: report a warning when using `-language-version` with versions 1.6 and 1.7
- 2.2.0: report a warning when using `-language-version` with versions 1.8 and 1.9; raise the warning to an error for versions 1.6 and 1.7

Enable invokedynamic for annotated lambdas by default

Issue: KTLC-278

Component: Core language

Incompatible change type: behavioral

Short summary: Lambdas with annotations now use `invokedynamic` through `LambdaMetafactory` by default, aligning their behavior with Java lambdas. This affects reflection-based code that relied on retrieving annotations from generated lambda classes. To revert to the old behavior, use the `-Xindy-allow-annotated-lambdas=false` compiler option.

Deprecation cycle:

- 2.2.0: enable `invokedynamic` for annotated lambdas by default

Prohibit constructor call and inheritance on type aliases with variance in expanded types in K2

Issue: KTLC-4

Component: Core language

Incompatible change type: source

Short summary: Constructor calls and inheritance using type aliases that expand to types that use variance modifiers such as `out` are no longer supported by the K2 compiler. This resolves inconsistencies where using the original type wasn't allowed, but the same usage through a type alias was permitted. To migrate, use the original type explicitly where needed.

Deprecation cycle:

- 2.0.0: report a warning for constructor calls or supertype usage on type aliases that expand to types with variance modifiers
- 2.2.0: raise the warning to an error

Prohibit synthetic properties from Kotlin getters

Issue: KTLC-272

Component: Core language

Incompatible change type: source

Short summary: Synthetic properties are no longer allowed for getters defined in Kotlin. This affects cases where Java classes extend Kotlin ones and when working with mapped types like `java.util.LinkedHashSet`. To migrate, replace property access with direct calls to the corresponding getter functions.

Deprecation cycle:

- 2.0.0: report a warning for accessing synthetic properties created from Kotlin getters
- 2.2.0: raise the warning to an error

Change default method generation for interface functions on JVM

Issue: KTLC-269

Component: Core language

Incompatible change type: binary

Short summary: Functions declared in interfaces are now compiled to JVM default methods unless configured otherwise. This may result in compilation errors in Java code when unrelated supertypes define conflicting implementations. The behavior is controlled by the stable `-jvm-default` compiler option, which replaces the now deprecated `-Xjvm-default` option. To restore the previous behavior, where default implementations are generated only in `DefaultImpls` classes and subclasses, use `-jvm-default=disable`.

Deprecation cycle:

- 2.2.0: `-jvm-default` compiler option is set to `enable` by default

Forbid field-targeted annotations on annotation properties

Issue: KTLC-7

Component: Core language

Incompatible change type: source

Short summary: Field-targeted annotations are no longer allowed on annotation properties. Although these annotations had no observable effect, this change may affect custom IR plugins that relied on them. To migrate, remove the field-targeted annotation from the property.

Deprecation cycle:

- 2.1.0: `@JvmField` annotation is deprecated with a warning on annotation properties
- 2.1.20: report a warning for all field-targeted annotations on annotation properties
- 2.2.0: raise the warning to an error

Forbid reified type parameters in type aliases

Issue: KTLC-5

Component: Core language

Incompatible change type: source

Short summary: The `reified` modifier is no longer allowed on type parameters in type aliases. Reified type parameters are only valid in inline functions, so using them in type aliases had no effect. To migrate, remove the `reified` modifier from `typealias` declarations.

Deprecation cycle:

- 2.1.0: report a warning for reified type parameters in type aliases
- 2.2.0: raise the warning to an error

Correct type checks on inline value classes for `Number` and `Comparable`

Issue: KTLC-21

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Inline value classes are no longer treated as implementors of `java.lang.Number` or `java.lang.Comparable` in `is` and `as` checks. These checks previously returned incorrect results when applied to boxed inline classes. The optimization now applies only to primitive types and their wrappers.

Deprecation cycle:

- 2.2.0: enable the new behavior

Prohibit inaccessible generic types from indirect dependencies

Issue: KTLC-3

Component: Core language

Incompatible change type: source

Short summary: The K2 compiler now reports errors when using types from indirect dependencies that are not visible to the compiler. This affects cases such as lambda parameters or generic type arguments, where the referenced type is not available due to a missing dependency.

Deprecation cycle:

- 2.0.0: report errors for inaccessible generic types in lambdas and selected usages of inaccessible generic type arguments; report warnings for inaccessible non-generic types in lambdas and inaccessible type arguments in expression and super types
- 2.1.0: raise the warning on inaccessible non-generic types in lambdas to an error
- 2.2.0: raise the warning on inaccessible type arguments in expression types to an error

Enforce visibility checks on type parameter bounds

Issue: KTLC-274

Component: Core language

Incompatible change type: source

Short summary: Functions and properties can no longer use a type parameter bound that has more restrictive visibility than the declaration itself. This prevents exposing inaccessible types indirectly, which previously compiled without errors but led to runtime failures or IR validation errors in some cases.

Deprecation cycle:

- 2.1.0: report a warning when a type parameter has a bound that is not visible from the declaration's visibility scope
- 2.2.0: raise the warning to an error

Report errors when exposing private types in non-private inline functions

Issue: KT-70916

Component: Core language

Incompatible change type: source

Short summary: Accessing private types, functions, or properties from non-private inline functions is no longer allowed. To migrate, either avoid referencing private entities, make the function private, or remove the `inline` modifier. Note that removing `inline` breaks binary compatibility.

Deprecation cycle:

- 2.2.0: report an error when accessing private types or members from non-private inline functions

Forbid non-local returns in lambdas used as parameter's default value

Issue: KTLC-286

Component: Core language

Incompatible change type: source

Short summary: Non-local return statements are no longer allowed in lambdas used as parameter's default value. This pattern previously compiled but led to runtime crashes. To migrate, rewrite the lambda to avoid non-local returns or move the logic outside the default value.

Deprecation cycle:

- 2.2.0: report an error for non-local returns in lambdas used as parameter's default value

Standard library

Deprecate `kotlin.native.Throws`

Issue: KT-72137

Component: Kotlin/Native

Incompatible change type: source

Short summary: `kotlin.native.Throws` is deprecated; use the common `kotlin.Throws` annotation instead.

Deprecation cycle:

- 1.9.0: report a warning when using `kotlin.native.Throws`
- 2.2.0: raise the warning to an error

Deprecate `AbstractDoubleTimeSource`

Issue: KT-72137

Component: kotlin-stdlib

Incompatible change type: source

Short summary: `AbstractDoubleTimeSource` is deprecated; use `AbstractLongTimeSource` instead.

Deprecation cycle:

- 1.8.20: report a warning when using `AbstractDoubleTimeSource`
- 2.2.0: raise the warning to an error

Tools

Correct `setSource()` function in `KotlinCompileTool` to replace sources

Issue: KT-59632

Component: Gradle

Incompatible change type: behavioral

Short summary: The `setSource()` function in the `KotlinCompileTool` interface now replaces configured sources instead of adding to them. If you want to add sources without replacing existing ones, use the `source()` function.

Deprecation cycle:

- 2.2.0: enable the new behavior

Deprecate `KotlinCompilationOutput#resourcesDirProvider` property

Issue: KT-70620

Component: Gradle

Incompatible change type: source

Short summary: The `KotlinCompilationOutput#resourcesDirProvider` property is deprecated. Use `KotlinSourceSet.resources` in your Gradle build script instead to add additional resource directories.

Deprecation cycle:

- 2.1.0: `KotlinCompilationOutput#resourcesDirProvider` is deprecated with a warning
- 2.2.0: raise the warning to an error

Deprecate `BaseKapt.annotationProcessorOptionProviders` property

Issue: KT-58009

Component: Gradle

Incompatible change type: source

Short summary: The `BaseKapt.annotationProcessorOptionProviders` property is deprecated in favor of `BaseKapt.annotationProcessorOptionsProviders`, which accepts a `ListProperty<CommandLineArgumentProvider>` instead of a `MutableList<Any>`. This clearly defines the expected element type and prevents runtime failures caused by adding incorrect elements, such as nested lists. If your current code adds a list as a single element, replace the `add()` function with the `addAll()` function.

Deprecation cycle:

- 2.2.0: enforce the new type in the API

Deprecate `kotlin-android-extensions` plugin

Issue: KT-72341

Component: Gradle

Incompatible change type: source

Short summary: The `kotlin-android-extensions` plugin is deprecated. Use a separate plugin, `kotlin-parcelize`, for the `Parcelable` implementation generator and the Android Jetpack's view bindings for synthetic views instead.

Deprecation cycle:

- 1.4.20: the plugin is deprecated
- 2.1.20: a configuration error is introduced, and no plugin code is executed
- 2.2.0: the plugin code is removed
- 2.4.0: remove the plugin ID

Deprecate `kotlinOptions` DSL

Issue: KT-54110

Component: Gradle

Incompatible change type: source

Short summary: The ability to configure compiler options through the `kotlinOptions` DSL and the related `KotlinCompile<KotlinOptions>` task interface is deprecated in favor of the new `compilerOptions` DSL. As part of this deprecation, all properties in the `kotlinOptions` interface are now also individually marked as deprecated. To migrate, use the `compilerOptions` DSL to configure compiler options. For guidance on the migration, see Migrate from `kotlinOptions {}` to `compilerOptions {}`.

Deprecation cycle:

- 2.0.0: report a warning for `kotlinOptions` DSL
- 2.2.0: raise the warning to an error and deprecate all properties in `kotlinOptions`

Remove `kotlin.incremental.useClasspathSnapshot` property

Issue: KT-62963

Component: Gradle

Incompatible change type: source

Short summary: The `kotlin.incremental.useClasspathSnapshot` Gradle property is removed. This property controlled the deprecated JVM history-based incremental compilation mode, which has been replaced by the classpath-based approach enabled by default since Kotlin 1.8.20.

Deprecation cycle:

- 2.0.20: deprecate the `kotlin.incremental.useClasspathSnapshot` property with a warning
- 2.2.0: remove the property

Deprecations to Kotlin scripting

Issues: KT-71685, KT-75632, KT-76196.

Component: Scripting

Incompatible change type: source

Short summary: Kotlin 2.2.0 deprecates support for:

- REPL: To continue to use REPL via `kotlinc`, opt in with the `-Xrepl` compiler option.
- JSR-223: Since the JSR is in **Withdrawn** state. The JSR-223 implementation continues to work with language version 1.9 but there are no plans to migrate to the K2 compiler in the future.
- The `KotlinScriptMojo` Maven plugin. You will see compiler warnings if you continue to use it.

For more information, see our blog post.

Deprecation cycle:

- 2.1.0: deprecate the use of REPL in `kotlinc` with a warning
- 2.2.0: to use REPL via `kotlinc`, opt in with the `-Xrepl` compiler option; deprecate JSR-223, support can be restored by switching to language version 1.9; deprecate the `KotlinScriptMojo` Maven plugin
- 2.4.0: remove Kotlin script execution through the `KotlinScriptMojo` Maven plugin

Deprecate disambiguation classifier properties

Issue: KT-58231

Component: Gradle

Incompatible change type: source

Short summary: Options that were used to control how the Kotlin Gradle plugin disambiguates source set names and IDE imports became obsolete. Therefore, the following properties are now deprecated in the `KotlinTarget` interface:

- `useDisambiguationClassifierAsSourceSetNamePrefix`
- `overrideDisambiguationClassifierOnIdeImport`

Deprecation cycle:

- 2.0.0: report a warning when the Gradle properties are used
- 2.1.0: raise this warning to an error
- 2.2.0: remove Gradle properties

Deprecate commonization parameters

Issue: KT-75161

Component: Gradle

Incompatible change type: source

Short summary: The parameters for experimental commonization modes are deprecated in the Kotlin Gradle plugin. These parameters may produce invalid compilation artifacts that are then cached. To delete affected artifacts:

1. Remove the following options from your `gradle.properties` file:

   ```
   kotlin.mpp.enableOptimisticNumberCommonization
   kotlin.mpp.enablePlatformIntegerCommonization
   ```

2. Clear the commonization cache in the `~/.konan/*/klib/commonized` directory or run the following command:

   ```
   ./gradlew cleanNativeDistributionCommonization
   ```

Deprecation cycle:

- 2.2.0: deprecate commonization parameters with an error
- 2.2.20: remove commonization parameters

Deprecate support for legacy metadata compilation

Issue: KT-61817

Component: Gradle

Incompatible change type: source

Short summary: Options that were used to set up hierarchical structure and create intermediate source sets between the common and intermediate source sets became obsolete. The following compiler options are removed:

- `isCompatibilityMetadataVariantEnabled`
- `withGranularMetadata`
- `isKotlinGranularMetadataEnabled`

Deprecation cycle:

- 2.2.0: remove compiler options from the Kotlin Gradle plugin

Deprecate `KotlinCompilation.source` API

Issue: KT-64991

Component: Gradle

Incompatible change type: source

Short summary: The access to `KotlinCompilation.source` API that allowed adding Kotlin source sets directly to the Kotlin compilation has been deprecated.

Deprecation cycle:

- 1.9.0: report a warning when `KotlinCompilation.source` is used
- 1.9.20: raise this warning to an error
- 2.2.0: remove `KotlinCompilation.source` from the Kotlin Gradle plugin; attempts to use it lead to "unresolved reference" errors during the buildscript compilation

Deprecate target presets APIs

Issue: KT-71698

Component: Gradle

Incompatible change type: source

Short summary: Target presets for Kotlin Multiplatform targets became obsolete; target DSL functions like `jvm()` or `iosSimulatorArm64()` now cover the same use cases. All presets-related APIs are deprecated:

- The `presets` property in `org.jetbrains.kotlin.gradle.dsl.KotlinMultiplatformExtension`
- The `org.jetbrains.kotlin.gradle.plugin.KotlinTargetPreset` interface and all its inheritors
- The `fromPreset` overloads

Deprecation cycle:

- 1.9.20: report a warning on any usages of the presets-related API
- 2.0.0: raise this warning to an error
- 2.2.0: remove the presets-related API from the public API of the Kotlin Gradle plugin; sources that still use it fail with "unresolved reference" errors, and binaries (for example, Gradle plugins) might fail with linkage errors unless recompiled against the latest versions of the Kotlin Gradle plugin

Deprecate Apple target shortcuts

Issue: KT-70615

Component: Gradle

Incompatible change type: source

Short summary: The `ios()`, `watchos()`, and `tvos()` target shortcuts are deprecated in Kotlin Multiplatform DSL. The shortcuts were designed to partially create a source set hierarchy for Apple targets. The Kotlin Multiplatform Gradle plugin now provides a built-in hierarchy template. Instead of shortcuts, specify the list of targets, and then the plugin automatically sets up intermediate source sets for them.

Deprecation cycle:

- 1.9.20: report a warning when target shortcuts are used; the default hierarchy template is enabled by default instead
- 2.1.0: report an error when target shortcuts are used
- 2.2.0: remove target shortcut DSL from the Kotlin Multiplatform Gradle plugin

Deprecate `publishAllLibraryVariants()` function

Issue: KT-60623

Component: Gradle

Incompatible change type: source

Short summary: The `publishAllLibraryVariants()` function is deprecated. It was designed to publish all build variants for Android targets. This approach is not recommended now, as it can cause problems with variant resolution, especially when multiple flavors and build types are used. Use the `publishLibraryVariants()` function, which specifies build variants, instead.

Deprecation cycle:

- 2.2.0: `publishAllLibraryVariants()` is deprecated

Deprecate `android` target

Issue: KT-71608

Component: Gradle

Incompatible change type: source

Short summary: The `android` target name is deprecated in the current Kotlin DSL. Use `androidTarget` instead.

Deprecation cycle:

- 1.9.0: introduce a deprecation warning when the `android` name is used in Kotlin Multiplatform projects
- 2.1.0: raise this warning to an error
- 2.2.0: remove the `android` target DSL from the Kotlin Multiplatform Gradle plugin

Deprecate `konanVersion` in `CInteropProcess`

Issue: KT-71069

Component: Gradle

Incompatible change type: source

Short summary: The `konanVersion` property in the `CInteropProcess` task is deprecated. Use `CInteropProcess.kotlinNativeVersion` instead.

Deprecation cycle:

- 2.1.0: report a warning when the `konanVersion` property is used
- 2.2.0: raise this warning to an error
- 2.3.0: remove the `konanVersion` property from the Kotlin Gradle plugin

Deprecate `destinationDir` in `CInteropProcess`

Issue: KT-71068

Component: Gradle

Incompatible change type: source

Short summary: The `destinationDir` property in the `CInteropProcess` task is deprecated. Use the `CInteropProcess.destinationDirectory.set()` function instead.

Deprecation cycle:

- 2.1.0: report a warning when the `destinationDir` property is used
- 2.2.0: raise this warning to an error
- 2.3.0: remove the `destinationDir` property from the Kotlin Gradle plugin

Deprecate `kotlinArtifacts` API

Issue: KT-74953

Component: Gradle

Incompatible change type: source

Short summary: The experimental `kotlinArtifacts` API is deprecated. Use the current DSL available in the Kotlin Gradle plugin to build final native binaries. If it's not sufficient for migration, leave a comment in this YouTrack issue.

Deprecation cycle:

- 2.2.0: report a warning when the `kotlinArtifacts` API is used
- 2.3.0: raise this warning to an error

What's new in Kotlin 2.1.20

Released: March 20, 2025

The Kotlin 2.1.20 release is here! Here are the main highlights:

- **K2 compiler updates**: Updates to the new kapt and Lombok plugins
- **Kotlin Multiplatform**: New DSL to replace Gradle's Application plugin
- **Kotlin/Native**: Support for Xcode 16.3 and a new inlining optimization
- **Kotlin/Wasm**: Default custom formatters, support for DWARF, and migration to Provider API
- **Gradle support**: Compatibility with Gradle's Isolated Projects and custom publication variants
- **Standard library**: Common atomic types, improved UUID support, and new time-tracking functionality
- **Compose compiler**: Relaxed restrictions on `@Composable` functions and other updates
- **Documentation**: Important improvements to the Kotlin documentation

For information about the Kotlin release cycle, see Kotlin release process.

IDE support

The Kotlin plugins that support 2.1.20 are bundled in the latest IntelliJ IDEA and Android Studio. You don't need to update the Kotlin plugin in your IDE. All you need to do is to change the Kotlin version to 2.1.20 in your build scripts.

See Update to a new release for details.

Download sources for Kotlin artifacts in projects with OSGi support

Sources of all dependencies of the `kotlin-osgi-bundle` library are now included in its distribution. This allows IntelliJ IDEA to download these sources to provide documentation for Kotlin symbols and improve the debugging experience.

Kotlin K2 compiler

We're continuing to improve plugin support for the new Kotlin K2 compiler. This release brings updates to the new kapt and Lombok plugins.

Beta

New default kapt plugin

Starting with Kotlin 2.1.20, the K2 implementation of the kapt compiler plugin is enabled by default for all the projects.

The JetBrains team launched the new implementation of the kapt plugin with the K2 compiler back in Kotlin 1.9.20. Since then, we have further developed the internal implementation of K2 kapt and made its behavior similar to that of the K1 version, while significantly improving its performance as well.

If you encounter any issues when using kapt with the K2 compiler, you can temporarily revert to the previous plugin implementation.

To do this, add the following option to the `gradle.properties` file of your project:

```
kapt.use.k2=false
```

Please report any issues to our issue tracker.

Experimental

Lombok compiler plugin: support for `@SuperBuilder` and updates on `@Builder`

The Kotlin Lombok compiler plugin now supports the `@SuperBuilder` annotation, making it easier to create builders for class hierarchies. Previously, developers using Lombok in Kotlin had to manually define builders when working with inheritance. With `@SuperBuilder`, the builder automatically inherits superclass fields, allowing you to initialize them when constructing an object.

Additionally, this update includes several improvements and bug fixes:

- The `@Builder` annotation now works on constructors, allowing more flexible object creation. For more details, see the corresponding YouTrack issue.
- Several issues related to Lombok's code generation in Kotlin have been resolved, improving overall compatibility. For more details, see the GitHub changelog.

For more information about the `@SuperBuilder` annotation, see the official Lombok documentation.

Experimental

Kotlin Multiplatform: new DSL to replace Gradle's Application plugin

Starting with Gradle 8.7, the Application plugin is no longer compatible with the Kotlin Multiplatform Gradle plugin. Kotlin 2.1.20 introduces an Experimental DSL to achieve similar functionality. The new `executable {}` block configures execution tasks and Gradle distributions for JVM targets.

Before the `executable {}` block in your build script, add the following `@OptIn` annotation:

```
@OptIn(ExperimentalKotlinGradlePluginApi::class)
```

For example:

```
kotlin {
    jvm {
        @OptIn(ExperimentalKotlinGradlePluginApi::class)
        binaries {
            // Configures a JavaExec task named "runJvm" and a Gradle distribution for the "main"
compilation in this target
            executable {
                mainClass.set("foo.MainKt")
            }

            // Configures a JavaExec task named "runJvmAnother" and a Gradle distribution for the
"main" compilation
            executable(KotlinCompilation.MAIN_COMPILATION_NAME, "another") {
                // Set a different class
                mainClass.set("foo.MainAnotherKt")
            }

            // Configures a JavaExec task named "runJvmTest" and a Gradle distribution for the "test"
compilation
            executable(KotlinCompilation.TEST_COMPILATION_NAME) {
                mainClass.set("foo.MainTestKt")
            }

            // Configures a JavaExec task named "runJvmTestAnother" and a Gradle distribution for the
"test" compilation
            executable(KotlinCompilation.TEST_COMPILATION_NAME, "another") {
                mainClass.set("foo.MainAnotherTestKt")
            }
        }
    }
}
```

In this example, Gradle's Distribution plugin is applied on the first `executable {}` block.

If you run into any issues, report them in our issue tracker or let us know in our public Slack channel.

Kotlin/Native

Support for Xcode 16.3

Starting with Kotlin **2.1.21**, the Kotlin/Native compiler supports Xcode 16.3 – the latest stable version of Xcode. Feel free to update your Xcode and continue working on your Kotlin projects for Apple operating systems.

The 2.1.21 release also fixes the related cinterop issue that caused compilation failures in Kotlin Multiplatform projects.

Experimental

New inlining optimization

Kotlin 2.1.20 introduces a new inlining optimization pass, which comes before the actual code generation phase.

The new inlining pass in the Kotlin/Native compiler should perform better than the standard LLVM inliner and improve the runtime performance of the generated code.

The new inlining pass is currently Experimental. To try it out, use the following compiler option:

```
-Xbinary=preCodegenInlineThreshold=40
```

Our experiments show that setting the threshold to 40 tokens (code units parsed by the compiler) provides a reasonable compromise for compilation optimization. According to our benchmarks, this gives an overall performance improvement of 9.5%. Of course, you can try out other values, too.

If you experience increased binary size or compilation time, please report such issues via YouTrack.

Kotlin/Wasm

This release improves Kotlin/Wasm debugging and property usage. Custom formatters now work out of the box in development builds, while DWARF debugging facilitates code inspection. Additionally, the Provider API simplifies property usage in Kotlin/Wasm and Kotlin/JS.

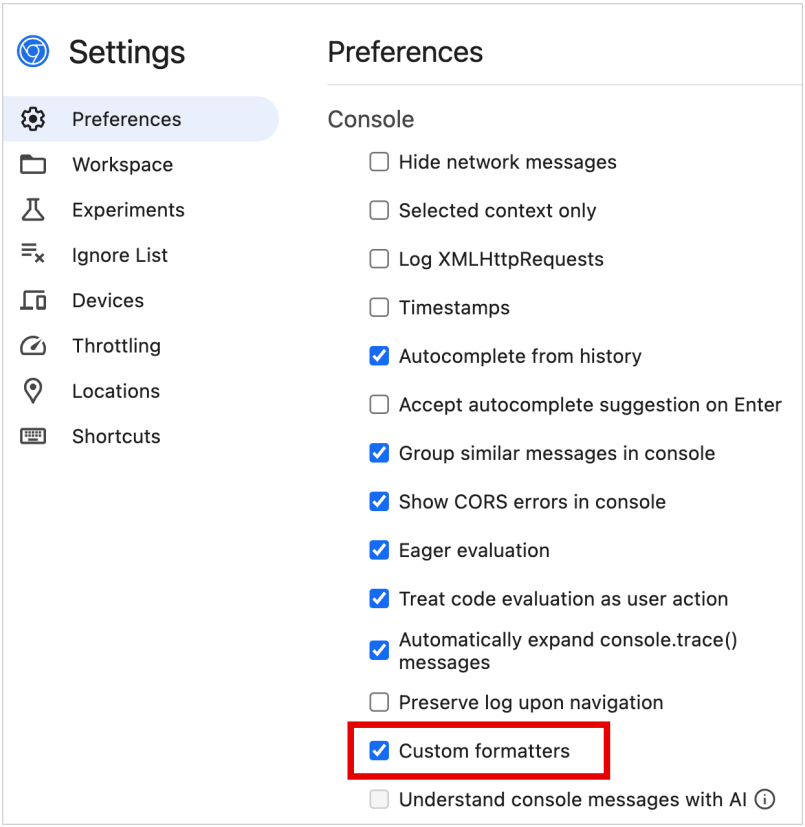
Custom formatters enabled by default

Before, you had to manually configure custom formatters to improve debugging in web browsers when working with Kotlin/Wasm code.

In this release, custom formatters are enabled by default in development builds, so you don't need additional Gradle configurations.

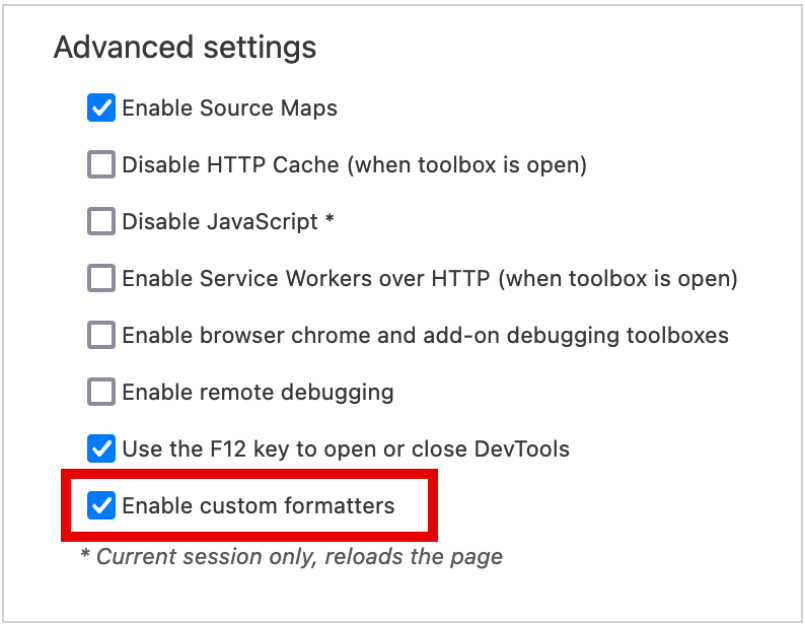
To use this feature, you only need to ensure that custom formatters are enabled in your browser's developer tools:

- In Chrome DevTools, find the custom formatters checkbox in **Settings | Preferences | Console**:



Enable custom formatters in Chrome

- In Firefox DevTools, find the custom formatters checkbox in **Settings | Advanced settings**:



Enable custom formatters in Firefox

This change primarily affects Kotlin/Wasm development builds. If you have specific requirements for production builds, you need to adjust your Gradle configuration accordingly. To do so, add the following compiler option to the `wasmJs {}` block:

```
// build.gradle.kts
kotlin {
    wasmJs {
        // ...

        compilerOptions {
            freeCompilerArgs.add("-Xwasm-debugger-custom-formatters")
        }
    }
}
```

Support for DWARF to debug Kotlin/Wasm code

Kotlin 2.1.20 introduces support for DWARF (debugging with arbitrary record format) in Kotlin/Wasm.

With this change, the Kotlin/Wasm compiler is able to embed DWARF data into the generated WebAssembly (Wasm) binary. Many debuggers and virtual machines can read this data to provide insights into the compiled code.

DWARF is mainly useful for debugging Kotlin/Wasm applications inside standalone Wasm virtual machines (VMs). To use this feature, the Wasm VM and debugger must support DWARF.

With DWARF support, you can step through Kotlin/Wasm applications, inspect variables, and gain code insights. To enable this feature, use the following compiler option:

```
-Xwasm-generate-dwarf
```

Migration to Provider API for Kotlin/Wasm and Kotlin/JS properties

Previously, properties in Kotlin/Wasm and Kotlin/JS extensions were mutable (`var`) and assigned directly in build scripts:

```
the<NodeJsExtension>().version = "2.0.0"
```

Now, properties are exposed through the Provider API, and you must use the `.set()` function to assign values:

```
the<NodeJsEnvSpec>().version.set("2.0.0")
```

The Provider API ensures that values are lazily computed and properly integrated with task dependencies, improving build performance.

With this change, direct property assignments are deprecated in favor of `*EnvSpec` classes, such as `NodeJsEnvSpec` and `YarnRootEnvSpec` .

Additionally, several alias tasks have been removed to avoid confusion:

Deprecated task	Replacement
`wasmJsRun`	`wasmJsBrowserDevelopmentRun`
`wasmJsBrowserRun`	`wasmJsBrowserDevelopmentRun`
`wasmJsNodeRun`	`wasmJsNodeDevelopmentRun`

Deprecated task	Replacement
`wasmJsBrowserWebpack`	`wasmJsBrowserProductionWebpack` or `wasmJsBrowserDistribution`
`jsRun`	`jsBrowserDevelopmentRun`
`jsBrowserRun`	`jsBrowserDevelopmentRun`
`jsNodeRun`	`jsNodeDevelopmentRun`
`jsBrowserWebpack`	`jsBrowserProductionWebpack` or `jsBrowserDistribution`

If you only use Kotlin/JS or Kotlin/Wasm in build scripts, no action is required as Gradle automatically handles assignments.

However, if you maintain a plugin based on the Kotlin Gradle Plugin, and your plugin does not apply `kotlin-dsl`, you must update property assignments to use the `.set()` function.

Gradle

Kotlin 2.1.20 is fully compatible with Gradle 7.6.3 through 8.11. You can also use Gradle versions up to the latest Gradle release. However, be aware that doing so may result in deprecation warnings, and some new Gradle features might not work.

This version of Kotlin includes compatibility of Kotlin Gradle plugins with Gradle's Isolated Projects as well as support for custom Gradle publication variants.

Experimental

Kotlin Gradle plugins compatible with Gradle's Isolated Projects

> This feature is currently in a pre-Alpha state in Gradle. JS and Wasm targets are not supported at the moment. Use it only with Gradle version 8.10 or higher and solely for evaluation purposes.

Since Kotlin 2.1.0, you've been able to preview Gradle's Isolated Projects feature in your projects.

Previously, you had to configure the Kotlin Gradle plugin to make your project compatible with the Isolated Projects feature before you could try it out. In Kotlin 2.1.20, this additional step is no longer necessary.

Now, to enable the Isolated Projects feature, you only need to set the system property.

Gradle's Isolated Projects feature is supported in Kotlin Gradle plugins for both multiplatform projects and projects that contain only the JVM or Android target.

Specifically for multiplatform projects, if you notice problems with your Gradle build after upgrading, you can opt out of the new Kotlin Gradle plugin behavior by adding:

```
kotlin.kmp.isolated-projects.support=disable
```

However, if you use this Gradle property in your multiplatform project, you can't use the Isolated Projects feature.

Let us know about your experience with this feature in YouTrack.

Experimental

Support for adding custom Gradle publication variants

Kotlin 2.1.20 introduces support for adding custom Gradle publication variants. This feature is available for multiplatform projects and projects targeting the JVM.

You cannot modify existing Gradle variants with this feature.

This feature is Experimental. To opt in, use the `@OptIn(ExperimentalKotlinGradlePluginApi::class)` annotation.

To add a custom Gradle publication variant, invoke the `adhocSoftwareComponent()` function, which returns an instance of `AdhocComponentWithVariants` that you can configure in the Kotlin DSL:

```
plugins {
    // Only JVM and Multiplatform are supported
    kotlin("jvm")
    // or
    kotlin("multiplatform")
}


kotlin {
    @OptIn(ExperimentalKotlinGradlePluginApi::class)
    publishing {
        // Returns an instance of AdhocSoftwareComponent
        adhocSoftwareComponent()
        // Alternatively, you can configure AdhocSoftwareComponent in the DSL block as follows
        adhocSoftwareComponent {
            // Add your custom variants here using the AdhocSoftwareComponent API
        }
    }
}
```

For more information on variants, see Gradle's Customizing publishing guide.

Standard library

This release brings new Experimental features to the standard library: common atomic types, improved support for UUIDs, and new time-tracking functionality.

Experimental

Common atomic types

In Kotlin 2.1.20, we are introducing common atomic types in the standard library's `kotlin.concurrent.atomics` package, enabling shared, platform-independent code for thread-safe operations. This simplifies development for Kotlin Multiplatform projects by removing the need to duplicate atomic-dependent logic across source sets.

The `kotlin.concurrent.atomics` package and its properties are Experimental. To opt in, use the `@OptIn(ExperimentalAtomicApi::class)` annotation or the compiler option `-opt-in=kotlin.ExperimentalAtomicApi`.

Here's an example that shows how you can use `AtomicInt` to safely count processed items across multiple threads:

```
// Imports the necessary libraries
import kotlin.concurrent.atomics.*
import kotlinx.coroutines.*

@OptIn(ExperimentalAtomicApi::class)
suspend fun main() {
    // Initializes the atomic counter for processed items
    var processedItems = AtomicInt(0)
    val totalItems = 100
    val items = List(totalItems) { "item$it" }
    // Splits the items into chunks for processing by multiple coroutines
    val chunkSize = 20
    val itemChunks = items.chunked(chunkSize)
    coroutineScope {
        for (chunk in itemChunks) {
            launch {
                for (item in chunk) {
                    println("Processing $item in thread ${Thread.currentThread()}")
                    processedItems += 1 // Increment counter atomically
                }
            }
        }
    }
    // Prints the total number of processed items
    println("Total processed items: ${processedItems.load()}")
}
```

To enable seamless interoperability between Kotlin's atomic types and Java's `java.util.concurrent.atomic` atomic types, the API provides the `.asJavaAtomic()` and `.asKotlinAtomic()` extension functions. On the JVM, Kotlin atomics and Java atomics are the same types in runtime, so you can transform Java atomics into Kotlin atomics and vice versa without any overhead.

Here's an example that shows how Kotlin and Java atomic types can work together:

```
// Imports the necessary libraries
import kotlin.concurrent.atomics.*
import java.util.concurrent.atomic.*

@OptIn(ExperimentalAtomicApi::class)
fun main() {
    // Converts Kotlin's AtomicInt to Java's AtomicInteger
    val kotlinAtomic = AtomicInt(42)
    val javaAtomic: AtomicInteger = kotlinAtomic.asJavaAtomic()
    println("Java atomic value: ${javaAtomic.get()}")
    // Java atomic value: 42

    // Converts Java's AtomicInteger back to Kotlin's AtomicInt
    val kotlinAgain: AtomicInt = javaAtomic.asKotlinAtomic()
    println("Kotlin atomic value: ${kotlinAgain.load()}")
    // Kotlin atomic value: 42
}
```

Experimental

Changes in UUID parsing, formatting, and comparability

The JetBrains team continues to improve the support for UUIDs introduced to the standard library in 2.0.20.

Previously, the `parse()` function only accepted UUIDs in the hex-and-dash format. With Kotlin 2.1.20, you can use `parse()` for *both* the hex-and-dash and the plain hexadecimal (without dashes) formats.

We've also introduced functions specific to operations with the hex-and-dash format in this release:

- `parseHexDash()` parses UUIDs from the hex-and-dash format.
- `toHexDashString()` converts a `Uuid` into a `String` in the hex-and-dash format (mirroring the functionality of `toString()`).

These functions work similarly to `parseHex()` and `toHexString()`, which were introduced earlier for the hexadecimal format. Explicit naming for parsing and formatting functionality should improve code clarity and your overall experience with UUIDs.

UUIDs in Kotlin are now `Comparable`. Starting with Kotlin 2.1.20, you can directly compare and sort values of the `Uuid` type. This enables the use of the `<` and `>` operators and standard library extensions available exclusively for `Comparable` types or their collections (such as `sorted()`), and it also allows passing UUIDs to any functions or APIs that require the `Comparable` interface.

Remember that the UUID support in the standard library is still Experimental. To opt in, use the `@OptIn(ExperimentalUuidApi::class)` annotation or the compiler option `-opt-in=kotlin.uuid.ExperimentalUuidApi`:

```
import kotlin.uuid.ExperimentalUuidApi
import kotlin.uuid.Uuid

@OptIn(ExperimentalUuidApi::class)
fun main() {
    // parse() accepts a UUID in a plain hexadecimal format
    val uuid = Uuid.parse("550e8400e29b41d4a716446655440000")

    // Converts it to the hex-and-dash format
    val hexDashFormat = uuid.toHexDashString()

    // Outputs the UUID in the hex-and-dash format
    println(hexDashFormat)

    // Outputs UUIDs in ascending order
    println(
        listOf(
            uuid,
            Uuid.parse("780e8400e29b41d4a716446655440005"),
            Uuid.parse("5ab88400e29b41d4a716446655440076")
        ).sorted()
    )
}
```

Experimental

New time tracking functionality

Starting with Kotlin 2.1.20, the standard library provides the ability to represent a moment in time. This functionality was previously only available in `kotlinx-datetime`, an official Kotlin library.

The `kotlinx.datetime.Clock` interface is introduced to the standard library as `kotlin.time.Clock` and the `kotlinx.datetime.Instant` class as `kotlin.time.Instant`. These concepts naturally align with the `time` package in the standard library because they're only concerned with moments in time compared to a more complex calendar and timezone functionality that remains in `kotlinx-datetime`.

`Instant` and `Clock` are useful when you need precise time tracking without considering time zones or dates. For example, you can use them to log events with timestamps, measure durations between two points in time, and obtain the current moment for system processes.

To provide interoperability with other languages, additional converter functions are available:

- `.toKotlinInstant()` converts a time value to a `kotlin.time.Instant` instance.
- `.toJavaInstant()` converts the `kotlin.time.Instant` value to a `java.time.Instant` value.
- `Instant.toJSDate()` converts the `kotlin.time.Instant` value to an instance of the JS `Date` class. This conversion is not precise; JS uses millisecond precision to represent dates, while Kotlin allows for nanosecond resolution.

The new time features of the standard library are still Experimental. To opt in, use the `@OptIn(ExperimentalTime::class)` annotation:

```
import kotlin.time.*

@OptIn(ExperimentalTime::class)
fun main() {

    // Get the current moment in time
    val currentInstant = Clock.System.now()
    println("Current time: $currentInstant")

    // Find the difference between two moments in time
    val pastInstant = Instant.parse("2023-01-01T00:00:00Z")
    val duration = currentInstant - pastInstant

    println("Time elapsed since 2023-01-01: $duration")
}
```

For more information on the implementation, see this KEEP proposal.

Compose compiler

In 2.1.20, the Compose compiler relaxes some restrictions on `@Composable` functions introduced in previous releases. In addition, the Compose compiler Gradle plugin is set to include source information by default, aligning the behavior on all platforms with Android.

Support for parameters with default values in open `@Composable` functions

The compiler previously restricted parameters with default values in open `@Composable` functions due to incorrect compiler output, which would result in crashes at runtime. The underlying issue is now resolved, and parameters with default values are fully supported when used with Kotlin 2.1.20 or newer.

Compose compiler allowed parameters with default values in open functions before version 1.5.8, so the support depends on project configuration:

- If an open composable function is compiled with Kotlin version 2.1.20 or newer, the compiler generates correct wrappers for parameters with default values. This includes wrappers compatible with pre-1.5.8 binaries, meaning that downstream libraries will also be able to use this open function.
- If the open composable function is compiled with Kotlin older than 2.1.20, Compose uses a compatibility mode, which might result in runtime crashes. When using the compatibility mode, the compiler emits a warning to highlight potential problems.

Final overridden functions are allowed to be restartable

Virtual functions (overrides of `open` and `abstract`, including interfaces) were forced to be non-restartable with the 2.1.0 release. This restriction is now relaxed for functions that are members of final classes or are `final` themselves – they will be restarted or skipped as usual.

You might observe some behavior changes in affected functions after upgrading to Kotlin 2.1.20. To force non-restartable logic from the previous version, apply the `@NonRestartableComposable` annotation to the function.

`ComposableSingletons` removed from public API

`ComposableSingletons` is a class created by the Compose compiler when optimizing `@Composable` lambdas. Lambdas that do not capture any parameters are allocated once and cached in a property of the class, saving allocations during runtime. The class is generated with internal visibility and is only intended for optimizing lambdas inside a compilation unit (usually a file).

However, this optimization was also applied to `inline` function bodies, which resulted in singleton lambda instances leaking into the public API. To fix this problem, starting with 2.1.20, `@Composable` lambdas are no longer optimized into singletons inside inline functions. At the same time, the Compose compiler will continue generating singleton classes and lambdas for inline functions to support binary compatibility for modules that were compiled under the previous model.

Source information included by default

The Compose compiler Gradle plugin already has the including source information feature enabled by default on Android. Starting with Kotlin 2.1.20, this feature will be enabled by default on all platforms.

Remember to check if you set this option using `freeCompilerArgs`. This method can cause the build to fail when used alongside the plugin, due to an option being effectively set twice.

Breaking changes and deprecations

- To align Kotlin Multiplatform with upcoming changes in Gradle, we are phasing out the `withJava()` function. Java source sets are now created by default. If you use the Java test fixtures Gradle plugin, upgrade directly to Kotlin 2.1.21 to avoid compatibility issues.
- The JetBrains team is proceeding with the deprecation of the `kotlin-android-extensions` plugin. If you try to use it in your project, you'll now get a configuration error, and no plugin code will be executed.
- The legacy `kotlin.incremental.classpath.snapshot.enabled` property has been removed from the Kotlin Gradle plugin. The property used to provide an opportunity to fall back to a built-in ABI snapshot on the JVM. The plugin now uses other methods to detect and avoid unnecessary recompilations, making the property obsolete.

Documentation updates

The Kotlin documentation has received some notable changes:

Revamped and new pages

- Kotlin roadmap – see the updated list of Kotlin's priorities on language and ecosystem evolution.

- Gradle best practices page – learn essential best practices for optimizing your Gradle builds and improving performance.
- Compose Multiplatform and Jetpack Compose – an overview of the relation between the two UI frameworks.
- Kotlin Multiplatform and Flutter – see the comparison of two popular cross-platform frameworks.
- Interoperability with C – explore the details of Kotlin's interoperability with C.
- Numbers – learn about different Kotlin types for representing numbers.

New and updated tutorials

- Publish your library to Maven Central – learn how to publish KMP library artifacts to the most popular Maven repository.
- Kotlin/Native as a dynamic library – create a dynamic Kotlin library.
- Kotlin/Native as an Apple framework – create your own framework and use Kotlin/Native code from Swift/Objective-C applications on macOS and iOS.

How to update to Kotlin 2.1.20

Starting from IntelliJ IDEA 2023.3 and Android Studio Iguana (2023.2.1) Canary 15, the Kotlin plugin is distributed as a bundled plugin included in your IDE. This means that you can't install the plugin from JetBrains Marketplace anymore.

To update to the new Kotlin version, change the Kotlin version to 2.1.20 in your build scripts.

What's new in Kotlin 2.1.0

Released: November 27, 2024

The Kotlin 2.1.0 release is here! Here are the main highlights:

- **New language features in preview**: Guard conditions in `when` with a subject, non-local `break` and `continue`, and multi-dollar string interpolation.
- **K2 compiler updates**: More flexibility around compiler checks and improvements to the kapt implementation.
- **Kotlin Multiplatform**: Introduced basic support for Swift export, stable Gradle DSL for compiler options, and more.
- **Kotlin/Native**: Improved support for `iosArm64` and other updates.
- **Kotlin/Wasm**: Multiple updates, including support for incremental compilation.
- **Gradle support**: Improved compatibility with newer versions of Gradle and the Android Gradle plugin, along with updates to the Kotlin Gradle plugin API.
- **Documentation**: Significant improvements to the Kotlin documentation.

For information about the Kotlin release cycle, see Kotlin release process.

IDE support

The Kotlin plugins that support 2.1.0 are bundled in the latest IntelliJ IDEA and Android Studio. You don't need to update the Kotlin plugin in your IDE. All you need to do is change the Kotlin version to 2.1.0 in your build scripts.

See Update to a new Kotlin version for details.

Language

After the Kotlin 2.0.0 release with the K2 compiler, the JetBrains team is focusing on improving the language with new features. In this release, we are excited to announce several new language design improvements.

These features are available in preview, and we encourage you to try them and share your feedback:

- Guard conditions in `when` with a subject
- Non-local `break` and `continue`
- Multi-dollar interpolation: improved handling of `$` in string literals

> All the features have IDE support in the latest 2024.3 version of IntelliJ IDEA with K2 mode enabled.
>
> Learn more in the IntelliJ IDEA 2024.3 blog post.

See the full list of Kotlin language design features and proposals.

This release also brings the following language updates:

- Support for requiring opt-in to extend APIs
- Improved overload resolution for functions with generic types
- Improved exhaustiveness checks for when expressions with sealed classes

Guard conditions in when with a subject

> This feature is In preview, and opt-in is required (see details below).
>
> We would appreciate your feedback in YouTrack.

Starting from 2.1.0, you can use guard conditions in `when` expressions or statements with subjects.

Guard conditions allow you to include more than one condition for the branches of a `when` expression, making complex control flows more explicit and concise as well as flattening the code structure.

To include a guard condition in a branch, place it after the primary condition, separated by `if`:

```
sealed interface Animal {
    data class Cat(val mouseHunter: Boolean) : Animal {
        fun feedCat() {}
    }

    data class Dog(val breed: String) : Animal {
        fun feedDog() {}
    }
}

fun feedAnimal(animal: Animal) {
    when (animal) {
        // Branch with only the primary condition. Calls `feedDog()` when `animal` is `Dog`
        is Animal.Dog -> animal.feedDog()
        // Branch with both primary and guard conditions. Calls `feedCat()` when `animal` is `Cat` and
is not `mouseHunter`
        is Animal.Cat if !animal.mouseHunter -> animal.feedCat()
        // Prints "Unknown animal" if none of the above conditions match
        else -> println("Unknown animal")
    }
}
```

In a single `when` expression, you can combine branches with and without guard conditions. The code in a branch with a guard condition runs only if both the primary condition and the guard condition are `true`. If the primary condition does not match, the guard condition is not evaluated. Additionally, guard conditions support `else if`.

To enable guard conditions in your project, use the following compiler option in the command line:

```
kotlinc -Xwhen-guards main.kt
```

Or add it to the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xwhen-guards")
    }
}
```

Non-local break and continue

> This feature is In preview, and opt-in is required (see details below).
>
> We would appreciate your feedback in YouTrack.

Kotlin 2.1.0 adds a preview of another long-awaited feature, the ability to use non-local `break` and `continue`. This feature expands the toolset you can use in the scope of inline functions and reduces boilerplate code in your project.

Previously, you could use only non-local returns. Now, Kotlin also supports `break` and `continue` jump expressions non-locally. This means that you can apply them within lambdas passed as arguments to an inline function that encloses the loop:

```
fun processList(elements: List<Int>): Boolean {
    for (element in elements) {
        val variable = element.nullableMethod() ?: run {
            log.warning("Element is null or invalid, continuing...")
            continue
        }
        if (variable == 0) return true // If variable is zero, return true
    }
    return false
}
```

To try the feature in your project, use the `-Xnon-local-break-continue` compiler option in the command line:

```
kotlinc -Xnon-local-break-continue main.kt
```

Or add it in the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xnon-local-break-continue")
    }
}
```

We're planning to make this feature Stable in future Kotlin releases. If you encounter any issues when using non-local `break` and `continue`, please report them to our issue tracker.

Multi-dollar string interpolation

> The feature is In preview and opt-in is required (see details below).
>
> We would appreciate your feedback in YouTrack.

Kotlin 2.1.0 introduces support for multi-dollar string interpolation, improving how the dollar sign (`$`) is handled within string literals. This feature is helpful in contexts that require multiple dollar signs, such as templating engines, JSON schemas, or other data formats.

String interpolation in Kotlin uses a single dollar sign. However, using a literal dollar sign in a string, which is common in financial data and templating systems, required workarounds such as `${'$'}`. With the multi-dollar interpolation feature enabled, you can configure how many dollar signs trigger interpolation, with fewer dollar signs being treated as string literals.

Here's an example of how to generate an JSON schema multi-line string with placeholders using `$` :

```
val KClass<*>.jsonSchema : String
    get() = $$"""
    {
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "$id": "https://example.com/product.schema.json",
      "$dynamicAnchor": "meta"
      "title": "$${simpleName ?: qualifiedName ?: "unknown"}",
      "type": "object"
    }
    """
```

In this example, the initial `$$` means that you need **two dollar signs** (`$$`) to trigger interpolation. It prevents `$schema` , `$id` , and `$dynamicAnchor` from being interpreted as interpolation markers.

This approach is especially helpful when working with systems that use dollar signs for placeholder syntax.

To enable the feature, use the following compiler option in the command line:

```
kotlinc -Xmulti-dollar-interpolation main.kt
```

Alternatively, update the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xmulti-dollar-interpolation")
    }
}
```

If your code already uses standard string interpolation with a single dollar sign, no changes are needed. You can use `$$` whenever you need literal dollar signs in your strings.

Support for requiring opt-in to extend APIs

Kotlin 2.1.0 introduces the `@SubclassOptInRequired` annotation, which allows library authors to require an explicit opt-in before users can implement experimental interfaces or extend experimental classes.

This feature can be useful when a library API is stable enough to use but might evolve with new abstract functions, making it unstable for inheritance.

To add the opt-in requirement to an API element, use the `@SubclassOptInRequired` annotation with a reference to the annotation class:

```
@RequiresOptIn(
level = RequiresOptIn.Level.WARNING,
message = "Interfaces in this library are experimental"
)
annotation class UnstableApi()

@SubclassOptInRequired(UnstableApi::class)
interface CoreLibraryApi
```

In this example, the `CoreLibraryApi` interface requires users to opt in before they can implement it. A user can opt in like this:

```
@OptIn(UnstableApi::class)
interface MyImplementation: CoreLibraryApi
```

> When you use the `@SubclassOptInRequired` annotation to require opt-in, the requirement is not propagated to any inner or nested classes.

For a real-world example of how to use the `@SubclassOptInRequired` annotation in your API, check out the `SharedFlow` interface in the `kotlinx.coroutines` library.

Improved overload resolution for functions with generic types

Previously, if you had a number of overloads for a function where some had value parameters of a generic type and others had function types in the same position, the resolution behavior could sometimes be inconsistent.

This led to different behavior depending on whether your overloads were member functions or extension functions. For example:

```
class KeyValueStore<K, V> {
    fun store(key: K, value: V) {} // 1
    fun store(key: K, lazyValue: () -> V) {} // 2
}

fun <K, V> KeyValueStore<K, V>.storeExtension(key: K, value: V) {} // 1
fun <K, V> KeyValueStore<K, V>.storeExtension(key: K, lazyValue: () -> V) {} // 2

fun test(kvs: KeyValueStore<String, Int>) {
    // Member functions
    kvs.store("", 1)    // Resolves to 1
    kvs.store("") { 1 } // Resolves to 2

    // Extension functions
    kvs.storeExtension("", 1)    // Resolves to 1
    kvs.storeExtension("") { 1 } // Doesn't resolve
}
```

In this example, the `KeyValueStore` class has two overloads for the `store()` function, where one overload has function parameters with generic types `K` and `V`, and another has a lambda function that returns a generic type `V`. Similarly, there are two overloads for the extension function: `storeExtension()`.

When the `store()` function was called with and without a lambda function, the compiler successfully resolved the correct overloads. However, when the extension function `storeExtension()` was called with a lambda function, the compiler didn't resolve the correct overload because it incorrectly considered both overloads to be applicable.

To fix this problem, we've introduced a new heuristic so that the compiler can discard a possible overload when a function parameter with a generic type can't accept a lambda function based on information from a different argument. This change makes the behavior of member functions and extension functions consistent, and it is enabled by default in Kotlin 2.1.0.

Improved exhaustiveness checks for when expressions with sealed classes

In previous versions of Kotlin, the compiler required an `else` branch in `when` expressions for type parameters with sealed upper bounds, even when all cases in the `sealed class` hierarchy were covered. This behavior is addressed and improved in Kotlin 2.1.0, making exhaustiveness checks more powerful and allowing you to remove redundant `else` branches, keeping `when` expressions cleaner and more intuitive.

Here's an example demonstrating the change:

```
sealed class Result
object Error: Result()
class Success(val value: String): Result()

fun <T : Result> render(result: T) = when (result) {
    Error -> "Error!"
    is Success -> result.value
    // Requires no else branch
}
```

Kotlin K2 compiler

With Kotlin 2.1.0, the K2 compiler now provides more flexibility when working with compiler checks and warnings, as well as improved support for the kapt plugin.

Extra compiler checks

With Kotlin 2.1.0, you can now enable additional checks in the K2 compiler. These are extra declaration, expression, and type checks that are usually not crucial for compilation but can still be useful if you want to validate the following cases:

Check type	Comment
`REDUNDANT_NULLABLE`	`Boolean??` is used instead of `Boolean?`
`PLATFORM_CLASS_MAPPED_TO_KOTLIN`	`java.lang.String` is used instead of `kotlin.String`
`ARRAY_EQUALITY_OPERATOR_CAN_BE_REPLACED_WITH_EQUALS`	`arrayOf("") == arrayOf("")` is used instead of `arrayOf("").contentEquals(arrayOf(""))`
`REDUNDANT_CALL_OF_CONVERSION_METHOD`	`42.toInt()` is used instead of `42`
`USELESS_CALL_ON_NOT_NULL`	`"".orEmpty()` is used instead of `""`
`REDUNDANT_SINGLE_EXPRESSION_STRING_TEMPLATE`	`"$string"` is used instead of `string`
`UNUSED_ANONYMOUS_PARAMETER`	A parameter is passed in the lambda expression but never used
`REDUNDANT_VISIBILITY_MODIFIER`	`public class Klass` is used instead of `class Klass`
`REDUNDANT_MODALITY_MODIFIER`	`final class Klass` is used instead of `class Klass`
`REDUNDANT_SETTER_PARAMETER_TYPE`	`set(value: Int)` is used instead of `set(value)`
`CAN_BE_VAL`	`var local = 0` is defined but never reassigned, can be `val local = 42` instead
`ASSIGNED_VALUE_IS_NEVER_READ`	`val local = 42` is defined but never used afterward in the code
`UNUSED_VARIABLE`	`val local = 0` is defined but never used in the code
`REDUNDANT_RETURN_UNIT_TYPE`	`fun foo(): Unit {}` is used instead of `fun foo() {}`
`UNREACHABLE_CODE`	Code statement is present but can never be executed

If the check is true, you'll receive a compiler warning with a suggestion on how to fix the problem.

Extra checks are disabled by default. To enable them, use the `-Wextra` compiler option in the command line or specify `extraWarnings` in the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        extraWarnings.set(true)
    }
}
```

For more information on how to define and use compiler options, see Compiler options in the Kotlin Gradle plugin.

Global warning suppression

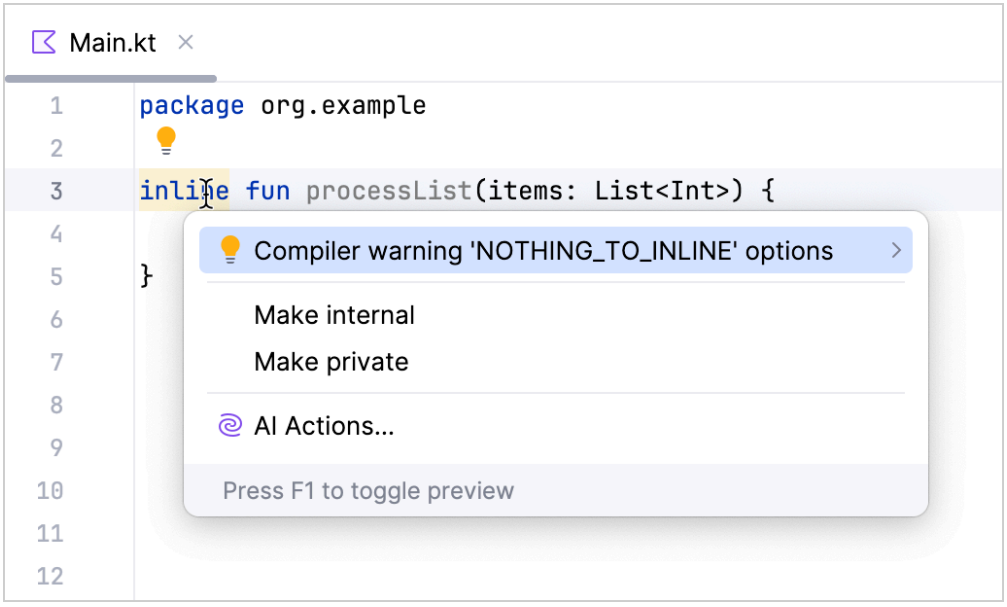
In 2.1.0, the Kotlin compiler has received a highly requested feature – the ability to suppress warnings globally.

You can now suppress specific warnings in the whole project by using the `-Xsuppress-warning=WARNING_NAME` syntax in the command line or the `freeCompilerArgs` attribute in the `compilerOptions {}` block of your build file.

For example, if you have extra compiler checks enabled in your project but want to suppress one of them, use:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        extraWarnings.set(true)
        freeCompilerArgs.add("-Xsuppress-warning=CAN_BE_VAL")
    }
}
```

If you want to suppress a warning but don't know its name, select the element and click the light bulb icon (or use Cmd + Enter/Alt + Enter):



Warning name intention

The new compiler option is currently Experimental. The following details are also worth noting:

- Error suppression is not allowed.

- If you pass an unknown warning name, compilation will result in an error.
- You can specify several warnings at once:

Command line

```
kotlinc -Xsuppress-warning=NOTHING_TO_INLINE -Xsuppress-warning=NO_TAIL_CALLS_FOUND main.kt
```

Build file

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.addAll(
            listOf(
                "-Xsuppress-warning=NOTHING_TO_INLINE",
                "-Xsuppress-warning=NO_TAIL_CALLS_FOUND"
            )
        )
    }
}
```

Improved K2 kapt implementation

The kapt plugin for the K2 compiler (K2 kapt) is in Alpha. It may be changed at any time.

We would appreciate your feedback in YouTrack.

Currently, projects using the kapt plugin work with the K1 compiler by default, supporting Kotlin versions up to 1.9.

In Kotlin 1.9.20, we launched an experimental implementation of the kapt plugin with the K2 compiler (K2 kapt). We have now improved K2 kapt's internal implementation to mitigate technical and performance issues.

While the new K2 kapt implementation doesn't introduce new features, its performance has significantly improved compared to the previous K2 kapt implementation. Additionally, the K2 kapt plugin's behavior is now much closer to that of K1 kapt.

To use the new K2 kapt plugin implementation, enable it just like you did the previous K2 kapt plugin. Add the following option to the `gradle.properties` file of your project:

```
kapt.use.k2=true
```

In upcoming releases, the K2 kapt implementation will be enabled by default instead of K1 kapt, so you will no longer need to enable it manually.

We highly appreciate your feedback before the new implementation is stabilized.

Resolution for overload conflicts between unsigned and non-primitive types

This release addresses the issue of resolution for overload conflicts that could occur in previous versions when functions were overloaded for unsigned and non-primitive types, as in the following examples:

Overloaded extension functions

```
fun Any.doStuff() = "Any"
fun UByte.doStuff() = "UByte"

fun main() {
    val uByte: UByte = UByte.MIN_VALUE
    uByte.doStuff() // Overload resolution ambiguity before Kotlin 2.1.0
}
```

In earlier versions, calling `uByte.doStuff()` led to ambiguity because both the `Any` and `UByte` extensions were applicable.

Overloaded top-level functions

```
fun doStuff(value: Any) = "Any"
fun doStuff(value: UByte) = "UByte"

fun main() {
    val uByte: UByte = UByte.MIN_VALUE
    doStuff(uByte) // Overload resolution ambiguity before Kotlin 2.1.0
}
```

Similarly, the call to `doStuff(uByte)` was ambiguous because the compiler couldn't decide whether to use the `Any` or `UByte` version. With 2.1.0, the compiler now handles these cases correctly, resolving the ambiguity by giving precedence to the more specific type, in this case `UByte`.

Kotlin/JVM

Starting with version 2.1.0, the compiler can generate classes containing Java 23 bytecode.

Change of JSpecify nullability mismatch diagnostics severity to strict

Kotlin 2.1.0 enforces strict handling of nullability annotations from `org.jspecify.annotations`, improving type safety for Java interoperability.

The following nullability annotations are affected:

- `org.jspecify.annotations.Nullable`
- `org.jspecify.annotations.NonNull`
- `org.jspecify.annotations.NullMarked`
- Legacy annotations in `org.jspecify.nullness` (JSpecify 0.2 and earlier)

Starting from Kotlin 2.1.0, nullability mismatches are raised from warnings to errors by default. This ensures that annotations like `@NonNull` and `@Nullable` are enforced during type checks, preventing unexpected nullability issues at runtime.

The `@NullMarked` annotation also affects the nullability of all members within its scope, making the behavior more predictable when you're working with annotated Java code.

Here's an example demonstrating the new default behavior:

```java
// Java
import org.jspecify.annotations.*;
public class SomeJavaClass {
    @NonNull
    public String foo() { // ...
    }

    @Nullable
    public String bar() { // ...
    }
}
```

```kotlin
// Kotlin
fun test(sjc: SomeJavaClass) {
    // Accesses a non-null result, which is allowed
    sjc.foo().length

    // Raises an error in the default strict mode because the result is nullable
    // To avoid the error, use ?.length instead
    sjc.bar().length
}
```

You can manually control the severity of diagnostics for these annotations. To do so, use the `-Xnullability-annotations` compiler option to choose a mode:

- `ignore` : Ignore nullability mismatches.
- `warning` : Report warnings for nullability mismatches.
- `strict` : Report errors for nullability mismatches (default mode).

For more information, see Nullability annotations.

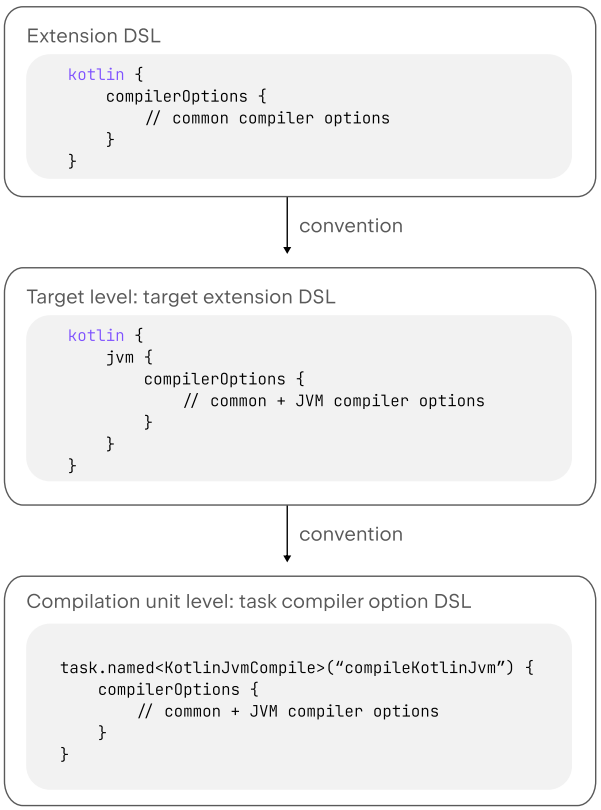
Kotlin Multiplatform

Kotlin 2.1.0 introduces basic support for Swift export and makes publishing Kotlin Multiplatform libraries easier. It also focuses on improvements around Gradle that stabilize the new DSL for configuring compiler options and bring a preview of the Isolated Projects feature.

New Gradle DSL for compiler options in multiplatform projects promoted to Stable

In Kotlin 2.0.0, we introduced a new Experimental Gradle DSL to simplify the configuration of compiler options across your multiplatform projects. In Kotlin 2.1.0, this DSL has been promoted to Stable.

The overall project configuration now has three layers. The highest is the extension level, then the target level, and the lowest is the compilation unit (which is usually a compilation task):



Kotlin compiler options levels

To learn more about the different levels and how compiler options can be configured between them, see Compiler options.

Preview Gradle's Isolated Projects in Kotlin Multiplatform

This feature is Experimental and is currently in a pre-Alpha state in Gradle. Use it only with Gradle version 8.10 and solely for evaluation purposes. The feature may be dropped or changed at any time.

We would appreciate your feedback on it in YouTrack. Opt-in is required (see details below).

In Kotlin 2.1.0, you can preview Gradle's Isolated Projects feature in your multiplatform projects.

The Isolated Projects feature in Gradle improves build performance by "isolating" configuration of individual Gradle projects from each other. Each project's build logic is restricted from directly accessing the mutable state of other projects, allowing them to safely run in parallel. To support this feature, we made some changes to the Kotlin Gradle plugin's model, and we are interested in hearing about your experiences during this preview phase.

There are two ways to enable the Kotlin Gradle plugin's new model:

- Option 1: **Testing compatibility without enabling Isolated Projects** – To check compatibility with the Kotlin Gradle plugin's new model without enabling the Isolated Projects feature, add the following Gradle property in the `gradle.properties` file of your project:

```
# gradle.properties
kotlin.kmp.isolated-projects.support=enable
```

- Option 2: **Testing with Isolated Projects enabled** – Enabling the Isolated Projects feature in Gradle automatically configures the Kotlin Gradle plugin to use the new model. To enable the Isolated Projects feature, set the system property. In this case, you don't need to add the Gradle property for the Kotlin Gradle plugin to your project.

Basic support for Swift export

> This feature is currently in the early stages of development. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Version 2.1.0 takes the first step towards providing support for Swift export in Kotlin, allowing you to export Kotlin sources directly to the Swift interface without using Objective-C headers. This should make multiplatform development for Apple targets easier.

The current basic support includes the ability to:

- Export multiple Gradle modules from Kotlin directly to Swift.
- Define custom Swift module names with the `moduleName` property.
- Set collapse rules for the package structure with the `flattenPackage` property.

You can use the following build file in your project as a starting point for setting up Swift export:

```
// build.gradle.kts
kotlin {

    iosX64()
    iosArm64()
    iosSimulatorArm64()

    @OptIn(ExperimentalSwiftExportDsl::class)
    swiftExport {
        // Root module name
        moduleName = "Shared"

        // Collapse rule
        // Removes package prefix from generated Swift code
        flattenPackage = "com.example.sandbox"

        // Export external modules
        export(project(":subproject")) {
            // Exported module name
            moduleName = "Subproject"
            // Collapse exported dependency rule
            flattenPackage = "com.subproject.library"
        }
    }
}
```

You can also clone our public sample with Swift export already set up.

The compiler automatically generates all the necessary files (including `swiftmodule` files, static `a` library, and header and `modulemap` files) and copies them into the app's build directory, which you can access from Xcode.

How to enable Swift export

Keep in mind that the feature is currently only in the early stages of development.

Swift export currently works in projects that use direct integration to connect the iOS framework to the Xcode project. This is a standard configuration for Kotlin Multiplatform projects created in Android Studio or through the web wizard.

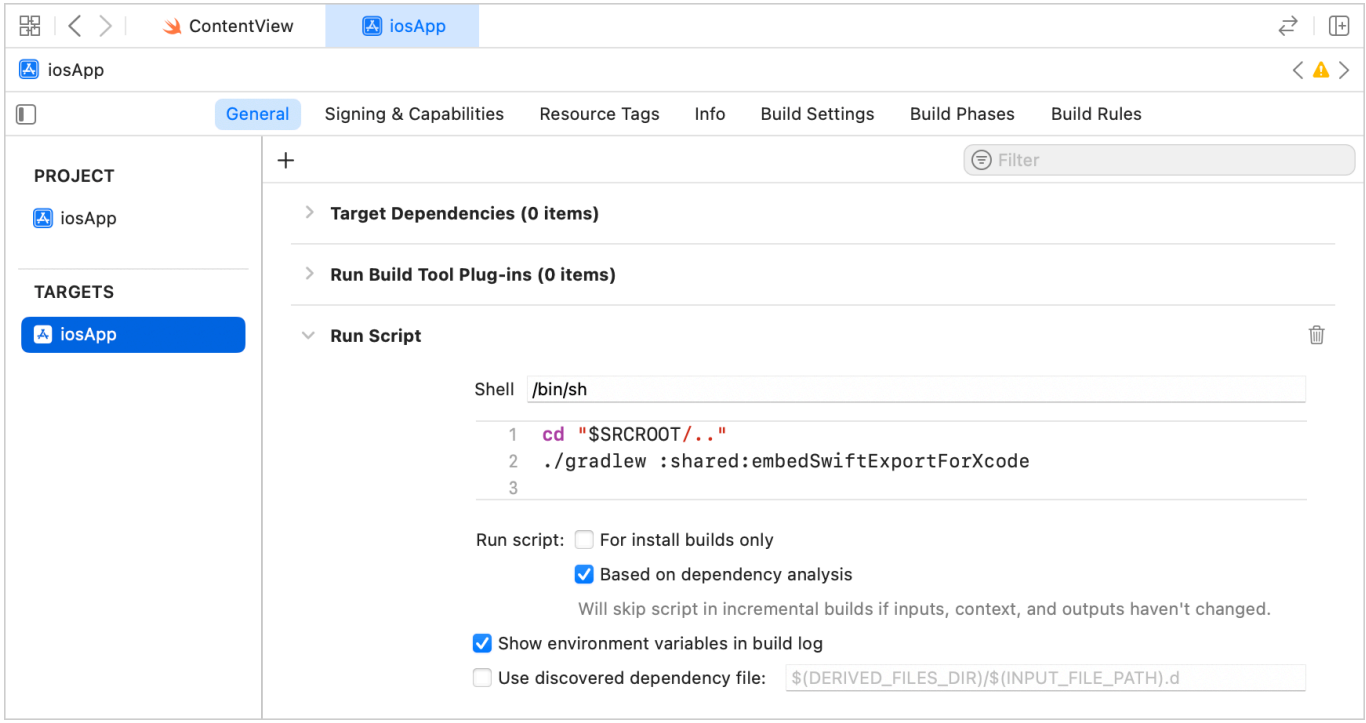
To try out Swift export in your project:

1. Add the following Gradle option to your `gradle.properties` file:

```
# gradle.properties
kotlin.experimental.swift-export.enabled=true
```

2. In Xcode, open the project settings.
3. On the **Build Phases** tab, locate the **Run Script** phase with the `embedAndSignAppleFrameworkForXcode` task.
4. Adjust the script to feature the `embedSwiftExportForXcode` task instead in the run script phase:

```
./gradlew :<Shared module name>:embedSwiftExportForXcode
```



Add the Swift export script

Leave feedback on Swift export

We're planning to expand and stabilize Swift export support in future Kotlin releases. Please leave your feedback in this YouTrack issue.

Ability to publish Kotlin libraries from any host

> This feature is currently Experimental. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

The Kotlin compiler produces `.klib` artifacts for publishing Kotlin libraries. Previously, you could get the necessary artifacts from any host, except for Apple platform targets that required a Mac machine. That put a special restraint on Kotlin Multiplatform projects that targeted iOS, macOS, tvOS, and watchOS targets.

Kotlin 2.1.0 lifts this restriction, adding support for cross-compilation. Now you can use any supported host to produce `.klib` artifacts, which should greatly simplify the publishing process for Kotlin and Kotlin Multiplatform libraries.

How to enable publishing libraries from any host

To try cross-compilation out in your project, add the following binary option to your `gradle.properties` file:

```
# gradle.properties
kotlin.native.enableKlibsCrossCompilation=true
```

This feature is currently Experimental and has some limitations. You still need to use a Mac machine if:

- Your library has a cinterop dependency.
- You have CocoaPods integration set up in your project.
- You need to build or test final binaries for Apple targets.

Leave feedback on publishing libraries from any host

We're planning to stabilize this feature and further improve library publication in future Kotlin releases. Please leave your feedback in our issue tracker YouTrack.

For more information, see Publishing multiplatform libraries.

Support for non-packed klibs

Kotlin 2.1.0 makes it possible to generate non-packed `.klib` file artifacts. This gives you the option to configure dependencies on klibs directly rather than unpack them first.

This change can also improve performance, decreasing compilation and linking time in your Kotlin/Wasm, Kotlin/JS, and Kotlin/Native projects.

For example, our benchmark shows a performance improvement of roughly 3% in total build time on the project with 1 linking and 10 compilation tasks (the project builds a single native executable binary that depends on 9 simplified projects). However, the actual impact on build time depends on both the number of subprojects and their respective sizes.

How to set up your project

By default, Kotlin compilation and linking tasks are now configured to use the new non-packed artifacts.

If you have set up custom build logic for resolving klibs and want to use the new unpacked artifacts, you need to explicitly specify the preferred variant of klib package resolution in your Gradle build file:

```
// build.gradle.kts
import org.jetbrains.kotlin.gradle.plugin.attributes.KlibPackaging
// ...
val resolvableConfiguration = configurations.resolvable("resolvable") {

    // For the new non-packed configuration:
    attributes.attribute(KlibPackaging.ATTRIBUTE, project.objects.named(KlibPackaging.NON_PACKED))

    // For the previous packed configuration:
    attributes.attribute(KlibPackaging.ATTRIBUTE, project.objects.named(KlibPackaging.PACKED))
}
```

Non-packed `.klib` files are generated at the same path in your project's build directory as the packed ones previously were. In turn, packed klibs are now located in the `build/libs` directory.

If no attribute is specified, the packed variant is used. You can check the list of available attributes and variants with the following console command:

```
./gradlew outgoingVariants
```

We would appreciate your feedback on this feature in YouTrack.

Further deprecation of old `android` target

In Kotlin 2.1.0, the deprecation warning for the old `android` target name has been raised to an error.

Currently, we recommend using the `androidTarget` option in your Kotlin Multiplatform projects targeting Android. This is a temporary solution that is necessary to free the `android` name for the upcoming Android/KMP plugin from Google.

We'll provide further migration instructions when the new plugin is available. The new DSL from Google will be the preferred option for Android target support in Kotlin Multiplatform.

For more information, see the Kotlin Multiplatform compatibility guide.

Dropped support for declaring multiple targets of the same type

Before Kotlin 2.1.0, you could declare multiple targets of the same type in your multiplatform projects. However, this made it challenging to distinguish between targets and to support shared source sets effectively. In most cases, a simpler setup, such as using separate Gradle projects, works better. For detailed guidance and an example of how to migrate, see Declaring several similar targets in the Kotlin Multiplatform compatibility guide.

Kotlin 1.9.20 triggered a deprecation warning if you declared multiple targets of the same type in your multiplatform projects. In Kotlin 2.1.0, this deprecation warning is now an error for all targets except Kotlin/JS ones. To learn more about why Kotlin/JS targets are exempt, see this issue in YouTrack.

Kotlin/Native

Kotlin 2.1.0 includes an upgrade for the `iosArm64` target support, improved cinterop caching process, and other updates.

iosArm64 promoted to Tier 1

The `iosArm64` target, which is crucial for Kotlin Multiplatform development, has been promoted to Tier 1. This is the highest level of support in the Kotlin/Native compiler.

This means the target is regularly tested on the CI pipeline to ensure that it's able to compile and run. We also provide source and binary compatibility between compiler releases for the target.

For more information on target tiers, see Kotlin/Native target support.

LLVM update from 11.1.0 to 16.0.0

In Kotlin 2.1.0, we updated LLVM from version 11.1.0 to 16.0.0. The new version includes bug fixes and security updates. In certain cases, it also provides compiler optimizations and faster compilation.

If you have Linux targets in your project, take note that the Kotlin/Native compiler now uses the `lld` linker by default for all Linux targets.

This update shouldn't affect your code, but if you encounter any issues, please report them to our issue tracker.

Changes to caching in cinterop

In Kotlin 2.1.0, we're making changes to the cinterop caching process. It no longer has the `CacheableTask` annotation type. The new recommended approach is to use the `cacheIf` output type to cache the results of the task.

This should resolve issues where `UP-TO-DATE` checks failed to detect changes to header files specified in the definition file, preventing the build system from recompiling the code.

Deprecation of the mimalloc memory allocator

Back in Kotlin 1.9.0, we introduced the new memory allocator, and then we enabled it by default in Kotlin 1.9.20. The new allocator has been designed to make garbage collection more efficient and improve the Kotlin/Native memory manager's runtime performance.

The new memory allocator replaced the previous default allocator, mimalloc. Now, it's time to deprecate mimalloc in the Kotlin/Native compiler.

You can now remove the `-Xallocator=mimalloc` compiler option from your build scripts. If you encounter any issues, please report them to our issue tracker.

For more information on the memory allocator and garbage collection in Kotlin, see Kotlin/Native memory management.

Kotlin/Wasm

Kotlin/Wasm received multiple updates along with support for incremental compilation.

Support for incremental compilation

Previously, when you changed something in your Kotlin code, the Kotlin/Wasm toolchain had to recompile the entire codebase.

Starting from 2.1.0, incremental compilation is supported for Wasm targets. In development tasks, the compiler now recompiles only files relevant to changes from the last compilation, which noticeably reduces the compilation time.

This change currently doubles the compilation speed, and there are plans to improve it further in future releases.

In the current setup, incremental compilation for Wasm targets is disabled by default. To enable incremental compilation, add the following line to your project's `local.properties` or `gradle.properties` file:

```
# gradle.properties
kotlin.incremental.wasm=true
```

Try out Kotlin/Wasm incremental compilation and share your feedback. Your insights will help make this feature Stable and enabled by default sooner.

Browser APIs moved to the kotlinx-browser stand-alone library

Previously, the declarations for web APIs and related target utilities were part of the Kotlin/Wasm standard library.

In this release, the `org.w3c.*` declarations have been moved from the Kotlin/Wasm standard library to the new kotlinx-browser library. This library also includes other web-related packages, such as `org.khronos.webgl`, `kotlin.dom`, and `kotlinx.browser`.

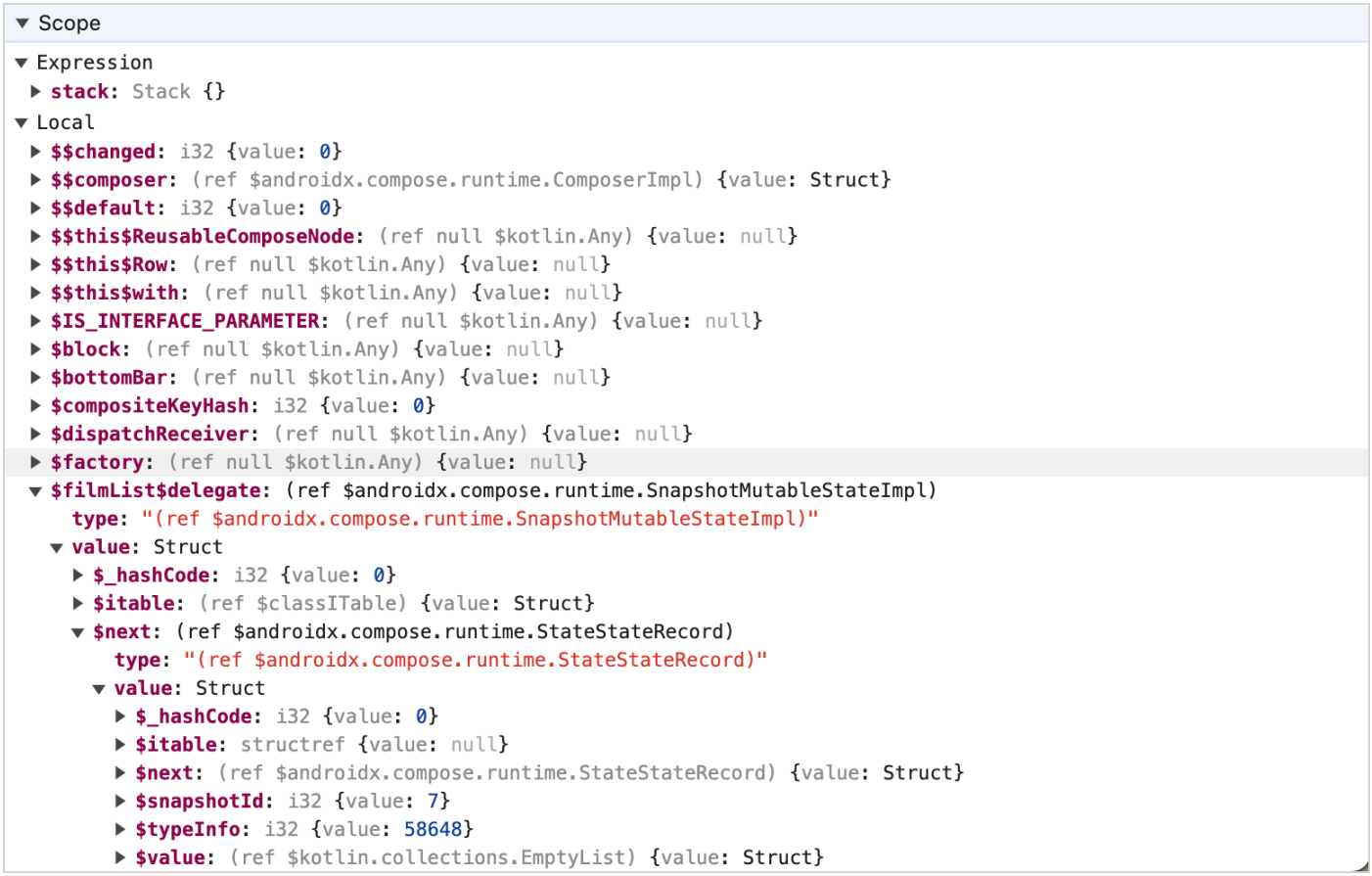
This separation provides modularity, enabling independent updates for web-related APIs outside of Kotlin's release cycle. Additionally, the Kotlin/Wasm standard library now contains only declarations available in any JavaScript environment.

To use the declarations from the moved packages, you need to add the `kotlinx-browser` dependency to your project's build configuration file:

```
// build.gradle.kts
val wasmJsMain by getting {
    dependencies {
        implementation("org.jetbrains.kotlinx:kotlinx-browser:0.3")
    }
}
```

Improved debugging experience for Kotlin/Wasm

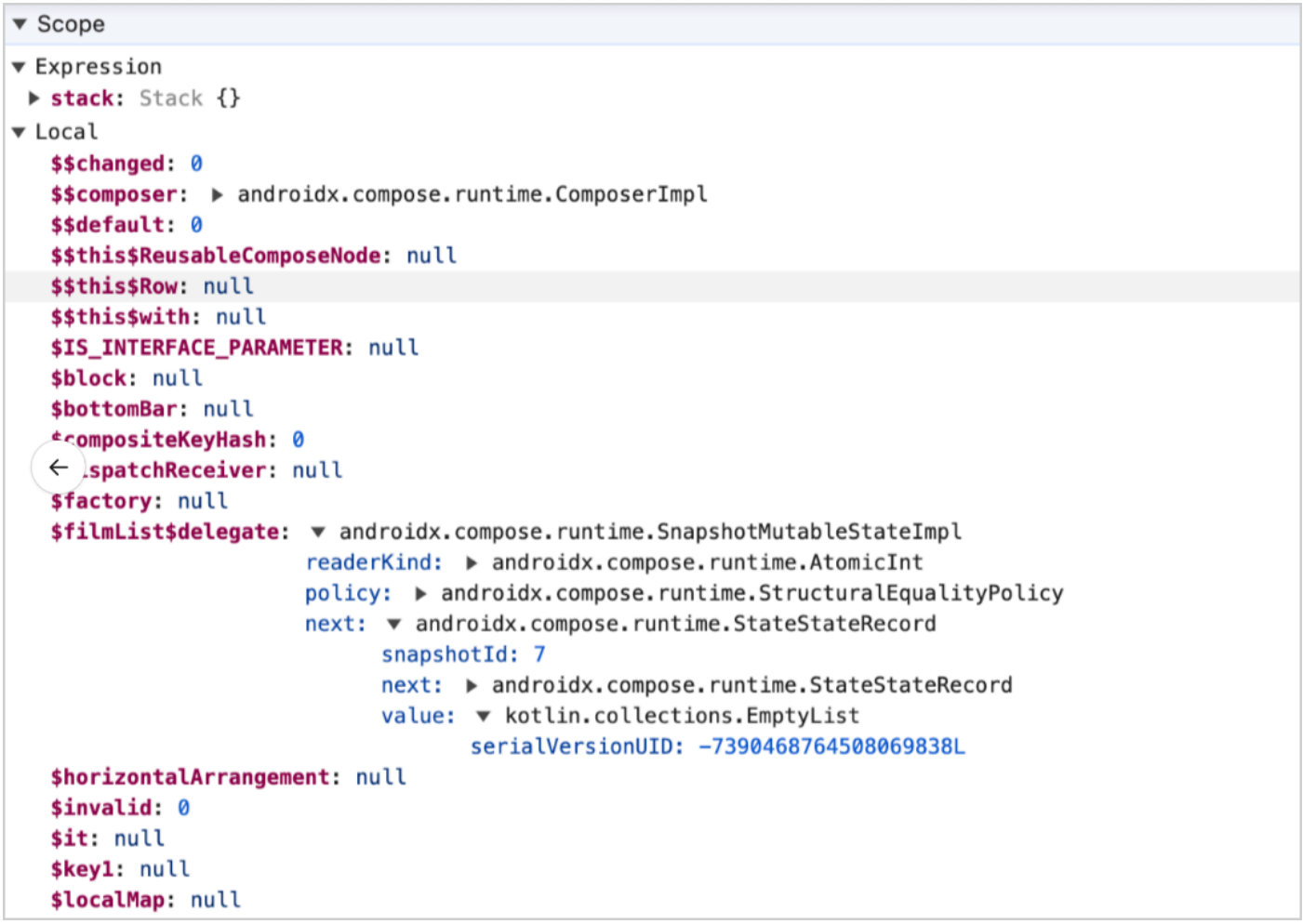
Previously, when debugging Kotlin/Wasm code in web browsers, you might have encountered a low-level representation of variable values in the debugging interface. This often made it challenging to track the current state of the application.



Kotlin/Wasm old debugger

To improve this experience, custom formatters have been added in the variable view. The implementation uses the custom formatters API, which is supported across major browsers like Firefox and Chromium-based ones.

With this change, you can now display and locate variable values in a more user-friendly and comprehensible manner.



Kotlin/Wasm improved debugger

To try the new debugging experience:

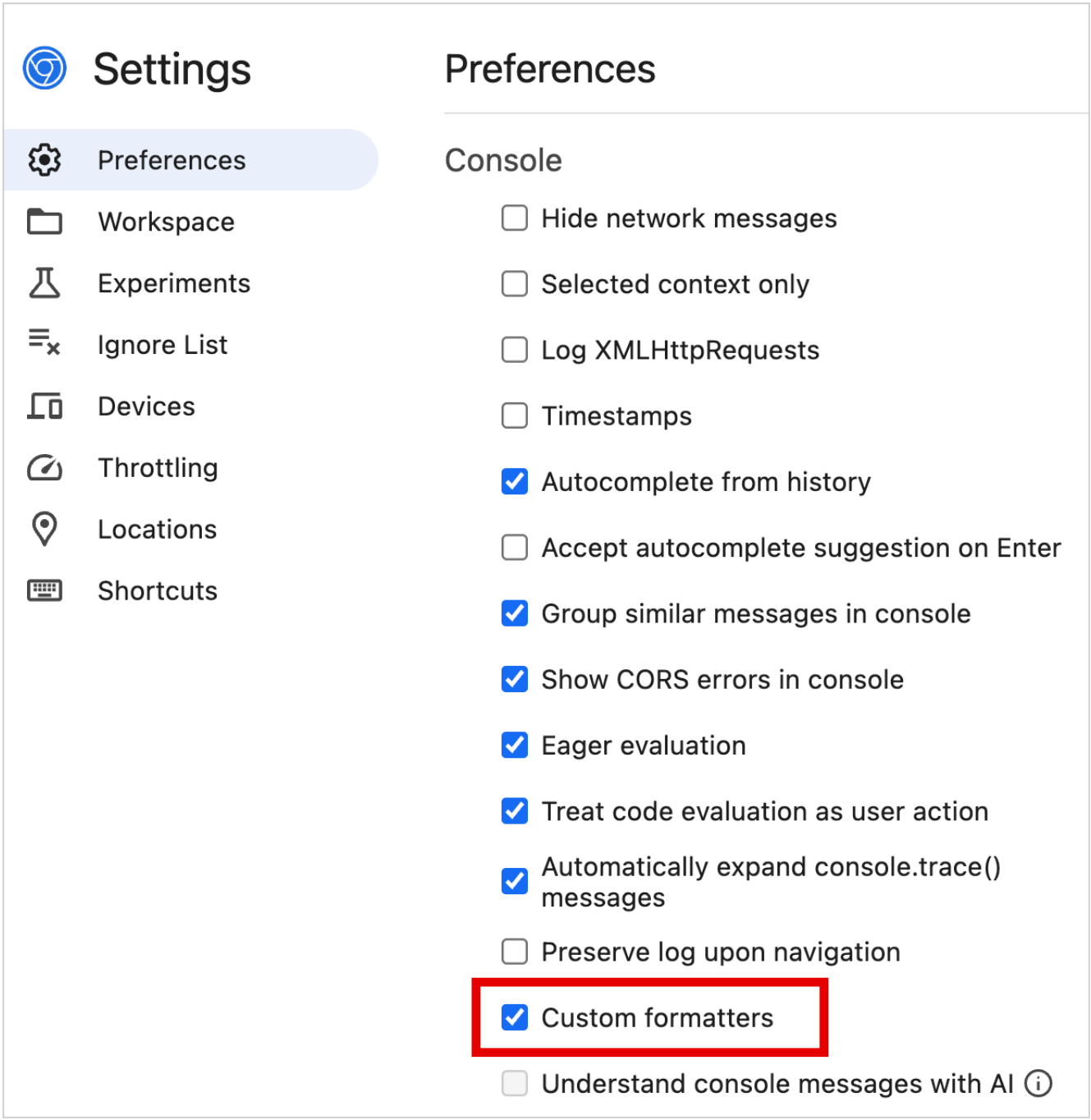
1. Add the following compiler option to the `wasmJs {}` compiler options:

```
// build.gradle.kts
kotlin {
    wasmJs {
        // ...

        compilerOptions {
            freeCompilerArgs.add("-Xwasm-debugger-custom-formatters")
        }
    }
}
```

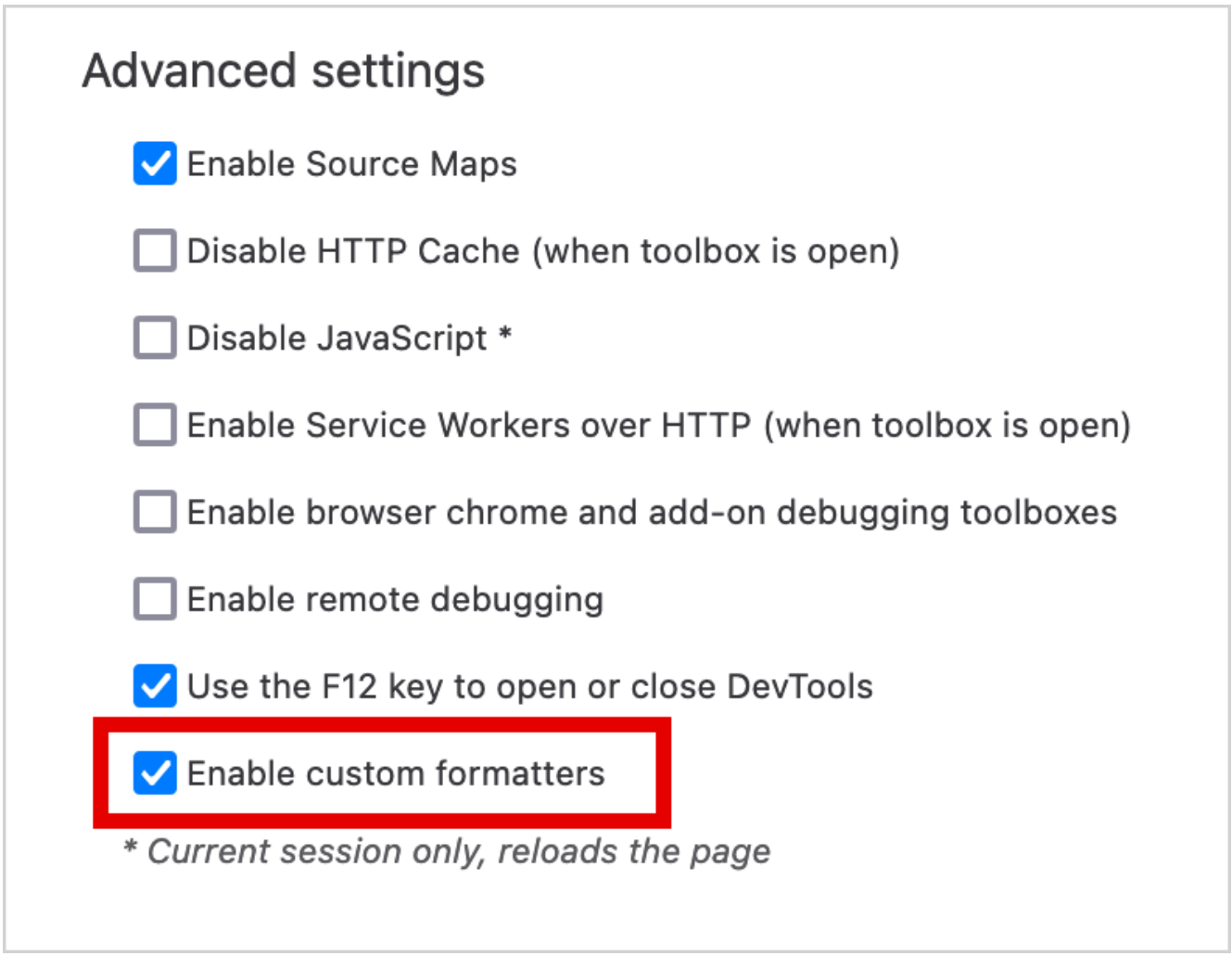
2. Enable custom formatters in your browser:

   - In Chrome DevTools, it's available via **Settings | Preferences | Console**:



Enable custom formatters in Chrome

- In Firefox DevTools, it's available via **Settings | Advanced settings**:



Enable custom formatters in Firefox

Reduced size of Kotlin/Wasm binaries

The size of your Wasm binaries produced by production builds will be reduced by up to 30%, and you may see some performance improvements. This is because the `--closed-world`, `--type-ssa`, and `--type-merging` Binaryen options are now considered safe to use for all Kotlin/Wasm projects and are enabled by default.

Improved JavaScript array interoperability in Kotlin/Wasm

While Kotlin/Wasm's standard library provides the `JsArray<T>` type for JavaScript arrays, there was no direct method to transform `JsArray<T>` into Kotlin's native `Array` or `List` types.

This gap required creating custom functions for array transformations, complicating interoperability between Kotlin and JavaScript code.

This release introduces an adapter function that automatically converts `JsArray<T>` to `Array<T>` and vice versa, simplifying array operations.

Here's an example of conversion between generic types: Kotlin `List<T>` and `Array<T>` to JavaScript `JsArray<T>`.

```
val list: List<JsString> =
    listOf("Kotlin", "Wasm").map { it.toJsString() }

// Uses .toJsArray() to convert List or Array to JsArray
val jsArray: JsArray<JsString> = list.toJsArray()

// Uses .toArray() and .toList() to convert it back to Kotlin types
val kotlinArray: Array<JsString> = jsArray.toArray()
val kotlinList: List<JsString> = jsArray.toList()
```

Similar methods are available for converting typed arrays to their Kotlin equivalents (for example, `IntArray` and `Int32Array`). For detailed information and implementation, see the `kotlinx-browser` repository.

Here's an example of conversion between typed arrays: Kotlin `IntArray` to JavaScript `Int32Array`.

```
import org.khronos.webgl.*

    // ...

    val intArray: IntArray = intArrayOf(1, 2, 3)

    // Uses .toInt32Array() to convert Kotlin IntArray to JavaScript Int32Array
    val jsInt32Array: Int32Array = intArray.toInt32Array()

    // Uses toIntArray() to convert JavaScript Int32Array back to Kotlin IntArray
    val kotlinIntArray: IntArray = jsInt32Array.toIntArray()
```

Support for accessing JavaScript exception details in Kotlin/Wasm

Previously, when a JavaScript exception occurred in Kotlin/Wasm, the `JsException` type provided only a generic message without details from the original JavaScript error.

Starting from Kotlin 2.1.0, you can configure `JsException` to include the original error message and stack trace by enabling a specific compiler option. This provides more context to help diagnose issues originating from JavaScript.

This behavior depends on the `WebAssembly.JSTag` API, which is available only in certain browsers:

- **Chrome**: Supported from version 115
- **Firefox**: Supported from version 129
- **Safari**: Not yet supported

To enable this feature, which is disabled by default, add the following compiler option to your `build.gradle.kts` file:

```
// build.gradle.kts
kotlin {
    wasmJs {
        compilerOptions {
            freeCompilerArgs.add("-Xwasm-attach-js-exception")
        }
    }
}
```

Here's an example demonstrating the new behavior:

```
external object JSON {
    fun <T: JsAny> parse(json: String): T
}

fun main() {
    try {
        JSON.parse("an invalid JSON")
    } catch (e: JsException) {
        println("Thrown value is: ${e.thrownValue}")
        // SyntaxError: Unexpected token 'a', "an invalid JSON" is not valid JSON

        println("Message: ${e.message}")
        // Message: Unexpected token 'a', "an invalid JSON" is not valid JSON

        println("Stacktrace:")
        // Stacktrace:

        // Prints the full JavaScript stack trace
        e.printStackTrace()
    }
}
```

With the `-Xwasm-attach-js-exception` option enabled, `JsException` provides specific details from the JavaScript error. Without the option, `JsException` includes only a generic message stating that an exception was thrown while running JavaScript code.

Deprecation of default exports

As part of the migration to named exports, an error was previously printed to the console when a default import was used for Kotlin/Wasm exports in JavaScript.

In 2.1.0, default imports have been completely removed to fully support named exports.

When coding in JavaScript for the Kotlin/Wasm target, you now need to use the corresponding named imports instead of default imports.

This change marks the last phase of a deprecation cycle to migrate to named exports:

In version 2.0.0: A warning message was printed to the console, explaining that exporting entities via default exports is deprecated.

In version 2.0.20: An error occurred, requesting the use of the corresponding named import.

In version 2.1.0: The use of default imports has been completely removed.

Subproject-specific Node.js settings

You can configure Node.js settings for your project by defining properties of the `NodeJsRootPlugin` class for `rootProject`. In 2.1.0, you can configure these settings for each subproject using a new class, `NodeJsPlugin`. Here's an example demonstrating how to set a specific Node.js version for a subproject:

```
// build.gradle.kts
project.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugin> {
    project.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsEnvSpec>().version = "22.0.0"
}
```

To use the new class for the entire project, add the same code in the `allprojects {}` block:

```
// build.gradle.kts
allprojects {
    project.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugin> {
        project.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsEnvSpec>().version = "your
Node.js version"
    }
}
```

You can also use Gradle convention plugins to apply the settings to a particular set of subprojects.

Kotlin/JS

Support for non-identifier characters in properties

Kotlin/JS previously did not allow using names for test methods with spaces enclosed in backticks.

Similarly, it was not possible to access JavaScript object properties that contained characters not permitted in Kotlin identifiers, such as hyphens or spaces:

```
external interface Headers {
    var accept: String?

    // Invalid Kotlin identifier due to hyphen
    var `content-length`: String?
}

val headers: Headers = TODO("value provided by a JS library")
val accept = headers.accept
// Causes error due to the hyphen in property name
val length = headers.`content-length`
```

This behavior differed from JavaScript and TypeScript, which allow such properties to be accessed using non-identifier characters.

Starting from Kotlin 2.1.0, this feature is enabled by default. Kotlin/JS now allows you to use the backticks (``) and the `@JsName` annotation to interact with JavaScript properties containing non-identifier characters and to use names for test methods.

Additionally, you can use the `@JsName` and `@JsQualifier` annotations to map Kotlin property names to JavaScript equivalents:

```
object Bar {
    val `property example`: String = "bar"
}

@JsQualifier("fooNamespace")
external object Foo {
    val `property example`: String
}

@JsExport
object Baz {
    val `property example`: String = "bar"
}

fun main() {
    // In JavaScript, this is compiled into Bar.property_example_HASH
    println(Bar.`property example`)
    // In JavaScript, this is compiled into fooNamespace["property example"]
    println(Foo.`property example`)
    // In JavaScript, this is compiled into Baz["property example"]
    println(Baz.`property example`)
}
```

Support for generating ES2015 arrow functions

In Kotlin 2.1.0, Kotlin/JS introduces support for generating ES2015 arrow functions, such as `(a, b) => expression`, instead of anonymous functions.

Using arrow functions can reduce the bundle size of your project, especially when using the experimental `-Xir-generate-inline-anonymous-functions` mode. This also makes the generated code more aligned with modern JS.

This feature is enabled by default when targeting ES2015. Alternatively, you can enable it by using the `-Xes-arrow-functions` command line argument.

Learn more about ES2015 (ECMAScript 2015, ES6) in the official documentation.

Gradle improvements

Kotlin 2.1.0 is fully compatible with Gradle 7.6.3 through 8.6. Gradle versions 8.7 to 8.10 are also supported, with only one exception. If you use the Kotlin Multiplatform Gradle plugin, you may see deprecation warnings in your multiplatform projects calling the `withJava()` function in the JVM target. We plan to fix this issue as soon as possible.

For more information, see the related issue in YouTrack.

You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

Minimum supported AGP version bumped to 7.3.1

Starting with Kotlin 2.1.0, the minimum supported Android Gradle plugin version is 7.3.1.

Minimum supported Gradle version bumped to 7.6.3

Starting with Kotlin 2.1.0, the minimum supported Gradle version is 7.6.3.

New API for Kotlin Gradle plugin extensions

Kotlin 2.1.0 introduces a new API to make it easier to create your own plugins for configuring the Kotlin Gradle plugin. This change deprecates the `KotlinTopLevelExtension` and `KotlinTopLevelExtensionConfig` interfaces and introduces the following interfaces for plugin authors:

Name	Description
`KotlinBaseExtension`	A plugin DSL extension type for configuring common Kotlin JVM, Android, and Multiplatform plugin options for the entire project: • `org.jetbrains.kotlin.jvm` • `org.jetbrains.kotlin.android` • `org.jetbrains.kotlin.multiplatform`
`KotlinJvmExtension`	A plugin DSL extension type for configuring Kotlin **JVM** plugin options for the entire project.
`KotlinAndroidExtension`	A plugin DSL extension type for configuring Kotlin **Android** plugin options for the entire project.

For example, if you want to configure compiler options for both JVM and Android projects, use `KotlinBaseExtension`:

```
configure<KotlinBaseExtension> {
    if (this is HasConfigurableKotlinCompilerOptions<*>) {
        with(compilerOptions) {
            if (this is KotlinJvmCompilerOptions) {
                jvmTarget.set(JvmTarget.JVM_17)
            }
        }
    }
}
```

This configures the JVM target to 17 for both JVM and Android projects.

To configure compiler options specifically for JVM projects, use `KotlinJvmExtension`:

```
configure<KotlinJvmExtension> {
    compilerOptions {
        jvmTarget.set(JvmTarget.JVM_17)
    }

    target.mavenPublication {
        groupId = "com.example"
        artifactId = "example-project"
        version = "1.0-SNAPSHOT"
    }
}
```

This example similarly configures the JVM target to 17 for JVM projects. It also configures a Maven publication for the project so that its output is published to a Maven repository.

You can use the `KotlinAndroidExtension` in exactly the same way.

Compiler symbols hidden from the Kotlin Gradle plugin API

Previously, KGP included `org.jetbrains.kotlin:kotlin-compiler-embeddable` in its runtime dependencies, making internal compiler symbols available in the build script classpath. These symbols were intended for internal use only.

Starting with Kotlin 2.1.0, KGP bundles a subset of `org.jetbrains.kotlin:kotlin-compiler-embeddable` class files in its JAR file and progressively removes them. This change aims to prevent compatibility issues and simplify KGP maintenance.

If other parts of your build logic, such as plugins like `kotlinter`, depend on a different version of `org.jetbrains.kotlin:kotlin-compiler-embeddable` than the one bundled with KGP, it can lead to conflicts and runtime exceptions.

To prevent such issues, KGP now shows a warning if `org.jetbrains.kotlin:kotlin-compiler-embeddable` is present in the build classpath alongside KGP.

As a long-term solution, if you're a plugin author using `org.jetbrains.kotlin:kotlin-compiler-embeddable` classes, we recommend running them in an isolated classloader. For example, you can achieve it using the Gradle Workers API with classloader or process isolation.

Using the Gradle Workers API

This example demonstrates how to safely use the Kotlin compiler in a project producing a Gradle plugin. First, add a compile-only dependency in your build script. This makes the symbol available at compile time only:

```
// build.gradle.kts
dependencies {
    compileOnly("org.jetbrains.kotlin:kotlin-compiler-embeddable:2.4.20")
}
```

Next, define a Gradle work action to print the Kotlin compiler version:

```
import org.gradle.workers.WorkAction
import org.gradle.workers.WorkParameters
import org.jetbrains.kotlin.config.KotlinCompilerVersion
abstract class ActionUsingKotlinCompiler : WorkAction<WorkParameters.None> {
    override fun execute() {
        println("Kotlin compiler version: ${KotlinCompilerVersion.getVersion()}")
    }
}
```

Now create a task that submits this action to the worker executor using classloader isolation:

```
import org.gradle.api.DefaultTask
import org.gradle.api.file.ConfigurableFileCollection
import org.gradle.api.tasks.Classpath
import org.gradle.api.tasks.TaskAction
import org.gradle.workers.WorkerExecutor
import javax.inject.Inject
abstract class TaskUsingKotlinCompiler: DefaultTask() {
    @get:Inject
    abstract val executor: WorkerExecutor

    @get:Classpath
    abstract val kotlinCompiler: ConfigurableFileCollection

    @TaskAction
    fun compile() {
        val workQueue = executor.classLoaderIsolation {
            classpath.from(kotlinCompiler)
        }
        workQueue.submit(ActionUsingKotlinCompiler::class.java) {}
    }
}
```

Finally, configure the Kotlin compiler classpath in your Gradle plugin:

```
import org.gradle.api.Plugin
import org.gradle.api.Project
abstract class MyPlugin: Plugin<Project> {
    override fun apply(target: Project) {
        val myDependencyScope = target.configurations.create("myDependencyScope")
        target.dependencies.add(myDependencyScope.name,
"$KOTLIN_COMPILER_EMBEDDABLE:$KOTLIN_COMPILER_VERSION")
        val myResolvableConfiguration = target.configurations.create("myResolvable") {
            extendsFrom(myDependencyScope)
        }
        target.tasks.register("myTask", TaskUsingKotlinCompiler::class.java) {
            kotlinCompiler.from(myResolvableConfiguration)
        }
    }

    companion object {
        const val KOTLIN_COMPILER_EMBEDDABLE = "org.jetbrains.kotlin:kotlin-compiler-embeddable"
        const val KOTLIN_COMPILER_VERSION = "2.4.20"
    }
}
```

Compose compiler updates

Support for multiple stability configuration files

The Compose compiler can interpret multiple stability configuration files, but the `stabilityConfigurationFile` option of the Compose Compiler Gradle plugin previously allowed for only a single file to be specified. In Kotlin 2.1.0, this functionality was reworked to allow you to use several stability configuration files for a single module:

- The `stabilityConfigurationFile` option is deprecated.
- There is a new option, `stabilityConfigurationFiles`, with the type `ListProperty<RegularFile>`.

Here's how to pass several files to the Compose compiler using the new option:

```
// build.gradle.kt
composeCompiler {
    stabilityConfigurationFiles.addAll(
        project.layout.projectDirectory.file("configuration-file1.conf"),
        project.layout.projectDirectory.file("configuration-file2.conf"),
    )
}
```

Pausable composition

Pausable composition is a new Experimental feature that changes how the compiler generates skippable functions. With this feature enabled, composition can be suspended on skipping points during runtime, allowing long-running composition processes to be split across multiple frames. Pausable composition is used in lazy lists and other performance-intensive components for prefetching content that might cause frames to drop when executed in a blocking manner.

To try out pausable composition, add the following feature flag in the Gradle configuration for the Compose compiler:

```
// build.gradle.kts
composeCompiler {
    featureFlags = setOf(
        ComposeFeatureFlag.PausableComposition
    )
}
```

Runtime support for this feature was added in the 1.8.0-alpha02 version of `androidx.compose.runtime`. The feature flag has no effect when used with older runtime versions.

Changes to open and overridden @Composable functions

Virtual (open, abstract, and overridden) `@Composable` functions can no longer be restartable. The codegen for restartable groups was generating calls that did not work correctly with inheritance, resulting in runtime crashes.

This means that virtual functions won't be restarted or skipped: whenever their state is invalidated, runtime will recompose their parent composable instead. If your code is sensitive to recompositions, you may notice changes in runtime behavior.

Performance improvements

The Compose compiler used to create a full copy of module's IR to transform `@Composable` types. Apart from increased memory consumption when copying elements that were not related to Compose, this behavior was also breaking downstream compiler plugins in certain edge cases.

This copy operation was removed, resulting in potentially faster compilation times.

Standard library

Changes to the deprecation severity of standard library APIs

In Kotlin 2.1.0, we are raising the deprecation severity level of several standard library APIs from warning to error. If your code relies on these APIs, you need to update it to ensure compatibility. The most notable changes include:

- **Locale-sensitive case conversion functions for `Char` and `String` are deprecated:** Functions like `Char.toLowerCase()`, `Char.toUpperCase()`, `String.toUpperCase()`, and `String.toLowerCase()` are now deprecated, and using them results in an error. Replace them with locale-agnostic function alternatives or other case conversion mechanisms. If you want to continue using the default locale, replace calls like `String.toLowerCase()` with `String.lowercase(Locale.getDefault())`, explicitly specifying the locale. For a locale-agnostic conversion, replace them with `String.lowercase()`, which uses the invariant locale by default.
- **Kotlin/Native freezing API is deprecated:** Using the freezing-related declarations previously marked with the `@FreezingIsDeprecated` annotation now results in an error. This change reflects the transition from the legacy memory manager in Kotlin/Native, which required freezing objects to share them between threads. To learn how to migrate from freezing-related APIs in the new memory model, see the Kotlin/Native migration guide. For more information, see the announcement about the deprecation of freezing.
- **`appendln()` is deprecated in favor of `appendLine()`:** The `StringBuilder.appendln()` and `Appendable.appendln()` functions are now deprecated, and using them results in an error. To replace them, use the `StringBuilder.appendLine()` or `Appendable.appendLine()` functions instead. The `appendln()` function is deprecated because, on Kotlin/JVM, it uses the `line.separator` system property, which has a different default value on each OS. On Kotlin/JVM, this property defaults to `\r\n` (CR LF) on Windows and `\n` (LF) on other systems. On the other hand, the `appendLine()` function consistently uses `\n` (LF) as the line separator, ensuring consistent behavior across platforms.

For a complete list of affected APIs in this release, see the KT-71628 YouTrack issue.

Stable file tree traversal extensions for java.nio.file.Path

Kotlin 1.7.20 introduced Experimental extension functions for the `java.nio.file.Path` class, which allows you to walk through a file tree. In Kotlin 2.1.0, the following file tree traversal extensions are now Stable:

- `walk()` lazily traverses the file tree rooted at the specified path.
- `fileVisitor()` makes it possible to create a `FileVisitor` separately. `FileVisitor` specifies the actions to be performed on directories and files during traversal.
- `visitFileTree(fileVisitor: FileVisitor, ...)` traverses through a file tree, invoking the specified `FileVisitor` on each encountered entry, and it uses the `java.nio.file.Files.walkFileTree()` function under the hood.
- `visitFileTree(..., builderAction: FileVisitorBuilder.() -> Unit)` creates a `FileVisitor` with the provided `builderAction` and calls the `visitFileTree(fileVisitor, ...)` function.
- `sealed interface FileVisitorBuilder` allows you to define a custom `FileVisitor` implementation.
- `enum class PathWalkOption` provides traversal options for the `Path.walk()` function.

The examples below demonstrate how to use these file traversal APIs to create custom `FileVisitor` behaviors, which allows you to define specific actions for visiting files and directories.

For instance, you can explicitly create a `FileVisitor` and use it later:

```
val cleanVisitor = fileVisitor {
    onPreVisitDirectory { directory, attributes ->
        // Placeholder: Add logic on visiting directories
        FileVisitResult.CONTINUE
    }

    onVisitFile { file, attributes ->
        // Placeholder: Add logic on visiting files
        FileVisitResult.CONTINUE
    }
}

// Placeholder: Add logic here for general setup before traversal
projectDirectory.visitFileTree(cleanVisitor)
```

You can also create a `FileVisitor` with the `builderAction` and use it immediately for the traversal:

```
projectDirectory.visitFileTree {
    // Defines the builderAction:
    onPreVisitDirectory { directory, attributes ->
        // Some logic on visiting directories
        FileVisitResult.CONTINUE
    }

    onVisitFile { file, attributes ->
        // Some logic on visiting files
        FileVisitResult.CONTINUE
    }
}
```

Additionally, you can traverse a file tree rooted at the specified path with the `walk()` function:

```
fun traverseFileTree() {
    val cleanVisitor = fileVisitor {
        onPreVisitDirectory { directory, _ ->
            if (directory.name == "build") {
                directory.toFile().deleteRecursively()
                FileVisitResult.SKIP_SUBTREE
            } else {
                FileVisitResult.CONTINUE
            }
        }

        // Deletes files with the .class extension
        onVisitFile { file, _ ->
            if (file.extension == "class") {
                file.deleteExisting()
            }
            FileVisitResult.CONTINUE
        }
    }

    // Sets up the root directory and files
    val rootDirectory = createTempDirectory("Project")

    // Creates the src directory with A.kt and A.class files
    rootDirectory.resolve("src").let { srcDirectory ->
        srcDirectory.createDirectory()
        srcDirectory.resolve("A.kt").createFile()
        srcDirectory.resolve("A.class").createFile()
    }

    // Creates the build directory with a Project.jar file
    rootDirectory.resolve("build").let { buildDirectory ->
        buildDirectory.createDirectory()
        buildDirectory.resolve("Project.jar").createFile()
    }

    // Uses the walk() function:
    val directoryStructure = rootDirectory.walk(PathWalkOption.INCLUDE_DIRECTORIES)
        .map { it.relativeTo(rootDirectory).toString() }
        .toList().sorted()
    println(directoryStructure)
    // "[, build, build/Project.jar, src, src/A.class, src/A.kt]"

    // Traverses the file tree with cleanVisitor, applying the
rootDirectory.visitFileTree(cleanVisitor) cleanup rules
    val directoryStructureAfterClean = rootDirectory.walk(PathWalkOption.INCLUDE_DIRECTORIES)
        .map { it.relativeTo(rootDirectory).toString() }
        .toList().sorted()
    println(directoryStructureAfterClean)
    // "[, src, src/A.kt]"
}
```

Documentation updates

The Kotlin documentation has received some notable changes:

Language concepts

- Improved Null safety page – Learn how to handle `null` values safely in your code.

- Improved Objects declarations and expressions page – Learn how to define a class and create an instance in a single step.
- Improved When expressions and statements section – Learn about the `when` conditional and how you can use it.
- Updated Kotlin roadmap, Kotlin evolution principles, and Kotlin language features and proposals pages – Learn about Kotlin's plans, ongoing developments, and guiding principles.

Compose compiler

- Compose compiler documentation now located in the Compiler and plugins section – Learn about the Compose compiler, the compiler options, and the steps to migrate.

API references

- New Kotlin Gradle plugins API reference – Explore the API references for the Kotlin Gradle plugin and the Compose compiler Gradle plugin.

Multiplatform development

- New Building a Kotlin library for multiplatform page – Learn how to design your Kotlin libraries for Kotlin Multiplatform.
- New Introduction to Kotlin Multiplatform page – Learn about Kotlin Multiplatform's key concepts, dependencies, libraries, and more.
- New iOS integration section – Learn how to integrate a Kotlin Multiplatform shared module into your iOS app.
- New Kotlin/Native's definition file page – Learn how to create a definition file to consume C and Objective-C libraries.
- Get started with WASI – Learn how to run a simple Kotlin/Wasm application using WASI in various WebAssembly virtual machines.

Tooling

- New Dokka migration guide – Learn how to migrate to Dokka Gradle plugin v2.

Compatibility guide for Kotlin 2.1.0

Kotlin 2.1.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of these changes in the Compatibility guide for Kotlin 2.1.0.

Install Kotlin 2.1.0

Starting from IntelliJ IDEA 2023.3 and Android Studio Iguana (2023.2.1) Canary 15, the Kotlin plugin is distributed as a bundled plugin included in your IDE. This means that you can't install the plugin from JetBrains Marketplace anymore.

To update to the new Kotlin version, change the Kotlin version to 2.1.0 in your build scripts.

Compatibility guide for Kotlin 2.1.x

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 2.0 to Kotlin 2.1.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Remove language versions 1.4 and 1.5

Issue: KT-60521

Component: Core language

Incompatible change type: source

Short summary: Kotlin 2.1 introduces language version 2.1 and removes support for language versions 1.4 and 1.5. Language versions 1.6 and 1.7 are deprecated.

Deprecation cycle:

- 1.6.0: report a warning for language version 1.4
- 1.9.0: report a warning for language version 1.5
- 2.1.0: report a warning for language versions 1.6 and 1.7; raise the warning to an error for language versions 1.4 and 1.5

Change the typeOf() function behavior on Kotlin/Native

Issue: KT-70754

Component: Core language

Incompatible change type: behavioral

Short summary: The behavior of the `typeOf()` function on Kotlin/Native is aligned with Kotlin/JVM to ensure consistency across platforms.

Deprecation cycle:

- 2.1.0: align the `typeOf()` function behavior on Kotlin/Native

Prohibit exposing types through type parameters' bounds

Issue: KT-69653

Component: Core language

Incompatible change type: source

Short summary: Exposing types with lower visibility through type parameter bounds is now prohibited, addressing inconsistencies in type visibility rules. This change ensures that bounds on type parameters follow the same visibility rules as classes, preventing issues like IR validation errors in the JVM.

Deprecation cycle:

- 2.1.0: report a warning for exposing types via type parameter bounds with lower visibility
- 2.2.0: raise the warning to an error

Prohibit inheriting an abstract var property and a val property with the same name

Issue: KT-58659

Component: Core language

Incompatible change type: source

Short summary: If a class inherits an abstract `var` property from an interface and a `val` property with the same name from a superclass, it now triggers a compilation error. This resolves runtime crashes caused by missing setters in such cases.

Deprecation cycle:

- 2.1.0: report a warning (or an error in progressive mode) when a class inherits an abstract `var` property from an interface and a `val` property with the same name from a superclass
- 2.2.0: raise the warning to an error

Report error when accessing uninitialized enum entries

Issue: KT-68451

Component: Core language

Incompatible change type: source

Short summary: The compiler now reports an error when uninitialized enum entries are accessed during enum class or entry initialization. This aligns the behavior with member property initialization rules, preventing runtime exceptions and ensuring consistent logic.

Deprecation cycle:

- 2.1.0: report an error when accessing uninitialized enum entries

Changes in K2 smart cast propagation

Issue: KTLC-34

Component: Core language

Incompatible change type: behavioral

Short summary: The K2 compiler changes its behavior for smart cast propagation by introducing bidirectional propagation of type information for inferred variables, like `val x = y`. Explicitly typed variables, such as `val x: T = y`, no longer propagate type information, ensuring stricter adherence to declared types.

Deprecation cycle:

- 2.1.0: enable the new behavior

Correct the handling of member-extension property overrides in Java subclasses

Issue: KTLC-35

Component: Core language

Incompatible change type: behavioral

Short summary: The getter for member-extension properties overridden by Java subclasses is now hidden in the subclass's scope, aligning its behavior with that of regular Kotlin properties.

Deprecation cycle:

- 2.1.0: enable the new behavior

Correct visibility alignment for getters and setters of var properties overriding a protected val

Issue: KTLC-36

Component: Core language

Incompatible change type: binary

Short summary: The visibility of getters and setters for `var` properties overriding a `protected val` property is now consistent, with both inheriting the visibility of the overridden `val` property.

Deprecation cycle:

- 2.1.0: enforce consistent visibility for both getters and setters in K2; K1 remains unaffected

Raise severity of JSpecify nullability mismatch diagnostics to errors

Issue: KTLC-11

Component: Core language

Incompatible change type: source

Short summary: Nullability mismatches from `org.jspecify.annotations`, such as `@NonNull`, `@Nullable`, and `@NullMarked` are now treated as errors instead of warnings, enforcing stricter type safety for Java interoperability. To adjust the severity of these diagnostics, use the `-Xnullability-annotations` compiler option.

Deprecation cycle:

- 1.6.0: report warnings for potential nullability mismatches
- 1.8.20: expand warnings to specific JSpecify annotations, including: `@Nullable`, `@NullnessUnspecified`, `@NullMarked`, and legacy annotations in `org.jspecify.nullness` (JSpecify 0.2 and earlier)
- 2.0.0: add support for the `@NonNull` annotation
- 2.1.0: change default mode to `strict` for JSpecify annotations, converting warnings into errors; use `-Xnullability-annotations=@org.jspecify.annotations:warning` or `-Xnullability-annotations=@org.jspecify.annotations:ignore` to override the default behavior

Change overload resolution to prioritize extension functions over invoke calls in ambiguous cases

Issue: KTLC-37

Component: Core language

Incompatible change type: behavioral

Short summary: Overload resolution now consistently prioritizes extension functions over invoke calls in ambiguous cases. This resolves inconsistencies in the resolution logic for local functions and properties. The change applies only after recompilation, without affecting precompiled binaries.

Deprecation cycle:

- 2.1.0: change overload resolution to consistently prioritize extension functions over `invoke` calls for extension functions with matching signatures; this change applies only after recompilation and does not affect precompiled binaries

Prohibit returning nullable values from lambdas in SAM constructors of JDK function interfaces

Issue: KTLC-42

Component: Core language

Incompatible change type: source

Short summary: Returning nullable values from lambdas in SAM constructors of JDK function interfaces now triggers a compilation error if the specified type argument is non-nullable. This resolves issues where nullability mismatches could lead to runtime exceptions, ensuring stricter type safety.

Deprecation cycle:

- 2.0.0: report a deprecation warning for nullable return values in SAM constructors of JDK function interfaces
- 2.1.0: enable the new behavior by default

Correct handling of private members conflicting with public members in Kotlin/Native

Issue: KTLC-43

Component: Core language

Incompatible change type: behavioral

Short summary: In Kotlin/Native, private members no longer override or conflict with public members in a superclass, aligning behavior with Kotlin/JVM. This resolves inconsistencies in override resolution and eliminates unexpected behavior caused by separate compilation.

Deprecation cycle:

- 2.1.0: private functions and properties in Kotlin/Native no longer override or affect public members in a superclass, aligning with JVM behavior

Forbid access to private operator functions in public inline functions

Issue: KTLC-71

Component: Core language

Incompatible change type: source

Short summary: Private operator functions such as `getValue()`, `setValue()`, `provideDelegate()`, `hasNext()`, and `next()` can no longer be accessed in public inline functions.

Deprecation cycle:

- 2.0.0: report a deprecation warning for accessing private operator functions in public inline functions
- 2.1.0: raise the warning to an error

Prohibit passing invalid arguments to invariant parameters annotated with @UnsafeVariance

Issue: KTLC-72

Component: Core language

Incompatible change type: source

Short summary: The compiler now ignores `@UnsafeVariance` annotations during type checks, enforcing stricter type safety for invariant type parameters. This prevents invalid calls that rely on `@UnsafeVariance` to bypass expected type checks.

Deprecation cycle:

- 2.1.0: activate the new behavior

Report nullability errors for error-level nullable arguments of warning-level Java types

Issue: KTLC-100

Component: Core language

Incompatible change type: source

Short summary: The compiler now detects nullability mismatches in Java methods where a warning-level nullable type contains type arguments with stricter, error-level nullability. This ensures that previously ignored errors in type arguments are reported correctly.

Deprecation cycle:

- 2.0.0: report a deprecation warning for nullability mismatches in Java methods with stricter type arguments
- 2.1.0: raise the warning to an error

Report implicit usages of inaccessible types

Issue: KTLC-3

Component: Core language

Incompatible change type: source

Short summary: The compiler now reports usages of inaccessible types in function literals and type arguments, preventing compilation and runtime failures caused by incomplete type information.

Deprecation cycle:

- 2.0.0: report warnings for function literals with parameters or receivers of inaccessible non-generic types and for types with inaccessible type argument; report errors for function literals with parameters or receivers of inaccessible generic types and for types with inaccessible generic type arguments in specific scenarios
- 2.1.0: raise warnings to errors for function literals with parameters and receivers of inaccessible non-generic types
- 2.2.0: raise warnings to errors for types with inaccessible type arguments

Standard library

Deprecate locale-sensitive case conversion functions for Char and String

Issue: KT-43023

Component: kotlin-stdlib

Incompatible change type: source

Short summary: Among other Kotlin standard library APIs, locale-sensitive case conversion functions for `Char` and `String`, such as `Char.toUpperCase()` and `String.toLowerCase()`, are deprecated. Replace them with locale-agnostic alternatives like `String.lowercase()` or explicitly specify the locale for locale-sensitive behavior, such as `String.lowercase(Locale.getDefault())`.

For a comprehensive list of deprecated Kotlin standard library APIs in Kotlin 2.1.0, see KT-71628.

Deprecation cycle:

- 1.4.30: introduce locale-agnostic alternatives as experimental API
- 1.5.0: deprecate locale-sensitive case conversion functions with a warning
- 2.1.0: raise the warning to an error

Remove kotlin-stdlib-common JAR artifact

Issue: KT-62159

Component: kotlin-stdlib

Incompatible change type: binary

Short summary: The `kotlin-stdlib-common.jar` artifact, previously used for legacy multiplatform declarations metadata, is deprecated and replaced by `.klib` files as the standard format for common multiplatform declarations metadata. This change does not affect the main `kotlin-stdlib.jar` or `kotlin-stdlib-all.jar` artifacts.

Deprecation cycle:

- 2.1.0: deprecate and remove `kotlin-stdlib-common.jar` artifact

Deprecate appendln() in favor of appendLine()

Issue: KTLC-27

Component: kotlin-stdlib

Incompatible change type: source

Short summary: `StringBuilder.appendln()` is deprecated in favor of `StringBuilder.appendLine()`.

Deprecation cycle:

- 1.4.0: the `appendln()` function is deprecated; report a warning on use
- 2.1.0: raise the warning to an error

Deprecate freezing-related APIs in Kotlin/Native

Issue: KT-69545

Component: kotlin-stdlib

Incompatible change type: source

Short summary: Freezing-related APIs in Kotlin/Native, previously marked with the `@FreezingIsDeprecated` annotation, are now deprecated. This aligns with the introduction of the new memory manager that removes the need for freezing objects for thread sharing. For migration details, see the Kotlin/Native migration guide.

Deprecation cycle:

- 1.7.20: deprecate freezing-related APIs with a warning
- 2.1.0: raise the warning to an error

Change Map.Entry behavior to fail-fast on structural modification

Issue: KTLC-23

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: Accessing a `Map.Entry` key-value pair after its associated map has been structurally modified now throws a `ConcurrentModificationException`.

Deprecation cycle:

- 2.1.0: throw an exception when a map structural modification is detected

Tools

Deprecate KotlinCompilationOutput#resourcesDirProvider

Issue: KT-69255

Component: Gradle

Incompatible change type: source

Short summary: The `KotlinCompilationOutput#resourcesDirProvider` field is deprecated. Use `KotlinSourceSet.resources` in your Gradle build script instead to add additional resource directories.

Deprecation cycle:

- 2.1.0: `KotlinCompilationOutput#resourcesDirProvider` is deprecated

Deprecate registerKotlinJvmCompileTask(taskName, moduleName) function

Issue: KT-69927

Component: Gradle

Incompatible change type: source

Short summary: The `registerKotlinJvmCompileTask(taskName, moduleName)` function is deprecated in favor of the new `registerKotlinJvmCompileTask(taskName, compilerOptions, explicitApiMode)` function, which now accepts `KotlinJvmCompilerOptions`. This allows you to pass a `compilerOptions` instance, typically from an extension or target, with values used as conventions for the task's options.

Deprecation cycle:

- 2.1.0: the `registerKotlinJvmCompileTask(taskName, moduleName)` function is deprecated

Deprecate registerKaptGenerateStubsTask(taskName) function

Issue: KT-70383

Component: Gradle

Incompatible change type: source

Short summary: The `registerKaptGenerateStubsTask(taskName)` function is deprecated. Use the new `registerKaptGenerateStubsTask(compileTask, kaptExtension, explicitApiMode)` function instead. This new version allows you to link values as conventions from the relevant `KotlinJvmCompile` task, ensuring both tasks are using the same set of options.

Deprecation cycle:

- 2.1.0: the `registerKaptGenerateStubsTask(taskName)` function is deprecated

Deprecate KotlinTopLevelExtension and KotlinTopLevelExtensionConfig interfaces

Issue: KT-71602

Component: Gradle

Incompatible change type: behavioral

Short summary: `KotlinTopLevelExtension` and `KotlinTopLevelExtensionConfig` interfaces are deprecated in favor of a new `KotlinTopLevelExtension` interface. This interface merges `KotlinTopLevelExtensionConfig`, `KotlinTopLevelExtension`, and `KotlinProjectExtension` to streamline API hierarchy, and provide official access to the JVM toolchain and compiler properties.

Deprecation cycle:

- 2.1.0: the `KotlinTopLevelExtension` and `KotlinTopLevelExtensionConfig` interfaces are deprecated

Remove kotlin-compiler-embeddable from build runtime dependencies

Issue: KT-61706

Component: Gradle

Incompatible change type: source

Short summary: The `kotlin-compiler-embeddable` dependency is removed from the runtime in Kotlin Gradle Plugin (KGP). Required modules are now included directly in KGP artifacts, with the Kotlin language version limited to 2.0 to support compatibility with Gradle Kotlin runtime in versions below 8.2.

Deprecation cycle:

- 2.1.0: report a warning on using `kotlin-compiler-embeddable`
- 2.2.0: raise the warning to an error

Hide compiler symbols from the Kotlin Gradle Plugin API

Issue: KT-70251

Component: Gradle

Incompatible change type: source

Short summary: Compiler module symbols bundled within the Kotlin Gradle Plugin (KGP), such as `KotlinCompilerVersion`, are hidden from the public API to prevent unintended access in build scripts.

Deprecation cycle:

- 2.1.0: report a warning on accessing these symbols
- 2.2.0: raise the warning to an error

Add support for multiple stability configuration files

Issue: KT-68345

Component: Gradle

Incompatible change type: source

Short summary: The `stabilityConfigurationFile` property in the Compose extension is deprecated in favor of a new `stabilityConfigurationFiles` property, which allows specifying multiple configuration files.

Deprecation cycle:

- 2.1.0: `stabilityConfigurationFile` property is deprecated

Remove deprecated platform plugin IDs

Issue: KT-65565

Component: Gradle

Incompatible change type: source

Short summary: Support for these platform plugin IDs have been removed:

- `kotlin-platform-common`
- `org.jetbrains.kotlin.platform.common`

Deprecation cycle:

- 1.3: the platform plugin IDs are deprecated
- 2.1.0: the platform plugin IDs are no longer supported

What's new in Kotlin 2.0.20

Released: August 22, 2024

The Kotlin 2.0.20 release is out! This version includes performance improvements and bug fixes for Kotlin 2.0.0, where we announced the Kotlin K2 compiler as Stable. Here are some additional highlights from this release:

- The data class copy function to have the same visibility as the constructor
- Static accessors for source sets from the default target hierarchy are now available in multiplatform projects
- Concurrent marking for Kotlin/Native has been made possible in the garbage collector
- The `@ExperimentalWasmDsl` annotation in Kotlin/Wasm has a new location
- Support has been added for Gradle versions 8.6–8.8
- A new option allows sharing JVM artifacts between Gradle projects as class files
- The Compose compiler has been updated
- Support for UUIDs has been added to the common Kotlin standard library

For information about the Kotlin release cycle, see Kotlin release process.

IDE support

The Kotlin plugins that support 2.0.20 are bundled in the latest IntelliJ IDEA and Android Studio. You don't need to update the Kotlin plugin in your IDE. All you need to do is to change the Kotlin version to 2.0.20 in your build scripts.

See Update to a new release for details.

Language

Kotlin 2.0.20 begins to introduce changes to improve consistency in data classes and replace the Experimental context receivers feature.

Data class copy function to have the same visibility as constructor

Currently, if you create a data class using a `private` constructor, the automatically generated `copy()` function doesn't have the same visibility. This can cause problems later in your code. In future Kotlin releases, we will introduce the behavior that the default visibility of the `copy()` function is the same as the constructor. This change will be introduced gradually to help you migrate your code as smoothly as possible.

Our migration plan starts with Kotlin 2.0.20, which issues warnings in your code where the visibility will change in the future. For example:

```
// Triggers a warning in 2.0.20
data class PositiveInteger private constructor(val number: Int) {
    companion object {
        fun create(number: Int): PositiveInteger? = if (number > 0) PositiveInteger(number) else null
    }
}

fun main() {
    val positiveNumber = PositiveInteger.create(42) ?: return
    // Triggers a warning in 2.0.20
    val negativeNumber = positiveNumber.copy(number = -1)
    // Warning: Non-public primary constructor is exposed via the generated 'copy()' method of the
'data' class.
    // The generated 'copy()' will change its visibility in future releases.
}
```

For the latest information about our migration plan, see the corresponding issue in YouTrack.

To give you more control over this behavior, in Kotlin 2.0.20 we've introduced two annotations:

- `@ConsistentCopyVisibility` to opt in to the behavior now before we make it the default in a later release.
- `@ExposedCopyVisibility` to opt out of the behavior and suppress warnings at the declaration site. Note that even with this annotation, the compiler still reports warnings when the `copy()` function is called.

If you want to opt in to the new behavior already in 2.0.20 for a whole module rather than in individual classes, you can use the `-Xconsistent-data-class-copy-visibility` compiler option. This option has the same effect as adding the `@ConsistentCopyVisibility` annotation to all data classes in a module.

Phased replacement of context receivers with context parameters

In Kotlin 1.6.20, we introduced context receivers as an Experimental feature. After listening to community feedback, we've decided not to continue with this approach and will take a different direction.

In future Kotlin releases, context receivers will be replaced by context parameters. Context parameters are still in the design phase, and you can find the proposal in the KEEP.

Since the implementation of context parameters requires significant changes to the compiler, we've decided not to support context receivers and context parameters simultaneously. This decision greatly simplifies the implementation and minimizes the risk of unstable behavior.

We understand that context receivers are already being used by a large number of developers. Therefore, we will begin gradually removing support for context receivers. Our migration plan starts with Kotlin 2.0.20, where warnings are issued in your code when context receivers are used with the `-Xcontext-receivers` compiler option. For example:

```
class MyContext

context(MyContext)
// Warning: Experimental context receivers are deprecated and will be superseded by context
parameters.
// Please don't use context receivers. You can either pass parameters explicitly or use members with
extensions.
fun someFunction() {
}
```

This warning will become an error in future Kotlin releases.

If you use context receivers in your code, we recommend that you migrate your code to use either of the following:

- Explicit parameters.

Before	After

`context(ContextReceiverType)`
`fun someFunction() {`
`  contextReceiverMember()`
`}`
	`fun someFunction(explicitContext: ContextReceiverType) {`
	`  explicitContext.contextReceiverMember()`
	`}`

- Extension member functions (if possible).

Before	After

`context(ContextReceiverType)`
`fun contextReceiverMember() = TODO()`
 	
`context(ContextReceiverType)`
`fun someFunction() {`
`  contextReceiverMember()`
`}`
	`class ContextReceiverType {`
	`  fun contextReceiverMember() = TODO()`
	`}`
	
	`fun ContextReceiverType.someFunction() {`
	`  contextReceiverMember()`
	`}`

Alternatively, you can wait until the Kotlin release where context parameters are supported in the compiler. Note that context parameters will initially be introduced as an Experimental feature.

Kotlin Multiplatform

Kotlin 2.0.20 brings improvements to source set management in multiplatform projects as well as deprecates compatibility with some Gradle Java plugins due to recent changes in Gradle.

Static accessors for source sets from the default target hierarchy

Since Kotlin 1.9.20, the default hierarchy template is automatically applied to all Kotlin Multiplatform projects. And for all of the source sets from the default hierarchy template, the Kotlin Gradle plugin provided type-safe accessors. That way, you could finally access source sets for all the specified targets without having to use `by getting` or `by creating` constructs.

Kotlin 2.0.20 aims to improve your IDE experience even further. It now provides static accessors in the `sourceSets {}` block for all the source sets from the default hierarchy template. We believe this change will make accessing source sets by name easier and more predictable.

Each such source set now has a detailed KDoc comment with a sample and a diagnostic message with a warning in case you try to access the source set without declaring the corresponding target first:

```
kotlin {
    jvm()
    linuxX64()
    linuxArm64()
    mingwX64()

    sourceSets {
        commonMain.languageSettings {
            progressiveMode = true
        }


        jvmMain { }
        linuxX64Main { }
        linuxArm64Main { }
        // Warning: accessing source set without registering the target
        iosX64Main { }
    }
}
```

```
public abstract val NamedDomainObjectContainer<KotlinSourceSet>.iosX64Main:
NamedDomainObjectProvider<KotlinSourceSet>
```

Static accessor for the main Kotlin Source Set of iosX64 target. Declare iosX64 target to access this source set. If iosX64 target wasn't declared, accessing this source set will cause a runtime error during configuration time.

Sample:

```
kotlin {
   iosX64() // Target is declared, iosX64Main source set is created

   sourceSets {
     iosX64Main.dependencies {
         // Add iosX64Main dependencies here
     }
   }
}
```

Since: 2.0.20

Accessing the source sets by name

Learn more about the hierarchical project structure in Kotlin Multiplatform.

Deprecated compatibility with Kotlin Multiplatform Gradle plugin and Gradle Java plugins

In Kotlin 2.0.20, we introduce a deprecation warning when you apply the Kotlin Multiplatform Gradle plugin and any of the following Gradle Java plugins to the same project: Java, Java Library, and Application. The warning also appears when another Gradle plugin in your multiplatform project applies a Gradle Java plugin. For example, the Spring Boot Gradle Plugin automatically applies the Application plugin.

We've added this deprecation warning due to fundamental compatibility issues between Kotlin Multiplatform's project model and Gradle's Java ecosystem plugins. Gradle's Java ecosystem plugins currently don't take into account that other plugins may:

- Also publish or compile for the JVM target in a different way than the Java ecosystem plugins.
- Have two different JVM targets in the same project, such as JVM and Android.
- Have a complex multiplatform project structure with potentially multiple non-JVM targets.

Unfortunately, Gradle doesn't currently provide any API to address these issues.

We previously used some workarounds in Kotlin Multiplatform to help with the integration of Java ecosystem plugins. However, these workarounds never truly solved the compatibility issues, and since the release of Gradle 8.8, these workarounds are no longer possible. For more information, see our YouTrack issue.

While we don't yet know exactly how to resolve this compatibility problem, we are committed to continuing support for some form of Java source compilation in your Kotlin Multiplatform projects. At a minimum, we will support the compilation of Java sources and using Gradle's `java-base` plugin within your multiplatform projects.

In the meantime, if you see this deprecation warning in your multiplatform project, we recommend that you:

1. Determine whether you actually need the Gradle Java plugin in your project. If not, consider removing it.
2. Check if the Gradle Java plugin is only used for a single task. If so, you might be able to remove the plugin without much effort. For example, if the task uses a Gradle Java plugin to create a Javadoc JAR file, you can define the Javadoc task manually instead.

Otherwise, if you want to use both the Kotlin Multiplatform Gradle plugin and these Gradle plugins for Java in your multiplatform project, we recommend that you:

1. Create a separate subproject in your multiplatform project.
2. In the separate subproject, apply the Gradle plugin for Java.
3. In the separate subproject, add a dependency on your parent multiplatform project.

> The separate subproject must **not** be a multiplatform project, and you must only use it to set up a dependency on your multiplatform project.

For example, you have a multiplatform project called `my-main-project` and you want to use the Application Gradle plugin to run a JVM application.

Once you've created a subproject, let's call it `subproject-A`, your parent project structure should look like this:

```
.
├── build.gradle.kts
├── settings.gradle
├── subproject-A
    └── build.gradle.kts
    └── src
        └── Main.java
```

In your subproject's `build.gradle.kts` file, apply the Application plugin in the `plugins {}` block:

Kotlin

```
plugins {
    id("application")
}
```

Groovy

```
plugins {
    id('application')
}
```

In your subproject's `build.gradle.kts` file, add a dependency on your parent multiplatform project:

Kotlin

```
dependencies {
    implementation(project(":my-main-project")) // The name of your parent multiplatform project
}
```

Groovy

```
dependencies {
    implementation project(':my-main-project') // The name of your parent multiplatform project
}
```

Your parent project is now set up to work with both plugins.

Kotlin/Native

Kotlin/Native receives improvements in the garbage collector and for calling Kotlin suspending functions from Swift/Objective-C.

Concurrent marking in garbage collector

In Kotlin 2.0.20, the JetBrains team takes another step toward improving Kotlin/Native runtime performance. We've added experimental support for concurrent marking in the garbage collector (GC).

By default, application threads must be paused when GC is marking objects in the heap. This greatly affects the duration of the GC pause time, which is important for the performance of latency-critical applications, such as UI applications built with Compose Multiplatform.

Now, the marking phase of the garbage collection can be run simultaneously with application threads. This should significantly shorten the GC pause time and help improve app responsiveness.

How to enable

The feature is currently Experimental. To enable it, set the following option in your `gradle.properties` file:

```
kotlin.native.binary.gc=cms
```

Please report any problems to our issue tracker YouTrack.

Support for bitcode embedding removed

Starting with Kotlin 2.0.20, the Kotlin/Native compiler no longer supports bitcode embedding. Bitcode embedding was deprecated in Xcode 14 and removed in Xcode 15 for all Apple targets.

Now, the `embedBitcode` parameter for the framework configuration, as well as the `-Xembed-bitcode` and `-Xembed-bitcode-marker` command line arguments are deprecated.

If you still use earlier versions of Xcode but want to upgrade to Kotlin 2.0.20, disable bitcode embedding in your Xcode projects.

Changes to GC performance monitoring with signposts

Kotlin 2.0.0 made it possible to monitor the performance of the Kotlin/Native garbage collector (GC) through Xcode Instruments. Instruments include the signposts tool, which can show GC pauses as events. This comes in handy when checking GC-related freezes in your iOS apps.

The feature was enabled by default, but unfortunately, it sometimes led to crashes when the application was run simultaneously with Xcode Instruments. Starting with Kotlin 2.0.20, it requires an explicit opt-in with the following compiler option:

```
-Xbinary=enableSafepointSignposts=true
```

Learn more about GC performance analysis in the documentation.

Ability to call Kotlin suspending functions from Swift/Objective-C on non-main threads

Previously, Kotlin/Native had a default restriction, limiting the ability to call Kotlin suspending functions from Swift and Objective-C to only the main thread. Kotlin 2.0.20 lifts that limitation, allowing you to run Kotlin `suspend` functions from Swift/Objective-C on any thread.

If you've previously switched the default behavior for non-main threads with the `kotlin.native.binary.objcExportSuspendFunctionLaunchThreadRestriction=none` binary option, you can now remove it from your `gradle.properties` file.

Kotlin/Wasm

In Kotlin 2.0.20, Kotlin/Wasm continues the migration towards named exports and relocates the `@ExperimentalWasmDsl` annotation.

Error in default export usage

As part of the migration towards named exports, a warning message was previously printed to the console when using a default import for Kotlin/Wasm exports in JavaScript.

To fully support named exports, this warning has now been upgraded to an error. If you use a default import, you encounter the following error message:

```
Do not use default import. Use the corresponding named import instead.
```

This change is part of a deprecation cycle to migrate towards named exports. Here's what you can expect during each phase:

- **In version 2.0.0**: A warning message is printed to the console, explaining that exporting entities via default exports is deprecated.
- **In version 2.0.20**: An error occurs, requesting the use of the corresponding named import.
- **In version 2.1.0**: The use of default imports is completely removed.

New location of ExperimentalWasmDsl annotation

Previously, the `@ExperimentalWasmDsl` annotation for WebAssembly (Wasm) features was placed in this location within the Kotlin Gradle plugin:

```
org.jetbrains.kotlin.gradle.targets.js.dsl.ExperimentalWasmDsl
```

In 2.0.20, the `@ExperimentalWasmDsl` annotation has been relocated to:

```
org.jetbrains.kotlin.gradle.ExperimentalWasmDsl
```

The previous location is now deprecated and might lead to build failures with unresolved references.

To reflect the new location of the `@ExperimentalWasmDsl` annotation, update the import statement in your Gradle build scripts. Use an explicit import for the new `@ExperimentalWasmDsl` location:

```
import org.jetbrains.kotlin.gradle.ExperimentalWasmDsl
```

Alternatively, remove this star import statement from the old package:

```
import org.jetbrains.kotlin.gradle.targets.js.dsl.*
```

Kotlin/JS

Kotlin/JS introduces some Experimental features to support static members in JavaScript and to create Kotlin collections from JavaScript.

Support for using Kotlin static members in JavaScript

> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Starting with Kotlin 2.0.20, you can use the `@JsStatic` annotation. It works similarly to @JvmStatic and instructs the compiler to generate additional static methods for the target declaration. This helps you use static members from your Kotlin code directly in JavaScript.

You can use the `@JsStatic` annotation for functions defined in named objects, as well as in companion objects declared inside classes and interfaces. The compiler generates both a static method of the object and an instance method in the object itself. For example:

```
class C {
    companion object {
        @JsStatic
        fun callStatic() {}
        fun callNonStatic() {}
    }
}
```

Now, `callStatic()` is static in JavaScript while `callNonStatic()` is not:

```
C.callStatic();              // Works, accessing the static function
C.callNonStatic();           // Error, not a static function in the generated JavaScript
C.Companion.callStatic();    // Instance method remains
C.Companion.callNonStatic(); // The only way it works
```

It's also possible to apply the `@JsStatic` annotation to a property of an object or a companion object, making its getter and setter methods static members in that object or the class containing the companion object.

Ability to create Kotlin collections from JavaScript

> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 2.0.0 introduced the ability to export Kotlin collections to JavaScript (and TypeScript). Now, the JetBrains team is taking another step to improve collection interoperability. Starting with Kotlin 2.0.20, it's possible to create Kotlin collections directly from the JavaScript/TypeScript side.

You can create Kotlin collections from JavaScript and pass them as arguments to the exported constructors or functions. As soon as you mention a collection inside an exported declaration, Kotlin generates a factory for the collection that is available in JavaScript/TypeScript.

Take a look at the following exported function:

```
// Kotlin
@JsExport
fun consumeMutableMap(map: MutableMap<String, Int>)
```

Since the `MutableMap` collection is mentioned, Kotlin generates an object with a factory method available from JavaScript/TypeScript. This factory method then creates a `MutableMap` from a JavaScript `Map`:

```
// JavaScript
import { consumeMutableMap } from "an-awesome-kotlin-module"
import { KtMutableMap } from "an-awesome-kotlin-module/kotlin-kotlin-stdlib"

consumeMutableMap(
    KtMutableMap.fromJsMap(new Map([["First", 1], ["Second", 2]]))
)
```

This feature is available for the `Set`, `Map`, and `List` Kotlin collection types and their mutable counterparts.

Gradle

Kotlin 2.0.20 is fully compatible with Gradle 6.8.3 through 8.6. Gradle 8.7 and 8.8 are also supported, with only one exception: If you use the Kotlin Multiplatform Gradle plugin, you may see deprecation warnings in your multiplatform projects calling the `withJava()` function in the JVM target. We plan to fix this issue as soon as possible.

For more information, see the issue in YouTrack.

You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

This version brings changes such as beginning the deprecation process for the old incremental compilation approach based on JVM history files, as well as a new way of sharing JVM artifacts between projects.

Deprecated incremental compilation based on JVM history files

In Kotlin 2.0.20, the incremental compilation approach based on JVM history files is deprecated in favor of the new incremental compilation approach that has been enabled by default since Kotlin 1.8.20.

The incremental compilation approach based on JVM history files suffered from limitations, such as not working with Gradle's build cache and not supporting compilation avoidance. In contrast, the new incremental compilation approach overcomes these limitations and has performed well since its introduction.

Given that the new incremental compilation approach has been used by default for the last two major Kotlin releases, the `kotlin.incremental.useClasspathSnapshot` Gradle property is deprecated in Kotlin 2.0.20. Therefore, if you use it to opt out, you will see a deprecation warning.

Option to share JVM artifacts between projects as class files

> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack. Opt-in is required (see details below).

In Kotlin 2.0.20, we introduce a new approach that changes the way the outputs of Kotlin/JVM compilations, such as JAR files, are shared between projects. With this approach, Gradle's `apiElements` configuration now has a secondary variant that provides access to the directory containing compiled `.class` files. When configured, your project uses this directory instead of requesting the compressed JAR artifact during compilation. This reduces the number of times JAR files are compressed and decompressed, especially for incremental builds.

Our testing shows that this new approach can provide build performance improvements for Linux and macOS hosts. However, on Windows hosts, we have seen a degradation in performance due to how Windows handles I/O operations when working with files.

To try this new approach, add the following property to your `gradle.properties` file:

```
kotlin.jvm.addClassesVariant=true
```

By default, this property is set to `false` and the `apiElements` variant in Gradle requests the compressed JAR artifact.

> Gradle has a related property that you can use in your Java-only projects to only expose the compressed JAR artifact during compilation **instead** of the directories containing compiled `.class` files:
>
> ```
> org.gradle.java.compile-classpath-packaging=true
> ```
>
> For more information on this property and its purpose, see Gradle's documentation on the Significant build performance drop on Windows for huge multi-projects.

We would appreciate your feedback on this new approach. Have you noticed any performance improvements while using it? Let us know by adding a comment in YouTrack.

Aligned dependency behavior of Kotlin Gradle plugin with java-test-fixtures plugin

Prior to Kotlin 2.0.20, if you used the `java-test-fixtures` plugin in your project, there was a difference between Gradle and the Kotlin Gradle plugin in how dependencies were propagated.

The Kotlin Gradle plugin propagated dependencies:

- From the `java-test-fixtures` plugin's `implementation` and `api` dependency types to the `test` source set compilation classpath.
- From the main source set's `implementation` and `api` dependency types to the `java-test-fixtures` plugin's source set compilation classpath.

However, Gradle only propagated dependencies in the `api` dependency types.

This difference in behavior led to some projects finding resource files multiple times in the classpath.

As of Kotlin 2.0.20, the Kotlin Gradle plugin's behavior is aligned with Gradle's `java-test-fixtures` plugin so this problem no longer occurs for this or other Gradle plugins.

As a result of this change, some dependencies in the `test` and `testFixtures` source sets may no longer be accessible. If this happens, either change the dependency declaration type from `implementation` to `api` or add a new dependency declaration on the affected source set.

Added task dependency for rare cases when the compile task lacks one on an artifact

Prior to 2.0.20, we found that there were scenarios where a compile task was missing a task dependency for one of its artifact inputs. This meant that the result of the dependent compile task was unstable, as sometimes the artifact had been generated in time, but sometimes, it hadn't.

To fix this issue, the Kotlin Gradle plugin now automatically adds the required task dependency in these scenarios.

In very rare cases, we've found that this new behavior can cause a circular dependency error. For example, if you have multiple compilations where one compilation can see all internal declarations of the other, and the generated artifact relies on the output of both compilation tasks, you could see an error like:

```
FAILURE: Build failed with an exception.

What went wrong:
Circular dependency between the following tasks:
:lib:compileKotlinJvm
--- :lib:jvmJar
     \--- :lib:compileKotlinJvm (*)
(*) - details omitted (listed previously)
```

To fix this circular dependency error, we've added a Gradle property: `archivesTaskOutputAsFriendModule`.

By default, this property is set to `true` to track the task dependency. To disable the use of the artifact in the compilation task, so that no task dependency is required, add the following in your `gradle.properties` file:

```
kotlin.build.archivesTaskOutputAsFriendModule=false
```

For more information, see the issue in YouTrack.

Compose compiler

In Kotlin 2.0.20, the Compose compiler gets a few improvements.

Fix for the unnecessary recompositions issue introduced in 2.0.0

Compose compiler 2.0.0 has an issue where it sometimes incorrectly infers the stability of types in multiplatform projects with non-JVM targets. This can lead to unnecessary (or even endless) recompositions. We strongly recommended updating your Compose apps made for Kotlin 2.0.0 to version 2.0.10 or newer.

If your app is built with Compose compiler 2.0.10 or newer but uses dependencies built with version 2.0.0, these older dependencies may still cause recomposition issues. To prevent this, update your dependencies to versions built with the same Compose compiler as your app.

New way to configure compiler options

We've introduced a new option configuration mechanism to avoid the churn of top-level parameters. It's harder for the Compose compiler team to test things out by creating or removing top-level entries for the `composeCompiler {}` block. So, options such as strong skipping mode and non-skipping group optimizations are now enabled through the `featureFlags` property. This property will be used to test new Compose compiler options that will eventually become default.

This change has also been applied to the Compose compiler Gradle plugin. To configure feature flags going forward, use the following syntax (this code will flip all of the default values):

```
composeCompiler {
    featureFlags = setOf(
        ComposeFeatureFlag.IntrinsicRemember.disabled(),
        ComposeFeatureFlag.OptimizeNonSkippingGroups,
        ComposeFeatureFlag.StrongSkipping.disabled()
    )
}
```

Or, if you are configuring the Compose compiler directly, use the following syntax:

```
-P plugin:androidx.compose.compiler.plugins.kotlin:featureFlag=IntrinsicRemember
```

The `enableIntrinsicRemember`, `enableNonSkippingGroupOptimization`, and `enableStrongSkippingMode` properties have been therefore deprecated.

We would appreciate any feedback you have on this new approach in YouTrack.

Strong skipping mode enabled by default

Strong skipping mode for the Compose compiler is now enabled by default.

Strong skipping mode is a Compose compiler configuration option that changes the rules for what composables can be skipped. With strong skipping mode enabled, composables with unstable parameters can now also be skipped. Strong skipping mode also automatically remembers lambdas used in composable functions, so you should no longer need to wrap your lambdas with `remember` to avoid recomposition.

For more details, see the strong skipping mode documentation.

Composition trace markers enabled by default

The `includeTraceMarkers` option is now set to `true` by default in the Compose compiler Gradle plugin to match the default value in the compiler plugin. This allows you to see composable functions in the Android Studio system trace profiler. For details about composition tracing, see this Android Developers blog post.

Non-skipping group optimizations

This release includes a new compiler option: when enabled, non-skippable and non-restartable composable functions will no longer generate a group around the body of the composable. This leads to fewer allocations and thus to improved performance. This option is experimental and disabled by default but can be enabled with the feature flag `OptimizeNonSkippingGroups` as shown above.

This feature flag is now ready for wider testing. Any issues found when enabling the feature can be filed on the Google issue tracker.

Support for default parameters in abstract composable functions

You can now add default parameters to abstract composable functions.

Previously, the Compose compiler would report an error when attempting to do this even though it is valid Kotlin code. We've now added support for this in the Compose compiler, and the restriction has been removed. This is especially useful for including default `Modifier` values:

```
abstract class Composables {
    @Composable
    abstract fun Composable(modifier: Modifier = Modifier)
}
```

Default parameters for open composable functions are still restricted in 2.0.20. This restriction will be addressed in future releases.

Standard library

The standard library now supports universally unique identifiers as an Experimental feature and includes some changes to Base64 decoding.

Support for UUIDs in the common Kotlin standard library

> This feature is Experimental. To opt in, use the `@ExperimentalUuidApi` annotation or the compiler option `-opt-in=kotlin.uuid.ExperimentalUuidApi`.

Kotlin 2.0.20 introduces a class for representing UUIDs (universally unique identifiers) in the common Kotlin standard library to address the challenge of uniquely identifying items.

Additionally, this feature provides APIs for the following UUID-related operations:

- Generating UUIDs.
- Parsing UUIDs from and formatting them to their string representations.
- Creating UUIDs from specified 128-bit values.
- Accessing the 128 bits of a UUID.

The following code example demonstrates these operations:

```
// Constructs a byte array for UUID creation
val byteArray = byteArrayOf(
    0x55, 0x0E, 0x84.toByte(), 0x00, 0xE2.toByte(), 0x9B.toByte(), 0x41, 0xD4.toByte(),
    0xA7.toByte(), 0x16, 0x44, 0x66, 0x55, 0x44, 0x00, 0x00
)

val uuid1 = Uuid.fromByteArray(byteArray)
val uuid2 = Uuid.fromULongs(0x550E8400E29B41D4uL, 0xA716446655440000uL)
val uuid3 = Uuid.parse("550e8400-e29b-41d4-a716-446655440000")

println(uuid1)
// 550e8400-e29b-41d4-a716-446655440000
println(uuid1 == uuid2)
// true
println(uuid2 == uuid3)
// true

// Accesses UUID bits
val version = uuid1.toLongs { mostSignificantBits, _ ->
    ((mostSignificantBits shr 12) and 0xF).toInt()
}
println(version)
// 4

// Generates a random UUID
val randomUuid = Uuid.random()

println(uuid1 == randomUuid)
// false
```

To maintain compatibility with APIs that use `java.util.UUID`, there are two extension functions in Kotlin/JVM for converting between `java.util.UUID` and `kotlin.uuid.Uuid`: `.toJavaUuid()` and `.toKotlinUuid()`. For example:

```
val kotlinUuid = Uuid.parseHex("550e8400e29b41d4a716446655440000")
// Converts Kotlin UUID to java.util.UUID
val javaUuid = kotlinUuid.toJavaUuid()

val javaUuid = java.util.UUID.fromString("550e8400-e29b-41d4-a716-446655440000")
// Converts Java UUID to kotlin.uuid.Uuid
val kotlinUuid = javaUuid.toKotlinUuid()
```

This feature and the provided APIs simplify multiplatform software development by allowing code sharing among multiple platforms. UUIDs are also ideal in environments where generating unique identifiers is difficult.

Some example use cases involving UUIDs include:

- Assigning unique IDs to database records.
- Generating web session identifiers.
- Any scenario requiring unique identification or tracking.

Support for minLength in HexFormat

> The `HexFormat` class and its properties are Experimental. To opt in, use the `@OptIn(ExperimentalStdlibApi::class)` annotation or the compiler option `-opt-in=kotlin.ExperimentalStdlibApi`.

Kotlin 2.0.20 adds a new `minLength` property to the `NumberHexFormat` class, accessed through `HexFormat.number`. This property lets you specify the minimum number of digits in hexadecimal representations of numeric values, enabling padding with zeros to meet the required length. Additionally, leading zeros can be trimmed using the `removeLeadingZeros` property:

```
fun main() {
    println(93.toHexString(HexFormat {
        number.minLength = 4
        number.removeLeadingZeros = true
    }))
    // "005d"
}
```

The `minLength` property does not affect parsing. However, parsing now allows hex strings to have more digits than the type's width if the extra leading digits are zeros.

Changes to the Base64's decoder behavior

The `Base64` class and its related features are Experimental. To opt in, use the `@OptIn(ExperimentalEncodingApi::class)` annotation or the compiler option `-opt-in=kotlin.io.encoding.ExperimentalEncodingApi`.

Two changes were introduced to the Base64 decoder's behavior in Kotlin 2.0.20:

- The Base64 decoder now requires padding
- A `withPadding` function has been added for padding configuration

The Base64 decoder now requires padding

The Base64 encoder now adds padding by default, and the decoder requires padding and prohibits non-zero pad bits when decoding.

withPadding function for padding configuration

A new `.withPadding()` function has been introduced to give users control over the padding behavior of Base64 encoding and decoding:

```
val base64 = Base64.UrlSafe.withPadding(Base64.PaddingOption.ABSENT_OPTIONAL)
```

This function enables the creation of `Base64` instances with different padding options:

`PaddingOption`	On encode	On decode
`PRESENT`	Add padding	Padding is required
`ABSENT`	Omit padding	No padding allowed
`PRESENT_OPTIONAL`	Add padding	Padding is optional

PaddingOption	On encode	On decode
ABSENT_OPTIONAL	Omit padding	Padding is optional

You can create `Base64` instances with different padding options and use them to encode and decode data:

```
import kotlin.io.encoding.Base64
import kotlin.io.encoding.ExperimentalEncodingApi

@OptIn(ExperimentalEncodingApi::class)
fun main() {
    // Example data to encode
    val data = "fooba".toByteArray()

    // Creates a Base64 instance with URL-safe alphabet and PRESENT padding
    val base64Present = Base64.UrlSafe.withPadding(Base64.PaddingOption.PRESENT)
    val encodedDataPresent = base64Present.encode(data)
    println("Encoded data with PRESENT padding: $encodedDataPresent")
    // Encoded data with PRESENT padding: Zm9vYmE=

    // Creates a Base64 instance with URL-safe alphabet and ABSENT padding
    val base64Absent = Base64.UrlSafe.withPadding(Base64.PaddingOption.ABSENT)
    val encodedDataAbsent = base64Absent.encode(data)
    println("Encoded data with ABSENT padding: $encodedDataAbsent")
    // Encoded data with ABSENT padding: Zm9vYmE

    // Decodes the data back
    val decodedDataPresent = base64Present.decode(encodedDataPresent)
    println("Decoded data with PRESENT padding: ${String(decodedDataPresent)}")
    // Decoded data with PRESENT padding: fooba

    val decodedDataAbsent = base64Absent.decode(encodedDataAbsent)
    println("Decoded data with ABSENT padding: ${String(decodedDataAbsent)}")
    // Decoded data with ABSENT padding: fooba
}
```

Documentation updates

The Kotlin documentation has received some notable changes:

- Improved Standard input page - Learn how to use Java Scanner and `readln()`.
- Improved K2 compiler migration guide - Learn about performance improvements, compatibility with Kotlin libraries and what to do with your custom compiler plugins.
- Improved Exceptions page - Learn about exceptions, how to throw and catch them.
- Improved Test code using JUnit in JVM - tutorial - Learn how to create tests using JUnit.
- Improved Interoperability with Swift/Objective-C page - Learn how to use Kotlin declarations in Swift/Objective-C code and Objective-C declarations in Kotlin code.
- Improved Swift package export setup page - Learn how to set up Kotlin/Native output that can be consumed by a Swift package manager dependency.

Install Kotlin 2.0.20

Starting from IntelliJ IDEA 2023.3 and Android Studio Iguana (2023.2.1) Canary 15, the Kotlin plugin is distributed as a bundled plugin included in your IDE. This means that you can't install the plugin from JetBrains Marketplace anymore.

To update to the new Kotlin version, change the Kotlin version to 2.0.20 in your build scripts.

What's new in Kotlin 2.0.0

Released: May 21, 2024

The Kotlin 2.0.0 release is out and the new Kotlin K2 compiler is Stable! Additionally, here are some other highlights:

- New Compose compiler Gradle plugin
- Generation of lambda functions using invokedynamic
- The kotlinx-metadata-jvm library is now Stable
- Monitoring GC performance in Kotlin/Native with signposts on Apple platforms
- Resolving conflicts in Kotlin/Native with Objective-C methods
- Support for named export in Kotlin/Wasm
- Support for unsigned primitive types in functions with @JsExport in Kotlin/Wasm
- Optimize production builds by default using Binaryen
- New Gradle DSL for compiler options in multiplatform projects
- Stable replacement of the enum class values generic function
- Stable AutoCloseable interface

Kotlin 2.0 is a huge milestone for the JetBrains team. This release was the center of KotlinConf 2024. Check out the opening keynote, where we announced exciting updates and addressed the recent work on the Kotlin language:

Watch video online.

For information about the Kotlin release cycle, see Kotlin release process.

IDE support

The Kotlin plugins that support Kotlin 2.0.0 are bundled in the latest IntelliJ IDEA and Android Studio. You don't need to update the Kotlin plugin in your IDE. All you need to do is to change the Kotlin version to Kotlin 2.0.0 in your build scripts.

- For details about IntelliJ IDEA's support for the Kotlin K2 compiler, see Support in IDEs.
- For more details about IntelliJ IDEA's support for Kotlin, see Kotlin releases.

Kotlin K2 compiler

The road to the K2 compiler has been a long one, but now the JetBrains team is finally ready to announce its stabilization. In Kotlin 2.0.0, the new Kotlin K2 compiler is used by default and it is Stable for all target platforms: JVM, Native, Wasm, and JS. The new compiler brings major performance improvements, speeds up new language feature development, unifies all platforms that Kotlin supports, and provides a better architecture for multiplatform projects.

The JetBrains team has ensured the quality of the new compiler by successfully compiling 10 million lines of code from selected user and internal projects. 18,000 developers were involved in the stabilization process, testing the new K2 compiler across a total of 80,000 projects and reporting any problems they found.

To help make the migration process to the new compiler as smooth as possible, we've created a K2 compiler migration guide. This guide explains the many benefits of the compiler, highlights any changes you might encounter, and describes how to roll back to the previous version if necessary.

In a blog post, we explored the performance of the K2 compiler in different projects. Check it out if you'd like to see real data on how the K2 compiler performs and find instructions on how to collect performance benchmarks from your own projects.

You can also watch this talk from KotlinConf 2024, where Michail Zarečenskij, the lead language designer, discusses the feature evolution in Kotlin and the K2 compiler:

Watch video online.

Current K2 compiler limitations

Enabling K2 in your Gradle project comes with certain limitations that can affect projects using Gradle versions below 8.3 in the following cases:

- Compilation of source code from `buildSrc`.
- Compilation of Gradle plugins in included builds.
- Compilation of other Gradle plugins if they are used in projects with Gradle versions below 8.3.
- Building Gradle plugin dependencies.

If you encounter any of the problems mentioned above, you can take the following steps to address them:

- Set the language version for `buildSrc`, any Gradle plugins, and their dependencies:

```
kotlin {
    compilerOptions {
        languageVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9)
        apiVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9)
    }
}
```

> If you configure language and API versions for specific tasks, these values will override the values set by the `compilerOptions` extension. In this case, language and API versions should not be higher than 1.9.

- Update the Gradle version in your project to 8.3 or later.

Smart cast improvements

The Kotlin compiler can automatically cast an object to a type in specific cases, saving you the trouble of having to explicitly cast it yourself. This is called smart casting. The Kotlin K2 compiler now performs smart casts in even more scenarios than before.

In Kotlin 2.0.0, we've made improvements related to smart casts in the following areas:

- Local variables and further scopes
- Type checks with logical `or` operator
- Inline functions
- Properties with function types
- Exception handling
- Increment and decrement operators

Local variables and further scopes

Previously, if a variable was evaluated as not `null` within an `if` condition, the variable would be smart-cast. Information about this variable would then be shared further within the scope of the `if` block.

However, if you declared the variable **outside** the `if` condition, no information about the variable would be available within the `if` condition, so it couldn't be smart-cast. This behavior was also seen with `when` expressions and `while` loops.

From Kotlin 2.0.0, if you declare a variable before using it in your `if`, `when`, or `while` condition, then any information collected by the compiler about the variable will be accessible in the corresponding block for smart-casting.

This can be useful when you want to do things like extract boolean conditions into variables. Then, you can give the variable a meaningful name, which will improve your code readability and make it possible to reuse the variable later in your code. For example:

```
class Cat {
    fun purr() {
        println("Purr purr")
    }
}

fun petAnimal(animal: Any) {
    val isCat = animal is Cat
    if (isCat) {
        // In Kotlin 2.0.0, the compiler can access
        // information about isCat, so it knows that
        // animal was smart-cast to the type Cat.
        // Therefore, the purr() function can be called.
        // In Kotlin 1.9.20, the compiler doesn't know
        // about the smart cast, so calling the purr()
        // function triggers an error.
        animal.purr()
    }
}

fun main() {
    val kitty = Cat()
    petAnimal(kitty)
    // Purr purr
}
```

Type checks with logical or operator

In Kotlin 2.0.0, if you combine type checks for objects with an `or` operator (`||`), a smart cast is made to their closest common supertype. Before this change, a smart cast was always made to the `Any` type.

In this case, you still had to manually check the object type afterward before you could access any of its properties or call its functions. For example:

```
interface Status {
    fun signal() {}
}

interface Ok : Status
interface Postponed : Status
interface Declined : Status

fun signalCheck(signalStatus: Any) {
    if (signalStatus is Postponed || signalStatus is Declined) {
        // signalStatus is smart-cast to a common supertype Status
        signalStatus.signal()
        // Prior to Kotlin 2.0.0, signalStatus is smart cast
        // to type Any, so calling the signal() function triggered an
        // Unresolved reference error. The signal() function can only
        // be called successfully after another type check:

        // check(signalStatus is Status)
        // signalStatus.signal()
    }
}
```

The common supertype is an **approximation** of a union type. Union types are not supported in Kotlin.

Inline functions

In Kotlin 2.0.0, the K2 compiler treats inline functions differently, allowing it to determine in combination with other compiler analyses whether it's safe to smart-cast.

Specifically, inline functions are now treated as having an implicit `callsInPlace` contract. This means that any lambda functions passed to an inline function are called in place. Since lambda functions are called in place, the compiler knows that a lambda function can't leak references to any variables contained within its function body.

The compiler uses this knowledge along with other compiler analyses to decide whether it's safe to smart-cast any of the captured variables. For example:

```
interface Processor {
    fun process()
}

inline fun inlineAction(f: () -> Unit) = f()

fun nextProcessor(): Processor? = null

fun runProcessor(): Processor? {
    var processor: Processor? = null
    inlineAction {
        // In Kotlin 2.0.0, the compiler knows that processor
        // is a local variable, and inlineAction() is an inline function, so
        // references to processor can't be leaked. Therefore, it's safe
        // to smart-cast processor.

        // If processor isn't null, processor is smart-cast
        if (processor ≠ null) {
            // The compiler knows that processor isn't null, so no safe call
            // is needed
            processor.process()

            // In Kotlin 1.9.20, you have to perform a safe call:
            // processor?.process()
        }

        processor = nextProcessor()
    }

    return processor
}
```

Properties with function types

In previous versions of Kotlin, there was a bug that meant that class properties with a function type weren't smart-cast. We fixed this behavior in Kotlin 2.0.0 and the K2 compiler. For example:

```
class Holder(val provider: (() -> Unit)?) {
    fun process() {
        // In Kotlin 2.0.0, if provider isn't null, then
        // provider is smart-cast
        if (provider != null) {
            // The compiler knows that provider isn't null
            provider()

            // In 1.9.20, the compiler doesn't know that provider isn't
            // null, so it triggers an error:
            // Reference has a nullable type '(() -> Unit)?', use explicit '?.invoke()' to make a
function-like call instead
        }
    }
}
```

This change also applies if you overload your `invoke` operator. For example:

```
interface Provider {
    operator fun invoke()
}

interface Processor : () -> String

class Holder(val provider: Provider?, val processor: Processor?) {
    fun process() {
        if (provider != null) {
            provider()
            // In 1.9.20, the compiler triggers an error:
            // Reference has a nullable type 'Provider?' use explicit '?.invoke()' to make a function-
like call instead
        }
    }
}
```

Exception handling

In Kotlin 2.0.0, we've made improvements to exception handling so that smart cast information can be passed on to `catch` and `finally` blocks. This change makes your code safer as the compiler keeps track of whether your object has a nullable type. For example:

```
fun testString() {
    var stringInput: String? = null
    // stringInput is smart-cast to String type
    stringInput = ""
    try {
        // The compiler knows that stringInput isn't null
        println(stringInput.length)
        // 0

        // The compiler rejects previous smart cast information for
        // stringInput. Now stringInput has the String? type.
        stringInput = null

        // Trigger an exception
        if (2 > 1) throw Exception()
        stringInput = ""
    } catch (exception: Exception) {
        // In Kotlin 2.0.0, the compiler knows stringInput
        // can be null, so stringInput stays nullable.
        println(stringInput?.length)
        // null

        // In Kotlin 1.9.20, the compiler says that a safe call isn't
        // needed, but this is incorrect.
    }
}

fun main() {
    testString()
}
```

Increment and decrement operators

Prior to Kotlin 2.0.0, the compiler didn't understand that the type of an object can change after using an increment or decrement operator. As the compiler couldn't accurately track the object type, your code could lead to unresolved reference errors. In Kotlin 2.0.0, this has been fixed:

```
interface Rho {
    operator fun inc(): Sigma = TODO()
}

interface Sigma : Rho {
    fun sigma() = Unit
}

interface Tau {
    fun tau() = Unit
}

fun main(input: Rho) {
    var unknownObject: Rho = input

    // Check if unknownObject inherits from the Tau interface
    // Note, it's possible that unknownObject inherits from both
    // Rho and Tau interfaces.
    if (unknownObject is Tau) {

        // Use the overloaded inc() operator from interface Rho.
        // In Kotlin 2.0.0, the type of unknownObject is smart-cast to
        // Sigma.
        ++unknownObject

        // In Kotlin 2.0.0, the compiler knows unknownObject has type
        // Sigma, so the sigma() function can be called successfully.
        unknownObject.sigma()

        // In Kotlin 1.9.20, the compiler doesn't perform a smart cast
        // when inc() is called so the compiler still thinks that
        // unknownObject has type Tau. Calling the sigma() function
        // throws a compile-time error.

        // In Kotlin 2.0.0, the compiler knows unknownObject has type
        // Sigma, so calling the tau() function throws a compile-time
        // error.
        unknownObject.tau()
        // Unresolved reference 'tau'

        // In Kotlin 1.9.20, since the compiler mistakenly thinks that
        // unknownObject has type Tau, the tau() function can be called,
        // but it throws a ClassCastException.
    }
}
```

Kotlin Multiplatform improvements

In Kotlin 2.0.0, we've made improvements in the K2 compiler related to Kotlin Multiplatform in the following areas:

- Separation of common and platform sources during compilation
- Different visibility levels of expected and actual declarations

Separation of common and platform sources during compilation

Previously, the design of the Kotlin compiler prevented it from keeping common and platform source sets separate at compile time. As a consequence, common code could access platform code, which resulted in different behavior between platforms. In addition, some compiler settings and dependencies from common code used to leak into platform code.

In Kotlin 2.0.0, our implementation of the new Kotlin K2 compiler included a redesign of the compilation scheme to ensure strict separation between common and platform source sets. This change is most noticeable when you use expected and actual functions. Previously, it was possible for a function call in your common code to resolve to a function in platform code. For example:

Common code

```
fun foo(x: Any) = println("common foo")

fun exampleFunction() {
    foo(42)
}
```

Platform code

```
// JVM
fun foo(x: Int) = println("platform foo")

// JavaScript
// There is no foo() function overload
// on the JavaScript platform
```

In this example, the common code has different behavior depending on which platform it is run on:

- On the JVM platform, calling the `foo()` function in the common code results in the `foo()` function from the platform code being called as `platform foo`.
- On the JavaScript platform, calling the `foo()` function in the common code results in the `foo()` function from the common code being called as `common foo`, as there is no such function available in the platform code.

In Kotlin 2.0.0, common code doesn't have access to platform code, so both platforms successfully resolve the `foo()` function to the `foo()` function in the common code: `common foo`.

In addition to the improved consistency of behavior across platforms, we also worked hard to fix cases where there was conflicting behavior between IntelliJ IDEA or Android Studio and the compiler. For instance, when you used expected and actual classes, the following would happen:

Common code

```
expect class Identity {
    fun confirmIdentity(): String
}

fun common() {
    // Before 2.0.0,
    // it triggers an IDE-only error
    Identity().confirmIdentity()
    // RESOLUTION_TO_CLASSIFIER : Expected
class
    // Identity has no default constructor.
}
```

Platform code

```
actual class Identity {
    actual fun confirmIdentity() = "expect class
fun: jvm"
}
```

In this example, the expected class `Identity` has no default constructor, so it can't be called successfully in common code. Previously, an error was only reported by the IDE, but the code still compiled successfully on the JVM. However, now the compiler correctly reports an error:

```
Expected class 'expect class Identity : Any' does not have default constructor
```

When resolution behavior doesn't change

We're still in the process of migrating to the new compilation scheme, so the resolution behavior is still the same when you call functions that aren't within the same source set. You'll notice this difference mainly when you use overloads from a multiplatform library in your common code.

Suppose you have a library, which has two `whichFun()` functions with different signatures:

```
// Example library

// MODULE: common
fun whichFun(x: Any) = println("common function")

// MODULE: JVM
fun whichFun(x: Int) = println("platform function")
```

If you call the `whichFun()` function in your common code, the function that has the most relevant argument type in the library is resolved:

```
// A project that uses the example library for the JVM target

// MODULE: common
fun main() {
    whichFun(2)
    // platform function
}
```

In comparison, if you declare the overloads for `whichFun()` within the same source set, the function from the common code will be resolved because your code doesn't have access to the platform-specific version:

```
// Example library isn't used

// MODULE: common
fun whichFun(x: Any) = println("common function")

fun main() {
    whichFun(2)
    // common function
}

// MODULE: JVM
fun whichFun(x: Int) = println("platform function")
```

Similar to multiplatform libraries, since the `commonTest` module is in a separate source set, it also still has access to platform-specific code. Therefore, the resolution of calls to functions in the `commonTest` module exhibits the same behavior as in the old compilation scheme.

In the future, these remaining cases will be more consistent with the new compilation scheme.

Different visibility levels of expected and actual declarations

Before Kotlin 2.0.0, if you used expected and actual declarations in your Kotlin Multiplatform project, they had to have the same visibility level. Kotlin 2.0.0 now also supports different visibility levels but **only** if the actual declaration is *more* permissive than the expected declaration. For example:

```
expect internal class Attribute // Visibility is internal
actual class Attribute          // Visibility is public by default,
                                // which is more permissive
```

Similarly, if you are using a type alias in your actual declaration, the visibility of the **underlying type** should be the same or more permissive than the expected declaration. For example:

```
expect internal class Attribute                 // Visibility is internal
internal actual typealias Attribute = Expanded

class Expanded                                  // Visibility is public by default,
                                                // which is more permissive
```

Compiler plugins support

Currently, the Kotlin K2 compiler supports the following Kotlin compiler plugins:

- `all-open`
- AtomicFU
- `jvm-abi-gen`
- `js-plain-objects`
- kapt
- Lombok
- `no-arg`
- Parcelize
- SAM with receiver
- serialization
- Power-assert

In addition, the Kotlin K2 compiler supports:

- The Jetpack Compose compiler plugin 2.0.0, which was moved into the Kotlin repository.
- The Kotlin Symbol Processing (KSP) plugin since KSP2.

If you use any additional compiler plugins, check their documentation to see if they are compatible with K2.

Experimental Kotlin Power-assert compiler plugin

The Kotlin Power-assert plugin is Experimental. It may be changed at any time.

Kotlin 2.0.0 introduces an experimental Power-assert compiler plugin. This plugin improves the experience of writing tests by including contextual information in failure messages, making debugging easier and more efficient.

Developers often need to use complex assertion libraries to write effective tests. The Power-assert plugin simplifies this process by automatically generating failure messages that include intermediate values of the assertion expression. This helps developers quickly understand why a test failed.

When an assertion fails in a test, the improved error message shows the values of all variables and sub-expressions within the assertion, making it clear which part of the condition caused the failure. This is particularly useful for complex assertions where multiple conditions are checked.

To enable the plugin in your project, configure it in your `build.gradle(.kts)` file:

Kotlin

```
plugins {
    kotlin("multiplatform") version "2.0.0"
    kotlin("plugin.power-assert") version "2.0.0"
}

powerAssert {
    functions = listOf("kotlin.assert", "kotlin.test.assertTrue")
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.0.0'
    id 'org.jetbrains.kotlin.plugin.power-assert' version '2.0.0'
}

powerAssert {
    functions = ["kotlin.assert", "kotlin.test.assertTrue"]
}
```

Learn more about the Kotlin Power-assert plugin in the documentation.

How to enable the Kotlin K2 compiler

Starting with Kotlin 2.0.0, the Kotlin K2 compiler is enabled by default. No additional actions are required.

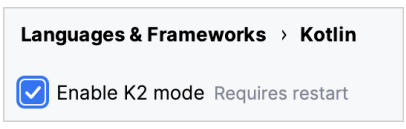
Try the Kotlin K2 compiler in Kotlin Playground

Kotlin Playground supports the 2.0.0 release. Check it out!

Support in IDEs

By default, IntelliJ IDEA and Android Studio still use the previous compiler for code analysis, code completion, highlighting, and other IDE-related features. To get the full Kotlin 2.0 experience in your IDE, enable K2 mode.

In your IDE, go to **Settings** | **Languages & Frameworks** | **Kotlin** and select the **Enable K2 mode** option. The IDE will analyze your code using its K2 mode.



Enable K2 mode

After enabling K2 mode, you may notice differences in IDE analysis due to changes in compiler behavior. Learn how the new K2 compiler differs from the previous one in our migration guide.

- Learn more about K2 mode in our blog.
- We are actively collecting feedback about K2 mode, so please share your thoughts in our public Slack channel.

Leave your feedback on the new K2 compiler

We would appreciate any feedback you may have!

- Report any problems you face with the new K2 compiler in our issue tracker.
- Enable the "Send usage statistics" option to allow JetBrains to collect anonymous data about K2 usage.

Kotlin/JVM

Starting with version 2.0.0, the compiler can generate classes containing Java 22 bytecode. This version also brings the following changes:

- Generation of lambda functions using invokedynamic
- The kotlinx-metadata-jvm library is now Stable

Generation of lambda functions using invokedynamic

Kotlin 2.0.0 introduces a new default method for generating lambda functions using `invokedynamic`. This change reduces the binary sizes of applications compared to the traditional anonymous class generation.

Since the first version, Kotlin has generated lambdas as anonymous classes. However, starting from Kotlin 1.5.0, the option for `invokedynamic` generation has been available by using the `-Xlambdas=indy` compiler option. In Kotlin 2.0.0, `invokedynamic` has become the default method for lambda generation. This method produces lighter binaries and aligns Kotlin with JVM optimizations, ensuring applications benefit from ongoing and future improvements in JVM performance.

Currently, it has three limitations compared to ordinary lambda compilation:

- A lambda compiled into `invokedynamic` is not serializable.
- Experimental `reflect()` API does not support lambdas generated by `invokedynamic`.
- Calling `.toString()` on such a lambda produces a less readable string representation:

```
fun main() {
    println({})

    // With Kotlin 1.9.24 and reflection, returns
    // () → kotlin.Unit

    // With Kotlin 2.0.0, returns
    // FileKt$$Lambda$13/0x00007f88a0004608@506e1b77
}
```

To retain the legacy behavior of generating lambda functions, you can either:

- Annotate specific lambdas with `@JvmSerializableLambda`.
- Use the compiler option `-Xlambdas=class` to generate all lambdas in a module using the legacy method.

The kotlinx-metadata-jvm library is Stable

In Kotlin 2.0.0, the `kotlinx-metadata-jvm` library became Stable. Now that the library has changed to the `kotlin` package and coordinates, you can find it as `kotlin-metadata-jvm` (without the "x").

Previously, the `kotlinx-metadata-jvm` library had its own publishing scheme and version. Now, we will build and publish the `kotlin-metadata-jvm` updates as part of the Kotlin release cycle, with the same backward compatibility guarantees as the Kotlin standard library.

The `kotlin-metadata-jvm` library provides an API to read and modify metadata of binary files generated by the Kotlin/JVM compiler.

Kotlin/Native

This version brings the following changes:

- Monitoring GC performance with signposts
- Resolving conflicts with Objective-C methods
- Changed log level for compiler arguments in Kotlin/Native
- Explicitly added standard library and platform dependencies to Kotlin/Native
- Tasks error in Gradle configuration cache

Monitoring GC performance with signposts on Apple platforms

Previously, it was only possible to monitor the performance of Kotlin/Native's garbage collector (GC) by looking into logs. However, these logs were not integrated with Xcode Instruments, a popular toolkit for investigating issues with iOS apps' performance.

Since Kotlin 2.0.0, GC reports pauses with signposts that are available in Instruments. Signposts allow for custom logging within your app, so now, when debugging iOS app performance, you can check if a GC pause corresponds to the application freeze.

Learn more about GC performance analysis in the documentation.

Resolving conflicts with Objective-C methods

Objective-C methods can have different names, but the same number and types of parameters. For example, `locationManager:didEnterRegion:` and `locationManager:didExitRegion:`. In Kotlin, these methods have the same signature, so an attempt to use them triggers a conflicting overloads error.

Previously, you had to manually suppress conflicting overloads to avoid this compilation error. To improve Kotlin interoperability with Objective-C, the Kotlin 2.0.0 introduces the new `@ObjCSignatureOverride` annotation.

The annotation instructs the Kotlin compiler to ignore conflicting overloads, in case several functions with the same argument types but different argument names are inherited from the Objective-C class.

Applying this annotation is also safer than general error suppression. This annotation can only be used in the case of overriding Objective-C methods, which are supported and tested, while general suppression may hide important errors and lead to silently broken code.

Changed log level for compiler arguments

In this release, the log level for compiler arguments in Kotlin/Native Gradle tasks, such as `compile`, `link`, and `cinterop`, has changed from `info` to `debug`.

With `debug` as its default value, the log level is consistent with other Gradle compilation tasks and provides detailed debugging information, including all compiler arguments.

Explicitly added standard library and platform dependencies to Kotlin/Native

Previously, the Kotlin/Native compiler resolved standard library and platform dependencies implicitly, which caused inconsistencies in the way the Kotlin Gradle plugin worked across Kotlin targets.

Now, each Kotlin/Native Gradle compilation explicitly includes standard library and platform dependencies in its compile-time library path via the `compileDependencyFiles` compilation parameter.

Tasks error in Gradle configuration cache

Since Kotlin 2.0.0, you may encounter a configuration cache error with messages indicating: `invocation of Task.project at execution time is unsupported`.

This error appears in tasks such as `NativeDistributionCommonizerTask` and `KotlinNativeCompile`.

However, this is a false-positive error. The underlying issue is the presence of tasks that are not compatible with the Gradle configuration cache, like the `publish*` task.

This discrepancy may not be immediately apparent, as the error message suggests a different root cause.

As the precise cause isn't explicitly stated in the error report, the Gradle team is already addressing the issue to fix reports.

Kotlin/Wasm

Kotlin 2.0.0 improves performance and interoperability with JavaScript:

- Optimized production builds by default using Binaryen
- Support for named export
- Support for unsigned primitive types in functions with `@JsExport`
- Generation of TypeScript declaration files in Kotlin/Wasm
- Support for catching JavaScript exceptions
- New exception handling proposal is now supported as an option
- The `withWasm()` function is split into JS and WASI variants

Optimized production builds by default using Binaryen

The Kotlin/Wasm toolchain now applies the Binaryen tool during production compilation to all projects, as opposed to the previous manual setup approach. By our estimations, it should improve runtime performance and reduce the binary size for your project.

This change only affects production compilation. The development compilation process stays the same.

Support for named export

Previously, all exported declarations from Kotlin/Wasm were imported into JavaScript using default export:

```
//JavaScript:
import Module from "./index.mjs"

Module.add()
```

Now, you can import each Kotlin declaration marked with `@JsExport` by name:

```
// Kotlin:
@JsExport
fun add(a: Int, b: Int) = a + b
```

```
//JavaScript:
import { add } from "./index.mjs"
```

Named exports make it easier to share code between Kotlin and JavaScript modules. They improve readability and help you manage dependencies between modules.

Support for unsigned primitive types in functions with @JsExport

Starting from Kotlin 2.0.0, you can use unsigned primitive types inside external declarations and functions with the `@JsExport` annotation that makes Kotlin/Wasm functions available in JavaScript code.

This helps to mitigate the previous limitation that prevented the unsigned primitives from being used directly inside exported and external declarations. Now you can export functions with unsigned primitives as a return or parameter type and consume external declarations that return or consume unsigned primitives.

For more information on Kotlin/Wasm interoperability with JavaScript, see the documentation.

Generation of TypeScript declaration files in Kotlin/Wasm

> Generating TypeScript declaration files in Kotlin/Wasm is Experimental. It may be dropped or changed at any time.

In Kotlin 2.0.0, the Kotlin/Wasm compiler is now capable of generating TypeScript definitions from any `@JsExport` declarations in your Kotlin code. These definitions can be used by IDEs and JavaScript tools to provide code autocompletion, help with type checks, and make it easier to include Kotlin code in JavaScript.

The Kotlin/Wasm compiler collects any top-level functions marked with `@JsExport` and automatically generates TypeScript definitions in a `.d.ts` file.

To generate TypeScript definitions, in your `build.gradle(.kts)` file in the `wasmJs {}` block, add the `generateTypeScriptDefinitions()` function:

```
kotlin {
    wasmJs {
        binaries.executable()
        browser {
        }
        generateTypeScriptDefinitions()
    }
}
```

Support for catching JavaScript exceptions

Previously, Kotlin/Wasm code could not catch JavaScript exceptions, making it difficult to handle errors originating from the JavaScript side of the program.

In Kotlin 2.0.0, we have implemented support for catching JavaScript exceptions within Kotlin/Wasm. This implementation allows you to use `try-catch` blocks, with specific types like `Throwable` or `JsException`, to handle these errors properly.

Additionally, `finally` blocks, which help execute code regardless of exceptions, also work correctly. While we're introducing support for catching JavaScript exceptions, no additional information is provided when a JavaScript exception, like a call stack, occurs. However, we are working on these implementations.

New exception handling proposal is now supported as an option

In this release, we introduce support for the new version of WebAssembly's exception handling proposal within Kotlin/Wasm.

This update ensures the new proposal aligns with Kotlin requirements, enabling the use of Kotlin/Wasm on virtual machines that only support the latest version of the proposal.

Activate the new exception handling proposal by using the `-Xwasm-use-new-exception-proposal` compiler option, which is turned off by default.

The withWasm() function is split into JS and WASI variants

The `withWasm()` function, which used to provide Wasm targets for hierarchy templates, is deprecated in favor of specialized `withWasmJs()` and `withWasmWasi()` functions.

Now you can separate the WASI and JS targets between different groups in the tree definition.

Kotlin/JS

Among other changes, this version brings modern JS compilation to Kotlin, supporting more features from the ES2015 standard:

- New compilation target
- Suspend functions as ES2015 generators
- Passing arguments to the main function
- Per-file compilation for Kotlin/JS projects
- Improved collection interoperability
- Support for createInstance()

- Support for type-safe plain JavaScript objects
- Support for npm package manager
- Changes to compilation tasks
- Discontinuing legacy Kotlin/JS JAR artifacts

New compilation target

In Kotlin 2.0.0, we're adding a new compilation target to Kotlin/JS, `es2015`. This is a new way for you to enable all the ES2015 features supported in Kotlin at once.

You can set it up in your `build.gradle(.kts)` file like this:

```
kotlin {
    js {
        compilerOptions {
            target.set("es2015")
        }
    }
}
```

The new target automatically turns on ES classes and modules and the newly supported ES generators.

Suspend functions as ES2015 generators

This release introduces Experimental support for ES2015 generators for compiling suspend functions.

Using generators instead of state machines should improve the final bundle size of your project. For example, the JetBrains team managed to decrease the bundle size of its Space project by 20% by using the ES2015 generators.

Learn more about ES2015 (ECMAScript 2015, ES6) in the official documentation.

Passing arguments to the main function

Starting with Kotlin 2.0.0, you can specify a source of your `args` for the `main()` function. This feature makes it easier to work with the command line and pass the arguments.

To do this, define the `js {}` block with the new `passAsArgumentToMainFunction()` function, which returns an array of strings:

```
kotlin {
    js {
        binary.executable()
        passAsArgumentToMainFunction("Deno.args")
    }
}
```

The function is executed at runtime. It takes the JavaScript expression and uses it as the `args: Array<String>` argument instead of the `main()` function call.

Also, if you use the Node.js runtime, you can take advantage of a special alias. It allows you to pass `process.argv` to the `args` parameter once instead of adding it manually every time:

```
kotlin {
    js {
        binary.executable()
        nodejs {
            passProcessArgvToMainFunction()
        }
    }
}
```

Per-file compilation for Kotlin/JS projects

Kotlin 2.0.0 introduces a new granularity option for the Kotlin/JS project output. You can now set up a per-file compilation that generates one JavaScript file for each Kotlin file. It helps to significantly optimize the size of the final bundle and improve the loading time of the program.

Previously, there were only two output options. The Kotlin/JS compiler could generate a single `.js` file for the whole project. However, this file might be too large and inconvenient to use. Whenever you wanted to use a function from your project, you had to include the entire JavaScript file as a dependency. Alternatively, you could configure a compilation of a separate `.js` file for each project module. This is still the default option.

Since module files could also be too large, with Kotlin 2.0.0, we add a more granular output that generates one (or two, if the file contains exported declarations) JavaScript file per each Kotlin file. To enable the per-file compilation mode:

1. Add the `useEsModules()` function to your build file to support ECMAScript modules:

   ```
   // build.gradle.kts
   kotlin {
       js(IR) {
           useEsModules() // Enables ES2015 modules
           browser()
       }
   }
   ```

   You can also use the new `es2015` compilation target for that.

2. Apply the `-Xir-per-file` compiler option or update your `gradle.properties` file with:

   ```
   # gradle.properties
   kotlin.js.ir.output.granularity=per-file // `per-module` is the default
   ```

Improved collection interoperability

Starting with Kotlin 2.0.0, it's possible to export declarations with a Kotlin collection type inside the signature to JavaScript (and TypeScript). This applies to `Set`, `Map`, and `List` collection types and their mutable counterparts.

To use Kotlin collections in JavaScript, first mark the necessary declarations with `@JsExport` annotation:

```kotlin
// Kotlin
@JsExport
data class User(
    val name: String,
    val friends: List<User> = emptyList()
)

@JsExport
val me = User(
    name = "Me",
    friends = listOf(User(name = "Kodee"))
)
```

You can then consume them from JavaScript as regular JavaScript arrays:

```javascript
// JavaScript
import { User, me, KtList } from "my-module"

const allMyFriendNames = me.friends
    .asJsReadonlyArrayView()
    .map(x => x.name) // ['Kodee']
```

Unfortunately, creating Kotlin collections from JavaScript is still unavailable. We're planning to add this functionality in Kotlin 2.0.20.

Support for createInstance()

Starting with Kotlin 2.0.0, you can use the `createInstance()` function from the Kotlin/JS target. Previously, it was only available on the JVM.

This function from the KClass interface creates a new instance of the specified class, which is useful for getting the runtime reference to a Kotlin class.

Support for type-safe plain JavaScript objects

The `js-plain-objects` plugin is Experimental. It may be dropped or changed at any time. The `js-plain-objects` plugin **only** supports the K2 compiler.

To make it easier to work with JavaScript APIs, in Kotlin 2.0.0, we provide a new plugin: `js-plain-objects`, which you can use to create type-safe plain JavaScript objects. The plugin checks your code for any external interfaces that have a `@JsPlainObject` annotation and adds:

- An inline `invoke` operator function inside the companion object that you can use as a constructor.
- A `.copy()` function that you can use to create a copy of your object while adjusting some of its properties.

For example:

```
import kotlinx.js.JsPlainObject

@JsPlainObject
external interface User {
    var name: String
    val age: Int
    val email: String?
}

fun main() {
    // Creates a JavaScript object
    val user = User(name = "Name", age = 10)
    // Copies the object and adds an email
    val copy = user.copy(age = 11, email = "some@user.com")

    println(JSON.stringify(user))
    // { "name": "Name", "age": 10 }
    println(JSON.stringify(copy))
    // { "name": "Name", "age": 11, "email": "some@user.com" }
}
```

Any JavaScript objects created with this approach are safer because instead of only seeing errors at runtime, you can see them at compile time or even highlighted by your IDE.

Consider this example, which uses a `fetch()` function to interact with a JavaScript API using external interfaces to describe the shape of the JavaScript objects:

```
import kotlinx.js.JsPlainObject

@JsPlainObject
external interface FetchOptions {
    val body: String?
    val method: String
}

// A wrapper for Window.fetch
suspend fun fetch(url: String, options: FetchOptions? = null) = TODO("Add your custom behavior here")

// A compile-time error is triggered as "metod" is not recognized
// as method
fetch("https://google.com", options = FetchOptions(metod = "POST"))
// A compile-time error is triggered as method is required
fetch("https://google.com", options = FetchOptions(body = "SOME STRING"))
```

In comparison, if you use the `js()` function instead to create your JavaScript objects, errors are only found at runtime or aren't triggered at all:

```
suspend fun fetch(url: String, options: FetchOptions? = null) = TODO("Add your custom behavior here")

// No error is triggered. As "metod" is not recognized, the wrong method
// (GET) is used.
fetch("https://google.com", options = js("{ metod: 'POST' }"))

// By default, the GET method is used. A runtime error is triggered as
// body shouldn't be present.
fetch("https://google.com", options = js("{ body: 'SOME STRING' }"))
// TypeError: Window.fetch: HEAD or GET Request cannot have a body
```

To use the `js-plain-objects` plugin, add the following to your `build.gradle(.kts)` file:

Kotlin

```
plugins {
    kotlin("plugin.js-plain-objects") version "2.0.0"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.plugin.js-plain-objects" version "2.0.0"
}
```

Support for npm package manager

Previously, it was only possible for the Kotlin Multiplatform Gradle plugin to use Yarn as a package manager to download and install npm dependencies. From Kotlin 2.0.0, you can use npm as your package manager instead. Using npm as a package manager means that you have one less tool to manage during your setup.

For backward compatibility, Yarn is still the default package manager. To use npm as your package manager, set the following property in your `gradle.properties` file:

```
kotlin.js.yarn = false
```

Changes to compilation tasks

Previously, the `webpack` and `distributeResources` compilation tasks both targeted the same directories. Moreover, the `distribution` task declared the `dist` as its output directory as well. This resulted in overlapping outputs and produced a compilation warning.

So, starting with Kotlin 2.0.0, we've implemented the following changes:

- The `webpack` task now targets a separate folder.
- The `distributeResources` task has been completely removed.
- The `distribution` task now has the `Copy` type and targets the `dist` folder.

Discontinuing legacy Kotlin/JS JAR artifacts

Starting with Kotlin 2.0.0, the Kotlin distribution no longer contains legacy Kotlin/JS artifacts with the `.jar` extension. Legacy artifacts were used in the unsupported old Kotlin/JS compiler and unnecessary for the IR compiler, which uses the `klib` format.

Gradle improvements

Kotlin 2.0.0 is fully compatible with Gradle 6.8.3 through 8.5. You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

This version brings the following changes:

- New Gradle DSL for compiler options in multiplatform projects

- New Compose compiler Gradle plugin
- New attribute to distinguish JVM and Android published libraries
- Improved Gradle dependency handling for CInteropProcess in Kotlin/Native
- Visibility changes in Gradle
- New directory for Kotlin data in Gradle projects
- Kotlin/Native compiler downloaded when needed
- Deprecating old ways of defining compiler options
- Bumped minimum AGP supported version
- New Gradle property for trying the latest language version
- New JSON output format for build reports
- kapt configurations inherit annotation processors from superconfigurations
- Kotlin Gradle plugin no longer uses deprecated Gradle conventions

New Gradle DSL for compiler options in multiplatform projects

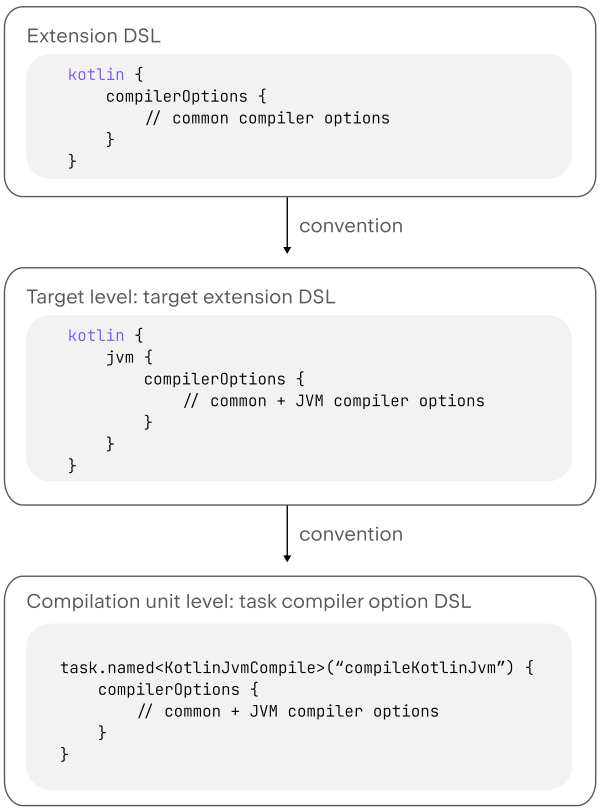
> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Prior to Kotlin 2.0.0, configuring compiler options in a multiplatform project with Gradle was only possible at a low level, such as per task, compilation, or source set. To make it easier to configure compiler options more generally in your projects, Kotlin 2.0.0 comes with a new Gradle DSL.

With this new DSL, you can configure compiler options at the extension level for all the targets and shared source sets like `commonMain` and at a target level for a specific target:

```
kotlin {
    compilerOptions {
        // Extension-level common compiler options that are used as defaults
        // for all targets and shared source sets
        allWarningsAsErrors.set(true)
    }
    jvm {
        compilerOptions {
            // Target-level JVM compiler options that are used as defaults
            // for all compilations in this target
            noJdk.set(true)
        }
    }
}
```

The overall project configuration now has three layers. The highest is the extension level, then the target level and the lowest is the compilation unit (which is usually a compilation task):



Kotlin compiler options levels

The settings at a higher level are used as a convention (default) for a lower level:

- The values of extension compiler options are the default for target compiler options, including shared source sets, like `commonMain`, `nativeMain`, and `commonTest`.
- The values of target compiler options are used as the default for compilation unit (task) compiler options, for example, `compileKotlinJvm` and `compileTestKotlinJvm` tasks.

In turn, configurations made at a lower level override related settings at a higher level:

- Task-level compiler options override related configurations at the target or the extension level.
- Target-level compiler options override related configurations at the extension level.

When configuring your project, keep in mind that some old ways of setting up compiler options have been deprecated.

We encourage you to try the new DSL out in your multiplatform projects and leave feedback in YouTrack, as we plan to make this DSL the recommended approach for configuring compiler options.

New Compose compiler Gradle plugin

The Jetpack Compose compiler, which translates composables into Kotlin code, has now been merged into the Kotlin repository. This will help transition Compose projects to Kotlin 2.0.0, as the Compose compiler will always ship simultaneously with Kotlin. This also bumps the Compose compiler version to 2.0.0.

To use the new Compose compiler in your projects, apply the `org.jetbrains.kotlin.plugin.compose` Gradle plugin in your `build.gradle(.kts)` file and set its version equal to Kotlin 2.0.0.

To learn more about this change and see the migration instructions, see the Compose compiler documentation.

New attribute to distinguish JVM and Android-published libraries

Starting with Kotlin 2.0.0, the `org.gradle.jvm.environment` Gradle attribute is published by default with all Kotlin variants.

The attribute helps distinguish JVM and Android variants of Kotlin Multiplatform libraries. It indicates that a certain library variant is better suited for a certain JVM environment. The target environment could be "android", "standard-jvm", or "no-jvm".

Publishing this attribute should make consuming Kotlin Multiplatform libraries with JVM and Android targets more robust from non-multiplatform clients as well, such as Java-only projects.

If necessary, you can disable attribute publication. To do that, add the following Gradle option to your `gradle.properties` file:

```
kotlin.publishJvmEnvironmentAttribute=false
```

Improved Gradle dependency handling for CInteropProcess in Kotlin/Native

In this release, we enhanced the handling of the `defFile` property to ensure better Gradle task dependency management in Kotlin/Native projects.

Before this update, Gradle builds could fail if the `defFile` property was designated as an output of another task that hadn't been executed yet. The workaround for this issue was to add a dependency on this task:

```
kotlin {
    macosArm64("native") {
        compilations.getByName("main") {
            cinterops {
                val cinterop by creating {
                    defFileProperty.set(createDefFileTask.flatMap { it.defFile.asFile })
                    project.tasks.named(interopProcessingTaskName).configure {
                        dependsOn(createDefFileTask)
                    }
                }
            }
        }
    }
}
```

To fix this, there is a new `RegularFileProperty` property called `definitionFile`. Now, Gradle lazily verifies the presence of the `definitionFile` property after the connected task has run later in the build process. This new approach eliminates the need for additional dependencies.

The `CInteropProcess` task and the `CInteropSettings` class use the `definitionFile` property instead of `defFile` and `defFileProperty`:

Kotlin

```
kotlin {
    macosArm64("native") {
        compilations.getByName("main") {
            cinterops {
                val cinterop by creating {
                    definitionFile.set(project.file("def-file.def"))
                }
            }
        }
    }
}
```

Groovy

```
kotlin {
    macosArm64("native") {
        compilations.main {
            cinterops {
                cinterop {
                    definitionFile.set(project.file("def-file.def"))
                }
            }
        }
    }
}
```

`defFile` and `defFileProperty` parameters are deprecated.

Visibility changes in Gradle

This change impacts only Kotlin DSL users.

In Kotlin 2.0.0, we've modified the Kotlin Gradle Plugin for better control and safety in your build scripts. Previously, certain Kotlin DSL functions and properties intended for a specific DSL context would inadvertently leak into other DSL contexts. This leakage could lead to the use of incorrect compiler options, settings being applied multiple times, and other misconfigurations:

```
kotlin {
    // Target DSL couldn't access methods and properties defined in the
    // kotlin{} extension DSL
    jvm {
        // Compilation DSL couldn't access methods and properties defined
        // in the kotlin{} extension DSL and Kotlin jvm{} target DSL
        compilations.configureEach {
            // Compilation task DSLs couldn't access methods and
            // properties defined in the kotlin{} extension, Kotlin jvm{}
            // target or Kotlin compilation DSL
            compileTaskProvider.configure {
                // For example:
                explicitApi()
                // ERROR as it is defined in the kotlin{} extension DSL
                mavenPublication {}
                // ERROR as it is defined in the Kotlin jvm{} target DSL
                defaultSourceSet {}
                // ERROR as it is defined in the Kotlin compilation DSL
            }
        }
    }
}
```

To fix this issue, we've added the `@KotlinGradlePluginDsl` annotation, preventing the exposure of the Kotlin Gradle plugin DSL functions and properties to levels where they are not intended to be available. The following levels are separated from each other:

- Kotlin extension
- Kotlin target
- Kotlin compilation
- Kotlin compilation task

For the most popular cases, we've added compiler warnings with suggestions on how to fix them if your build script is configured incorrectly. For example:

```
kotlin {
    jvm {
        sourceSets.getByName("jvmMain").dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.7.3")
        }
    }
}
```

In this case, the warning message for `sourceSets` is:

```
[DEPRECATION] 'sourceSets: NamedDomainObjectContainer<KotlinSourceSet>' is deprecated.Accessing
'sourceSets' container on the Kotlin target level DSL is deprecated. Consider configuring 'sourceSets'
on the Kotlin extension level.
```

We would appreciate your feedback on this change! Share your comments directly to Kotlin developers in our #gradle Slack channel. Get a Slack invite.

New directory for Kotlin data in Gradle projects

> Do not commit the `.kotlin` directory to version control. For example, if you are using Git, add `.kotlin` to your project's `.gitignore` file.

In Kotlin 1.8.20, the Kotlin Gradle plugin switched to storing its data in the Gradle project cache directory: `<project-root-directory>/.gradle/kotlin`. However, the `.gradle` directory is reserved for Gradle only, and as a result it's not future-proof.

To solve this, as of Kotlin 2.0.0, we will store Kotlin data in your `<project-root-directory>/.kotlin` by default. We will continue to store some data in the `.gradle/kotlin` directory for backward compatibility.

The new Gradle properties you can configure are:

Gradle property	Description
`kotlin.project.persistent.dir`	Configures the location where your project-level data is stored. Default: `<project-root-directory>/.kotlin`
`kotlin.project.persistent.dir.gradle.disableWrite`	A boolean value that controls whether writing Kotlin data to the `.gradle` directory is disabled. Default: `false`

Add these properties to the `gradle.properties` file in your projects for them to take effect.

Kotlin/Native compiler downloaded when needed

Before Kotlin 2.0.0, if you had a Kotlin/Native target configured in the Gradle build script of your multiplatform project, Gradle would always download the Kotlin/Native compiler in the configuration phase.

This happened even if there was no task to compile code for a Kotlin/Native target that was due to run in the execution phase. Downloading the Kotlin/Native compiler in this way was particularly inefficient for users who only wanted to check the JVM or JavaScript code in their projects. For example, to perform tests or checks with their Kotlin project as part of a CI process.

In Kotlin 2.0.0, we changed this behavior in the Kotlin Gradle plugin so that the Kotlin/Native compiler is downloaded in the execution phase and **only** when a compilation is requested for a Kotlin/Native target.

In turn, the Kotlin/Native compiler's dependencies are now downloaded not as a part of the compiler, but in the execution phase as well.

If you encounter any issues with the new behavior, you can temporarily switch back to the previous behavior by adding the following Gradle property to your `gradle.properties` file:

```
kotlin.native.toolchain.enabled=false
```

Starting with Kotlin 1.9.20-Beta, the Kotlin/Native distribution is published to Maven Central along with the CDN.

This allowed us to change how Kotlin looks for and downloads the necessary artifacts. Instead of the CDN, by default, it now uses the Maven repositories that you specified in the `repositories {}` block of your project.

You can temporarily switch this behavior back by setting the following Gradle property in your `gradle.properties` file:

```
kotlin.native.distribution.downloadFromMaven=false
```

Please report any problems to our issue tracker YouTrack. Both of these Gradle properties that change the default behavior are temporary and will be removed in future releases.

Deprecated old ways of defining compiler options

In this release, we continue to refine how you can set up compiler options. It should resolve ambiguity between different ways and make the project configuration more straightforward.

Since Kotlin 2.0.0, the following DSLs for specifying compiler options are deprecated:

- The `kotlinOptions` DSL from the `KotlinCompile` interface that implements all Kotlin compilation tasks. Use `KotlinCompilationTask<CompilerOptions>` instead.
- The `compilerOptions` property with the `HasCompilerOptions` type from the `KotlinCompilation` interface. This DSL was inconsistent with other DSLs and configured the same `KotlinCommonCompilerOptions` object as `compilerOptions` inside the `KotlinCompilation.compileTaskProvider` compilation task, which was confusing.

  Instead, we recommend using the `compilerOptions` property from the Kotlin compilation task:

  ```
  kotlinCompilation.compileTaskProvider.configure {
      compilerOptions { ... }
  }
  ```

  For example:

  ```
  kotlin {
      js(IR) {
          compilations.all {
              compileTaskProvider.configure {
                  compilerOptions.freeCompilerArgs.add("-Xir-minimized-member-names=false")
              }
          }
      }
  }
  ```

- The `kotlinOptions` DSL from the `KotlinCompilation` interface.
- The `kotlinOptions` DSL from the `KotlinNativeArtifactConfig` interface, the `KotlinNativeLink` class, and the `KotlinNativeLinkArtifactTask` class. Use the `toolOptions` DSL instead.
- The `dceOptions` DSL from the `KotlinJsDce` interface. Use the `toolOptions` DSL instead.

For more information on how to specify compiler options in the Kotlin Gradle plugin, see How to define options.

Bumped minimum supported AGP version

Starting with Kotlin 2.0.0, the minimum supported Android Gradle plugin version is 7.1.3.

New Gradle property for trying the latest language version

Prior to Kotlin 2.0.0, we had the following Gradle property to try out the new K2 compiler: `kotlin.experimental.tryK2`. Now that the K2 compiler is enabled by default in Kotlin 2.0.0, we decided to evolve this property into a new form that you can use to try the latest language version in your projects:

`kotlin.experimental.tryNext`. When you use this property in your `gradle.properties` file, the Kotlin Gradle plugin increments the language version to one above the default value for your Kotlin version. For example, in Kotlin 2.0.0, the default language version is 2.0, so the property configures language version 2.1.

This new Gradle property produces similar metrics in build reports as before with `kotlin.experimental.tryK2`. The language version configured is included in the output. For example:

```
##### 'kotlin.experimental.tryNext' results #####
:app:compileKotlin: 2.1 language version
:lib:compileKotlin: 2.1 language version
##### 100% (2/2) tasks have been compiled with Kotlin 2.1 #####
```

To learn more about how to enable build reports and their content, see Build reports.

New JSON output format for build reports

In Kotlin 1.7.0, we introduced build reports to help track compiler performance. Over time, we've added more metrics to make these reports even more detailed and helpful when investigating performance issues. Previously, the only output format for a local file was the `*.txt` format. In Kotlin 2.0.0, we support the JSON output format to make it even easier to analyze using other tools.

To configure JSON output format for your build reports, declare the following properties in your `gradle.properties` file:

```
kotlin.build.report.output=json

// The directory to store your build reports
kotlin.build.report.json.directory=my/directory/path
```

Alternatively, you can run the following command:

```
./gradlew assemble -Pkotlin.build.report.output=json -
Pkotlin.build.report.json.directory="my/directory/path"
```

Once configured, Gradle generates your build reports in the directory that you specify with the name: `${project_name}-date-time-<sequence_number>.json`.

Here's an example snippet from a build report with JSON output format that contains build metrics and aggregated metrics:

```
"buildOperationRecord": [
    {
     "path": ":lib:compileKotlin",
      "classFqName": "org.jetbrains.kotlin.gradle.tasks.KotlinCompile_Decorated",
      "startTimeMs": 1714730820601,
      "totalTimeMs": 2724,
      "buildMetrics": {
        "buildTimes": {
          "buildTimesNs": {
            "CLEAR_OUTPUT": 713417,
            "SHRINK_AND_SAVE_CURRENT_CLASSPATH_SNAPSHOT_AFTER_COMPILATION": 19699333,
            "IR_TRANSLATION": 281000000,
            "NON_INCREMENTAL_LOAD_CURRENT_CLASSPATH_SNAPSHOT": 14088042,
            "CALCULATE_OUTPUT_SIZE": 1301500,
            "GRADLE_TASK": 2724000000,
            "COMPILER_INITIALIZATION": 263000000,
            "IR_GENERATION": 74000000,
...
          }
        }
...
 "aggregatedMetrics": {
    "buildTimes": {
      "buildTimesNs": {
        "CLEAR_OUTPUT": 782667,
        "SHRINK_AND_SAVE_CURRENT_CLASSPATH_SNAPSHOT_AFTER_COMPILATION": 22031833,
        "IR_TRANSLATION": 333000000,
        "NON_INCREMENTAL_LOAD_CURRENT_CLASSPATH_SNAPSHOT": 14890292,
        "CALCULATE_OUTPUT_SIZE": 2370750,
        "GRADLE_TASK": 3234000000,
        "COMPILER_INITIALIZATION": 292000000,
        "IR_GENERATION": 89000000,
...
      }
    }
```


kapt configurations inherit annotation processors from superconfigurations

Prior to Kotlin 2.0.0, if you wanted to define a common set of annotation processors in a separate Gradle configuration and extend this configuration in kapt-specific configurations for your subprojects, kapt would skip annotation processing because it couldn't find any annotation processors. In Kotlin 2.0.0, kapt can successfully detect that there are indirect dependencies on your annotation processors.

As an example, for a subproject using Dagger, in your `build.gradle(.kts)` file, use the following configuration:

```
val commonAnnotationProcessors by configurations.creating
configurations.named("kapt") { extendsFrom(commonAnnotationProcessors) }

dependencies {
    implementation("com.google.dagger:dagger:2.48.1")
    commonAnnotationProcessors("com.google.dagger:dagger-compiler:2.48.1")
}
```

In this example, the `commonAnnotationProcessors` Gradle configuration is your common configuration for annotation processing that you want to be used for all your projects. You use the `extendsFrom()` method to add `commonAnnotationProcessors` as a superconfiguration. kapt sees that the `commonAnnotationProcessors` Gradle configuration has a dependency on the Dagger annotation processor. Therefore, kapt includes the Dagger annotation processor in its configuration for annotation processing.

Thanks to Christoph Loy for the implementation!

Kotlin Gradle plugin no longer uses deprecated Gradle conventions

Prior to Kotlin 2.0.0, if you used Gradle 8.2 or higher, the Kotlin Gradle plugin incorrectly used Gradle conventions that had been deprecated in Gradle 8.2. This led to Gradle reporting build deprecations. In Kotlin 2.0.0, the Kotlin Gradle plugin has been updated to no longer trigger these deprecation warnings when you use Gradle 8.2 or higher.

Standard library

This release brings further stability to the Kotlin standard library and makes even more existing functions common for all platforms:

- Stable replacement of the enum class values generic function
- Stable AutoCloseable interface
- Common protected property AbstractMutableList.modCount
- Common protected function AbstractMutableList.removeRange
- Common String.toCharArray(destination)

Stable replacement of the enum class values generic function

In Kotlin 2.0.0, the `enumEntries<T>()` function becomes Stable. The `enumEntries<T>()` function is a replacement for the generic `enumValues<T>()` function. The new function returns a list of all enum entries for the given enum type `T`. The `entries` property for enum classes was previously introduced and also stabilized to replace the synthetic `values()` function. For more information about the entries property, see What's new in Kotlin 1.8.20.

> The `enumValues<T>()` function is still supported, but we recommend that you use the `enumEntries<T>()` function instead because it has less of a performance impact. Every time you call `enumValues<T>()`, a new array is created, whereas whenever you call `enumEntries<T>()`, the same list is returned each time, which is far more efficient.

For example:

```
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumEntries<T>().joinToString { it.name })
}

printAllValues<RGB>()
// RED, GREEN, BLUE
```

Stable AutoCloseable interface

In Kotlin 2.0.0, the common `AutoCloseable` interface becomes Stable. It allows you to easily close resources and includes a couple of useful functions:

- The `use()` extension function, which executes a given block function on the selected resource and then closes it down correctly, whether an exception is thrown or not.
- The `AutoCloseable()` constructor function, which creates instances of the `AutoCloseable` interface.

In the example below, we define the `XMLWriter` interface and assume that there is a resource that implements it. For example, this resource could be a class that opens a file, writes XML content, and then closes it:

```
interface XMLWriter {
    fun document(encoding: String, version: String, content: XMLWriter.() -> Unit)
    fun element(name: String, content: XMLWriter.() -> Unit)
    fun attribute(name: String, value: String)
    fun text(value: String)

    fun flushAndClose()
}

fun writeBooksTo(writer: XMLWriter) {
    val autoCloseable = AutoCloseable { writer.flushAndClose() }
    autoCloseable.use {
        writer.document(encoding = "UTF-8", version = "1.0") {
            element("bookstore") {
                element("book") {
                    attribute("category", "fiction")
                    element("title") { text("Harry Potter and the Prisoner of Azkaban") }
                    element("author") { text("J. K. Rowling") }
                    element("year") { text("1999") }
                    element("price") { text("29.99") }
                }
                element("book") {
                    attribute("category", "programming")
                    element("title") { text("Kotlin in Action") }
                    element("author") { text("Dmitry Jemerov") }
                    element("author") { text("Svetlana Isakova") }
                    element("year") { text("2017") }
                    element("price") { text("25.19") }
                }
            }
        }
    }
}
```

Common protected property AbstractMutableList.modCount

In this release, the `modCount` `protected` property of the `AbstractMutableList` interface becomes common. Previously, the `modCount` property was available on each platform but not for the common target. Now, you can create custom implementations of `AbstractMutableList` and access the property in common code.

The property keeps track of the number of structural modifications made to the collection. This includes operations that change the collection size or alter the list in a way that may cause iterations in progress to return incorrect results.

You can use the `modCount` property to register and detect concurrent modifications when implementing a custom list.

Common protected function AbstractMutableList.removeRange

In this release, the `removeRange()` `protected` function of the `AbstractMutableList` interface becomes common. Previously, it was available on each platform but not for the common target. Now, you can create custom implementations of `AbstractMutableList` and override the function in common code.

The function removes elements from this list following the specified range. By overriding this function, you can take advantage of the custom implementations and improve the performance of the list operation.

Common String.toCharArray(destination) function

This release introduces a common `String.toCharArray(destination)` function. Previously, it was only available on the JVM.

Let's compare it with the existing `String.toCharArray()` function. It creates a new `CharArray` that contains characters from the specified string. The new common `String.toCharArray(destination)` function, however, moves `String` characters into an existing destination `CharArray`. This is useful if you already have a buffer that you want to fill:

```
fun main() {
    val myString = "Kotlin is awesome!"
    val destinationArray = CharArray(myString.length)

    // Convert the string and store it in the destinationArray:
    myString.toCharArray(destinationArray)

    for (char in destinationArray) {
        print("$char ")
        // K o t l i n  i s  a w e s o m e !
    }
}
```

Install Kotlin 2.0.0

Starting from IntelliJ IDEA 2023.3 and Android Studio Iguana (2023.2.1) Canary 15, the Kotlin plugin is distributed as a bundled plugin included in your IDE. This means that you can't install the plugin from JetBrains Marketplace anymore.

To update to the new Kotlin version, change the Kotlin version to 2.0.0 in your build scripts.

Compatibility guide for Kotlin 2.0.x

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like updated changelogs or compiler warnings, this document provides a complete reference for migration from Kotlin 1.9 to Kotlin 2.0.

> The Kotlin K2 compiler is introduced as part of Kotlin 2.0. For information on the benefits of the new compiler, changes you might encounter during migration, and how to roll back to the previous compiler, see K2 compiler migration guide.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Deprecate use of a synthetic setter on a projected receiver

Issue: KT-54309

Component: Core language

Incompatible change type: source

Short summary: If you use the synthetic setter of a Java class to assign a type that conflicts with the class's projected type, an error is triggered.

Deprecation cycle:

- 1.8.20: report a warning when a synthetic property setter has a projected parameter type in contravariant position making call-site argument types incompatible
- 2.0.0: raise the warning to an error

Correct mangling when calling functions with inline class parameters that are overloaded in a Java subclass

Issue: KT-56545

Component: Core language

Incompatible change type: behavioral

Deprecation cycle:

- 2.0.0: use the correct mangling behavior in function invocations; to revert to the previous behaviour, use the `-XXLanguage:-MangleCallsToJavaMethodsWithValueClasses` compiler option.

Correct type approximation algorithm for contravariant captured types

Issue: KT-49404

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.8.20: report a warning on problematic calls
- 2.0.0: raise the warning to an error

Prohibit accessing property value before property initialization

Issue: KT-56408

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: report an error when a property is accessed before initialization in the affected contexts

Report error when there's ambiguity in imported classes with the same name

Issue: KT-57750

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: report an error when resolving a class name that is present in several packages imported with a star import

Generate Kotlin lambdas via invokedynamic and LambdaMetafactory by default

Issue: KT-45375

Component: Core language

Incompatible change type: behavioral

Deprecation cycle:

- 2.0.0: implement new behavior; lambdas are generated using `invokedynamic` and `LambdaMetafactory` by default

Forbid if condition with one branch when an expression is required

Issue: KT-57871

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: report an error in case the `if` condition has only one branch

Prohibit violation of self upper bounds by passing a star-projection of a generic type

Issue: KT-61718

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: report an error when self upper bounds are violated by passing a star-projection of a generic type

Approximate anonymous types in private inline functions return type

Issue: KT-54862

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.9.0: report a warning on private inline functions if the inferred return type contains an anonymous type
- 2.0.0: approximate return type of such private inline functions to a supertype

Change overload resolution behavior to prioritize local extension function calls over invoke conventions of local functional type properties

Issue: KT-37592

Component: Core language

Incompatible change type: behavioral

Deprecation cycle:

- 2.0.0: new overload resolution behavior; function calls are consistently prioritized over invoke conventions

Report error when an inherited member conflict occurs due to a change in a supertype from binary dependency

Issue: KT-51194

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.7.0: report a warning CONFLICTING_INHERITED_MEMBERS_WARNING on declarations where inherited member conflict has occurred in the supertype from binary dependency
- 2.0.0: raise the warning to an error: CONFLICTING_INHERITED_MEMBERS

Ignore @UnsafeVariance annotations on parameters in invariant types

Issue: KT-57609

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: implement new behavior; the `@UnsafeVariance` annotation is ignored when reporting errors about type mismatch in contravariant parameters

Change type for out-of-call references to a companion object's member

Issue: KT-54316

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.8.20: report a warning on a companion object function reference type inferred as an unbound reference
- 2.0.0: change the behavior so that companion object function references are inferred as bound references in all usage contexts

Prohibit exposure of anonymous types from private inline functions

Issue: KT-33917

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.3.0: report a warning on calls to own members of anonymous objects, returned from private inline functions
- 2.0.0: approximate return type of such private inline functions to a supertype and don't resolve calls to anonymous object members

Report error for an unsound smart cast after a while-loop break

Issue: KT-22379

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: implement new behavior; the old behavior can be restored by switching to language version 1.9

Report error when a variable of an intersection type is assigned a value that is not a subtype of that intersection type

Issue: KT-53752

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: report an error when a variable having an intersection type is assigned a value that is not a subtype of that intersection type

Require opt-in when an interface constructed with a SAM constructor contains a method that requires an opt-in

Issue: KT-52628

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.7.20: report a warning for `OptIn` usages via SAM constructor
- 2.0.0: raise the warning to an error for `OptIn` usages via SAM constructor (or keep reporting the warning if `OptIn` marker severity is a warning)

Prohibit upper bound violation in typealias constructors

Issue: KT-54066

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.8.0: introduce a warning for cases when the upper bound is violated in typealias constructors
- 2.0.0: raise the warning to an error in the K2 compiler

Make the real type of a destructuring variable consistent with the explicit type when specified

Issue: KT-57011

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: implement new behavior; the real type of destructuring variable is now consistent with the explicit type when specified

Require opt-in when calling a constructor that has parameter types with default values that require an opt-in

Issue: KT-55111

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.8.20: report a warning on constructor calls that has parameter types which require opt-in
- 2.0.0: raise the warning to an error (or keep reporting a warning if the `OptIn` marker severity is a warning)

Report ambiguity between a property and an enum entry with the same name at the same scope level

Issue: KT-52802

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.7.20: report a warning when the compiler resolves to a property instead of an enum entry at the same scope level
- 2.0.0: report ambiguity when the compiler encounters both a property and an enum entry with the same name at the same scope level in the K2 compiler (leaving the warning as is in the old compiler)

Change qualifier resolution behavior to prefer companion property over enum entry

Issue: KT-47310

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: implement new resolution behavior; companion property is preferred over enum entry

Resolve invoke call receiver type and the invoke function type as if written in desugared form

Issue: KT-58260

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: resolve invoke call receiver type and the invoke function type independently as if they were written in a desugared form

Prohibit exposing private class members through non-private inline functions

Issue: KT-55179

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.9.0: report the `PRIVATE_CLASS_MEMBER_FROM_INLINE_WARNING` warning when calling private class companion object member from internal inline functions
- 2.0.0: raise this warning to the `PRIVATE_CLASS_MEMBER_FROM_INLINE` error

Correct nullability of definitely non-null types in projected generic types

Issue: KT-54663

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: implement new behavior; projected types take into account all in-place not-null types

Change inferred type of prefix increment to match getter's return type instead of inc() operator's return type

Issue: KT-57178

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: implement new behavior; inferred type of prefix increment is changed to match getter's return type instead of the `inc()` operator's return type

Enforce bound checks when inheriting inner classes from generic inner classes declared in superclasses

Issue: KT-61749

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: report an error when upper bound of the type parameter of a generic inner superclass is violated

Forbid assigning callable references with SAM types when the expected type is a function type with a function type parameter

Issue: KT-64342

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: report a compilation error on callable references with SAM types when the expected type is a function type with a function type parameter

Consider companion object scope for annotation resolution on companion objects

Issue: KT-64299

Component: Core language

Incompatible change type: behavioral

Deprecation cycle:

- 2.0.0: implement new behavior; the companion object scope is now not ignored during annotation resolution on companion objects

Change evaluation semantics for combination of safe calls and convention operators

Issue: KT-41034

Component: Core language

Incompatible change type: behavioral

Deprecation cycle:

- 1.4.0: report warnings on each incorrect call
- 2.0.0: implement new resolution behavior

Require properties with backing field and a custom setter to be immediately initialized

Issue: KT-58589

Component: Core language

Incompatible change type: behavioral

Deprecation cycle:

- 1.9.20: introduce the `MUST_BE_INITIALIZED` warning for cases where there is no primary constructor
- 2.0.0: raise the warning to an error

Prohibit Unit conversion on arbitrary expressions in invoke operator convention call

Issue: KT-61182

Component: Core language

Incompatible change type: behavioral

Deprecation cycle:

- 2.0.0: report an error when Unit conversion is applied to an arbitrary expression on variable and invoke resolution; use the `-XXLanguage:+UnitConversionsOnArbitraryExpressions` compiler option to keep the previous behavior with affected expressions.

Forbid nullable assignment to non-null Java field when the field is accessed with a safe call

Issue: KT-62998

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: report to an error in case a nullable is assigned to a non-null Java field

Require star-projected type when overriding a Java method containing a raw-type parameter

Issue: KT-57600

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: implement new behavior; overriding is forbidden for raw type parameters

Change (V)::foo reference resolution when V has a companion

Issue: KT-47313

Component: Core language

Incompatible change type: behavioral

Deprecation cycle:

- 1.6.0: report a warning on callable references currently bound to companion object instances
- 2.0.0: implement new behavior; adding parentheses around a type no longer makes it a reference to the type's companion object instance

Forbid implicit non-public API access in effectively public inline functions

Issue: KT-54997

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.8.20: report a compilation warning when implicit non-public API is accessed in public inline functions
- 2.0.0: raise the warning to an error

Prohibit use-site get annotations on property getters

Issue: KT-57422

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.9.0: report a warning (an error in the progressive mode) on use-site `get` annotations on getters
- 2.0.0: raise the warning to the `INAPPLICABLE_TARGET_ON_PROPERTY` error; use `-XXLanguage:-ProhibitUseSiteGetTargetAnnotations` to revert to a warning

Prevent implicit inference of type parameters into upper bounds in builder inference lambda functions

Issue: KT-47986

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.7.20: report a warning (or an error in the progressive mode) when the type parameter for a type argument cannot be inferred into declared upper bounds
- 2.0.0: raise the warning to an error

Keep nullability when approximating local types in public signatures

Issue: KT-53982

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.8.0: flexible types are approximated by flexible supertypes; report a warning when a declaration is inferred to have a non-nullable type that should be nullable, prompting to specify the type explicitly to avoid NPEs
- 2.0.0: nullable types are approximated by nullable supertypes

Remove special handling for false && ... and false || ... for the purposes of smart-casting

Issue: KT-65776

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: implement new behavior; no special handling for `false && ...` and `false || ...`

Forbid inline open functions in enums

Issue: KT-34372

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 1.8.0: report a warning on inline open functions in enums
- 2.0.0: raise the warning to an error

Tools

Visibility changes in Gradle

Issue: KT-64653

Component: Gradle

Incompatible change type: source

Short summary: Previously, certain Kotlin DSL functions and properties intended for a specific DSL context would inadvertently leak into other DSL contexts. We've added the `@KotlinGradlePluginDsl` annotation, which prevents the exposure of the Kotlin Gradle plugin DSL functions and properties to levels where they are not intended to be available. The following levels are separated from each other:

- Kotlin extension
- Kotlin target
- Kotlin compilation
- Kotlin compilation task

Deprecation cycle:

- 2.0.0: for most popular cases, the compiler reports warnings with suggestions on how to fix them if your build script is configured incorrectly; otherwise, the compiler reports an error

Deprecate kotlinOptions DSL

Issue: KT-63419

Component: Gradle

Incompatible change type: source

Short summary: The ability to configure compiler options through the `kotlinOptions` DSL and the related `KotlinCompile<KotlinOptions>` task interface have been deprecated.

Deprecation cycle:

- 2.0.0: report a warning

Deprecate compilerOptions in KotlinCompilation DSL

Issue: KT-65568

Component: Gradle

Incompatible change type: source

Short summary: The ability to configure the `compilerOptions` property in the `KotlinCompilation` DSL has been deprecated.

Deprecation cycle:

- 2.0.0: report a warning

Deprecate old ways of CInteropProcess handling

Issue: KT-62795

Component: Gradle

Incompatible change type: source

Short summary: the `CInteropProcess` task and the `CInteropSettings` class now use the `definitionFile` property instead of `defFile` and `defFileProperty`.

This eliminates the need to add extra `dependsOn` relations between the `CInteropProcess` task and the task that generates `defFile` when the `defFile` is dynamically generated.

In Kotlin/Native projects, Gradle now lazily verifies the presence of the `definitionFile` property after the connected task has run later in the build process.

Deprecation cycle:

- 2.0.0: `defFile` and `defFileProperty` parameters are deprecated
- 2.4.0: report an error for the deprecated `defFile` property

Remove kotlin.useK2 Gradle property

Issue: KT-64379

Component: Gradle

Incompatible change type: behavioral

Short summary: The `kotlin.useK2` Gradle property has been removed. In Kotlin 1.9.*, it could be used to enable the K2 compiler. In Kotlin 2.0.0 and later, the K2 compiler is enabled by default, so the property has no effect and cannot be used to switch back to the previous compiler.

Deprecation cycle:

- 1.8.20: the `kotlin.useK2` Gradle property is deprecated
- 2.0.0: the `kotlin.useK2` Gradle property is removed

Remove deprecated platform plugin IDs

Issue: KT-65187

Component: Gradle

Incompatible change type: source

Short summary: support for these platform plugin IDs have been removed:

- `kotlin-platform-android`
- `kotlin-platform-jvm`
- `kotlin-platform-js`
- `org.jetbrains.kotlin.platform.android`
- `org.jetbrains.kotlin.platform.jvm`
- `org.jetbrains.kotlin.platform.js`

Deprecation cycle:

- 1.3: the platform plugin IDs are deprecated
- 2.0.0: the platform plugin IDs are no longer supported

Remove outputFile JavaScript compiler option

Issue: KT-61116

Component: Gradle

Incompatible change type: source

Short summary: The `outputFile` JavaScript compiler option has been removed. Instead, you can use the `destinationDirectory` property of the `Kotlin2JsCompile` task to specify the directory where the compiled JavaScript output files are written.

Deprecation cycle:

- 1.9.25: the `outputFile` compiler option is deprecated
- 2.0.0: the `outputFile` compiler option is removed

What's new in Kotlin 1.9.20

Released: November 1, 2023

The Kotlin 1.9.20 release is out, the K2 compiler for all the targets is now in Beta, and Kotlin Multiplatform is now Stable. Additionally, here are some of the main highlights:

- New default hierarchy template for setting up multiplatform projects
- Full support for the Gradle configuration cache in Kotlin Multiplatform
- Custom memory allocator enabled by default in Kotlin/Native
- Performance improvements for the garbage collector in Kotlin/Native
- New and renamed targets in Kotlin/Wasm
- Support for the WASI API in the standard library for Kotlin/Wasm

You can also find a short overview of the updates in this video:

Watch video online.

For information about the Kotlin release cycle, see Kotlin release process.

IDE support

The Kotlin plugins that support 1.9.20 are available for:

IDE	Supported versions
IntelliJ IDEA	2023.1.x, 2023.2.x, 2023.x
Android Studio	Hedgehog (2023.1.1), Iguana (2023.2.1)

Starting from IntelliJ IDEA 2023.3.x and Android Studio Iguana (2023.2.1) Canary 15, the Kotlin plugin is automatically included and updated. All you need to do is update the Kotlin version in your projects.

New Kotlin K2 compiler updates

The Kotlin team at JetBrains is continuing to stabilize the new K2 compiler, which will bring major performance improvements, speed up new language feature development, unify all the platforms that Kotlin supports, and provide a better architecture for multiplatform projects.

K2 is currently in **Beta** for all targets. Read more in the release blog post

Support for Kotlin/Wasm

Since this release, the Kotlin/Wasm supports the new K2 compiler. Learn how to enable it in your project.

Preview kapt compiler plugin with K2

Support for K2 in the kapt compiler plugin is Experimental. Opt-in is required (see details below), and you should use it only for evaluation purposes.

In 1.9.20, you can try using the kapt compiler plugin with the K2 compiler. To use the K2 compiler in your project, add the following options to your `gradle.properties` file:

```
kotlin.experimental.tryK2=true
kapt.use.k2=true
```

Alternatively, you can enable K2 for kapt by completing the following steps:

1. In your `build.gradle.kts` file, set the language version to `2.0`.
2. In your `gradle.properties` file, add `kapt.use.k2=true`.

If you encounter any issues when using kapt with the K2 compiler, please report them to our issue tracker.

How to enable the Kotlin K2 compiler

Enable K2 in Gradle

To enable and test the Kotlin K2 compiler, use the new language version with the following compiler option:

```
-language-version 2.0
```

You can specify it in your `build.gradle.kts` file:

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = "2.0"
        }
    }
}
```

Enable K2 in Maven

To enable and test the Kotlin K2 compiler, update the `<project/>` section of your `pom.xml` file:

```
<properties>
    <kotlin.compiler.languageVersion>2.0</kotlin.compiler.languageVersion>
</properties>
```

Enable K2 in IntelliJ IDEA

To enable and test the Kotlin K2 compiler in IntelliJ IDEA, go to **Settings** | **Build, Execution, Deployment** | **Compiler** | **Kotlin Compiler** and update the **Language Version** field to `2.0 (experimental)`.

Leave your feedback on the new K2 compiler

We would appreciate any feedback you may have!

- Provide your feedback directly to K2 developers on Kotlin Slack – get an invite and join the #k2-early-adopters channel.
- Report any problems you faced with the new K2 compiler on our issue tracker.
- Enable the Send usage statistics option to allow JetBrains to collect anonymous data about K2 usage.

Kotlin/JVM

Starting with version 1.9.20, the compiler can generate classes containing Java 21 bytecode.

Kotlin/Native

Kotlin 1.9.20 includes a Stable memory manager with the new memory allocator enabled by default, performance improvements for the garbage collector, and other updates:

- Custom memory allocator enabled by default
- Performance improvements for the garbage collector
- Incremental compilation of `klib` artifacts
- Managing library linkage issues
- Companion object initialization on class constructor calls
- Opt-in requirement for all cinterop declarations
- Custom message for linker errors
- Removal of the legacy memory manager
- Change to our target tiers policy

Custom memory allocator enabled by default

Kotlin 1.9.20 comes with the new memory allocator enabled by default. It's designed to replace the previous default allocator, `mimalloc`, to make garbage collection more efficient and improve the runtime performance of the Kotlin/Native memory manager.

The new custom allocator divides system memory into pages, allowing independent sweeping in consecutive order. Each allocation becomes a memory block within a page, and the page keeps track of block sizes. Different page types are optimized for various allocation sizes. The consecutive arrangement of memory blocks ensures efficient iteration through all allocated blocks.

When a thread allocates memory, it searches for a suitable page based on the allocation size. Threads maintain a set of pages for different size categories. Typically, the current page for a given size can accommodate the allocation. If not, the thread requests a different page from the shared allocation space. This page may already be available, require sweeping, or have to be created first.

The new allocator allows for multiple independent allocation spaces simultaneously, which will enable the Kotlin team to experiment with different page layouts to improve performance even further.

How to enable the custom memory allocator

Starting with Kotlin 1.9.20, the new memory allocator is the default. No additional setup is required.

If you experience high memory consumption, you can switch back to `mimalloc` or the system allocator with `-Xallocator=mimalloc` or `-Xallocator=std` in your Gradle build script. Please report such issues in YouTrack to help us improve the new memory allocator.

For the technical details of the new allocator's design, see this README.

Performance improvements for the garbage collector

The Kotlin team continues to improve the performance and stability of the new Kotlin/Native memory manager. This release brings a number of significant changes to the garbage collector (GC), including the following 1.9.20 highlights:

- Full parallel mark to reduce the pause time for the GC
- Tracking memory in big chunks to improve the allocation performance

Full parallel mark to reduce the pause time for the GC

Previously, the default garbage collector performed only a partial parallel mark. When the mutator thread was paused, it would mark the GC's start from its own roots, like thread–local variables and the call stack. Meanwhile, a separate GC thread was responsible for marking the start from global roots, as well as the roots of all mutators that were actively running the native code and therefore not paused.

This approach worked well in cases where there were a limited number of global objects and the mutator threads spent a considerable amount of time in a runnable state executing Kotlin code. However, this is not the case for typical iOS applications.

Now the GC uses a full parallel mark that combines paused mutators, the GC thread, and optional marker threads to process the mark queue. By default, the marking process is performed by:

- Paused mutators. Instead of processing their own roots and then being idle while not actively executing code, they contribute to the whole marking process.
- The GC thread. This ensures that at least one thread will perform marking.

This new approach makes the marking process more efficient, reducing the pause time of the GC.

Tracking memory in big chunks to improve the allocation performance

Previously, the GC scheduler tracked the allocation of each object individually. However, neither the new default custom allocator nor the `mimalloc` memory allocator allocates separate storage for each object; they allocate large areas for several objects at once.

In Kotlin 1.9.20, the GC tracks areas instead of individual objects. This speeds up the allocation of small objects by reducing the number of tasks performed on each allocation and, therefore, helps to minimize the garbage collector's memory usage.

Incremental compilation of klib artifacts

> This feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 1.9.20 introduces a new compilation time optimization for Kotlin/Native. The compilation of `klib` artifacts into native code is now partially incremental.

When compiling Kotlin source code into native binary in debug mode, the compilation goes through two stages:

1. Source code is compiled into `klib` artifacts.
2. `klib` artifacts, along with dependencies, are compiled into a binary.

To optimize the compilation time in the second stage, the team has already implemented compiler caches for dependencies. They are compiled into native code only once, and the result is reused every time a binary is compiled. But `klib` artifacts built from project sources were always fully recompiled into native code at every project change.

With the new incremental compilation, if the project module change causes only a partial recompilation of source code into `klib` artifacts, just a part of the `klib` is further recompiled into a binary.

To enable incremental compilation, add the following option to your `gradle.properties` file:

```
kotlin.incremental.native=true
```

If you face any issues, report such cases to YouTrack.

Managing library linkage issues

This release improves the way the Kotlin/Native compiler handles linkage issues in Kotlin libraries. Error messages now include more readable declarations as they use signature names instead of hashes, helping you find and fix the issue more easily. Here's an example:

```
No function found for symbol
'org.samples/MyClass.removedFunction|removedFunction(kotlin.Int;kotlin.String){}[0]'
```

The Kotlin/Native compiler detects linkage issues between third-party Kotlin libraries and reports errors at runtime. You might face such issues if the author of one third-party Kotlin library makes an incompatible change in experimental APIs that another third-party Kotlin library consumes.

Starting with Kotlin 1.9.20, the compiler detects linkage issues in silent mode by default. You can adjust this setting in your projects:

- If you want to record these issues in your compilation logs, enable warnings with the `-Xpartial-linkage-loglevel=WARNING` compiler option.
- It's also possible to raise the severity of reported warnings to compilation errors with `-Xpartial-linkage-loglevel=ERROR`. In this case, the compilation fails, and you get all the errors in the compilation log. Use this option to examine the linkage issues more closely.

```
// An example of passing compiler options in a Gradle build file:
kotlin {
    macosX64("native") {
        binaries.executable()

        compilations.configureEach {
            compilerOptions.configure {
                // To report linkage issues as warnings:
                freeCompilerArgs.add("-Xpartial-linkage-loglevel=WARNING")

                // To raise linkage warnings to errors:
                freeCompilerArgs.add("-Xpartial-linkage-loglevel=ERROR")
            }
        }
    }
}
```

If you face unexpected problems with this feature, you can always opt out with the `-Xpartial-linkage=disable` compiler option. Don't hesitate to report such cases to our issue tracker.

Companion object initialization on class constructor calls

Starting with Kotlin 1.9.20, the Kotlin/Native backend calls static initializers for companion objects in class constructors:

```
class Greeting {
    companion object {
        init {
            print("Hello, Kotlin!")
        }
    }
}

fun main() {
    val start = Greeting() // Prints "Hello, Kotlin!"
}
```

The behavior is now unified with Kotlin/JVM, where a companion object is initialized when the corresponding class matching the semantics of a Java static initializer is loaded (resolved).

Now that the implementation of this feature is more consistent between platforms, it's easier to share code in Kotlin Multiplatform projects.

Opt-in requirement for all cinterop declarations

Starting with Kotlin 1.9.20, all Kotlin declarations generated by the `cinterop` tool from C and Objective-C libraries, like libcurl and libxml, are marked with `@ExperimentalForeignApi`. If the opt-in annotation is missing, your code won't compile.

This requirement reflects the Experimental status of the import of C and Objective-C libraries. We recommend that you confine its use to specific areas in your projects. This will make your migration easier once we begin stabilizing the import.

> As for native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX), only some of their APIs need an opt-in with `@ExperimentalForeignApi`. In such cases, you get a warning with an opt-in requirement.

Custom message for linker errors

If you're a library author, you can now help your users resolve linker errors with custom messages.

If your Kotlin library depends on C or Objective-C libraries, for example, using the CocoaPods integration, its users need to have these dependent libraries locally on the machine or configure them explicitly in the project build script. If this was not the case, users used to get a confusing "Framework not found" message.

You can now provide a specific instruction or a link in the compilation failure message. To do that, pass the `-Xuser-setup-hint` compiler option to `cinterop` or add a `userSetupHint=message` property to your `.def` file.

Removal of the legacy memory manager

The new memory manager was introduced in Kotlin 1.6.20 and became the default in 1.7.20. Since then, it has been receiving further updates and performance improvements and has become Stable.

The time has come to complete the deprecation cycle and remove the legacy memory manager. If you're still using it, remove the `kotlin.native.binary.memoryModel=strict` option from your `gradle.properties` and follow our Migration guide to make the necessary changes.

Change to our target tiers policy

We've decided to upgrade the requirements for tier 1 support. The Kotlin team is now committed to providing source and binary compatibility between compiler releases for targets eligible for tier 1. They must also be regularly tested with CI tools to be able to compile and run. Currently, tier 1 includes the following targets for macOS hosts:

- `macosX64`
- `macosArm64`
- `iosSimulatorArm64`
- `iosX64`

In Kotlin 1.9.20, we've also removed a number of previously deprecated targets, namely:

- `iosArm32`
- `watchosX86`
- `wasm32`
- `mingwX86`
- `linuxMips32`
- `linuxMipsel32`

See the full list of currently supported targets.

Kotlin Multiplatform

Kotlin 1.9.20 focuses on the stabilization of Kotlin Multiplatform and makes new steps in improving developer experience with the new project wizards and other notable features:

- Kotlin Multiplatform is Stable
- Template for configuring multiplatform projects
- New project wizard
- Full support for the Gradle Configuration cache
- Easier configuration of new standard library versions in Gradle
- Default support for third-party cinterop libraries
- Support for Kotlin/Native compilation caches in Compose Multiplatform projects
- Compatibility guidelines

Kotlin Multiplatform is Stable

The 1.9.20 release marks an important milestone in the evolution of Kotlin: Kotlin Multiplatform has finally become Stable. This means that the technology is safe to use in your projects and 100% ready for production. It also means that further development of Kotlin Multiplatform will continue according to our strict backward compatibility rules.

Please note that some advanced features of Kotlin Multiplatform are still evolving. When using them, you'll receive a warning that describes the current stability status of the feature you're using. Before using any experimental functionality in IntelliJ IDEA,
you'll need to enable it explicitly in **Settings** | **Advanced Settings** | **Kotlin** | **Experimental Multiplatform**.

- Visit the Kotlin blog to learn more about the Kotlin Multiplatform stabilization and future plans.
- Check out the Multiplatform compatibility guide to see what significant changes were made on the way to stabilization.
- Read about the mechanism of expected and actual declarations, an important part of Kotlin Multiplatform that was also partially stabilized in this release.

Template for configuring multiplatform projects

Starting with Kotlin 1.9.20, the Kotlin Gradle plugin automatically creates shared source sets for popular multiplatform scenarios. If your project setup is one of them, you don't need to configure the source set hierarchy manually. Just explicitly specify the targets necessary for your project.

Setup is now easier thanks to the default hierarchy template, a new feature of the Kotlin Gradle plugin. It's a predefined template of a source set hierarchy built into the plugin. It includes intermediate source sets that Kotlin automatically creates for the targets you declared. See the full template.

Create your project easier

Consider a multiplatform project that targets both Android and iPhone devices and is developed on an Apple silicon MacBook. Compare how this project is set up between different versions of Kotlin:

Kotlin 1.9.0 and earlier (a standard setup)

```
kotlin {
    androidTarget()
    iosArm64()
    iosSimulatorArm64()

    sourceSets {
        val commonMain by getting

        val iosMain by creating {
            dependsOn(commonMain)
        }

        val iosArm64Main by getting {
            dependsOn(iosMain)
        }

        val iosSimulatorArm64Main by getting
{
            dependsOn(iosMain)
        }
    }
}
```

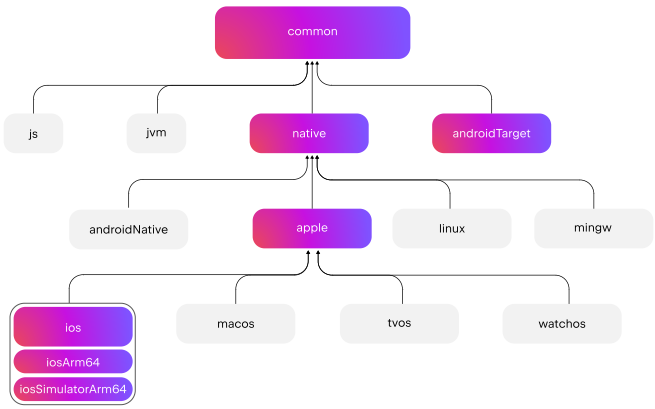
Kotlin 1.9.20

```
kotlin {
    androidTarget()
    iosArm64()
    iosSimulatorArm64()

    // The iosMain source set is created
automatically
}
```

Notice how the use of the default hierarchy template considerably reduces the amount of boilerplate code needed to set up your project.

When you declare the `androidTarget`, `iosArm64`, and `iosSimulatorArm64` targets in your code, the Kotlin Gradle plugin finds suitable shared source sets from the template and creates them for you. The resulting hierarchy looks like this:



An example of the default target hierarchy in use

Green source sets are actually created and included in the project, while gray ones from the default template are ignored.

Use completion for source sets

To make it easier to work with the created project structure, IntelliJ IDEA now provides completion for source sets created with the default hierarchy template:



Kotlin also warns you if you attempt to access a source set that doesn't exist because you haven't declared the respective target. In the example below, there is no JVM target (only `androidTarget`, which is not the same). But let's try to use the `jvmMain` source set and see what happens:

```
kotlin {
    androidTarget()
    iosArm64()
    iosSimulatorArm64()

    sourceSets {
        jvmMain {
        }
    }
}
```

In this case, Kotlin reports a warning in the build log:

```
w: Accessed 'source set jvmMain' without registering the jvm target:
  kotlin {
      jvm() /* ← register the 'jvm' target */

      sourceSets.jvmMain.dependencies {

      }
  }
```

Set up the target hierarchy

Starting with Kotlin 1.9.20, the default hierarchy template is automatically enabled. In most cases, no additional setup is required.

However, if you're migrating existing projects created before 1.9.20, you might encounter a warning if you had previously introduced intermediate sources manually with `dependsOn()` calls. To solve this issue, do the following:

- If your intermediate source sets are currently covered by the default hierarchy template, remove all manual `dependsOn()` calls and source sets created with `by creating` constructions.

  To check the list of all default source sets, see the full hierarchy template.

- If you want to have additional source sets that the default hierarchy template doesn't provide, for example, one that shares code between a macOS and a JVM target, adjust the hierarchy by reapplying the template explicitly with `applyDefaultHierarchyTemplate()` and configuring additional source sets manually as usual with `dependsOn()`:

  ```
  kotlin {
      jvm()
      macosArm64()
      iosArm64()
      iosSimulatorArm64()

      // Apply the default hierarchy explicitly. It'll create, for example, the iosMain source set:
      applyDefaultHierarchyTemplate()

      sourceSets {
          // Create an additional jvmAndMacos source set
          val jvmAndMacos by creating {
              dependsOn(commonMain.get())
          }

          macosArm64Main.get().dependsOn(jvmAndMacos)
          jvmMain.get().dependsOn(jvmAndMacos)
      }
  }
  ```

- If there are already source sets in your project that have the exact same names as those generated by the template but that are shared among different sets of targets, there's currently no way to modify the default `dependsOn` relations between the template's source sets.

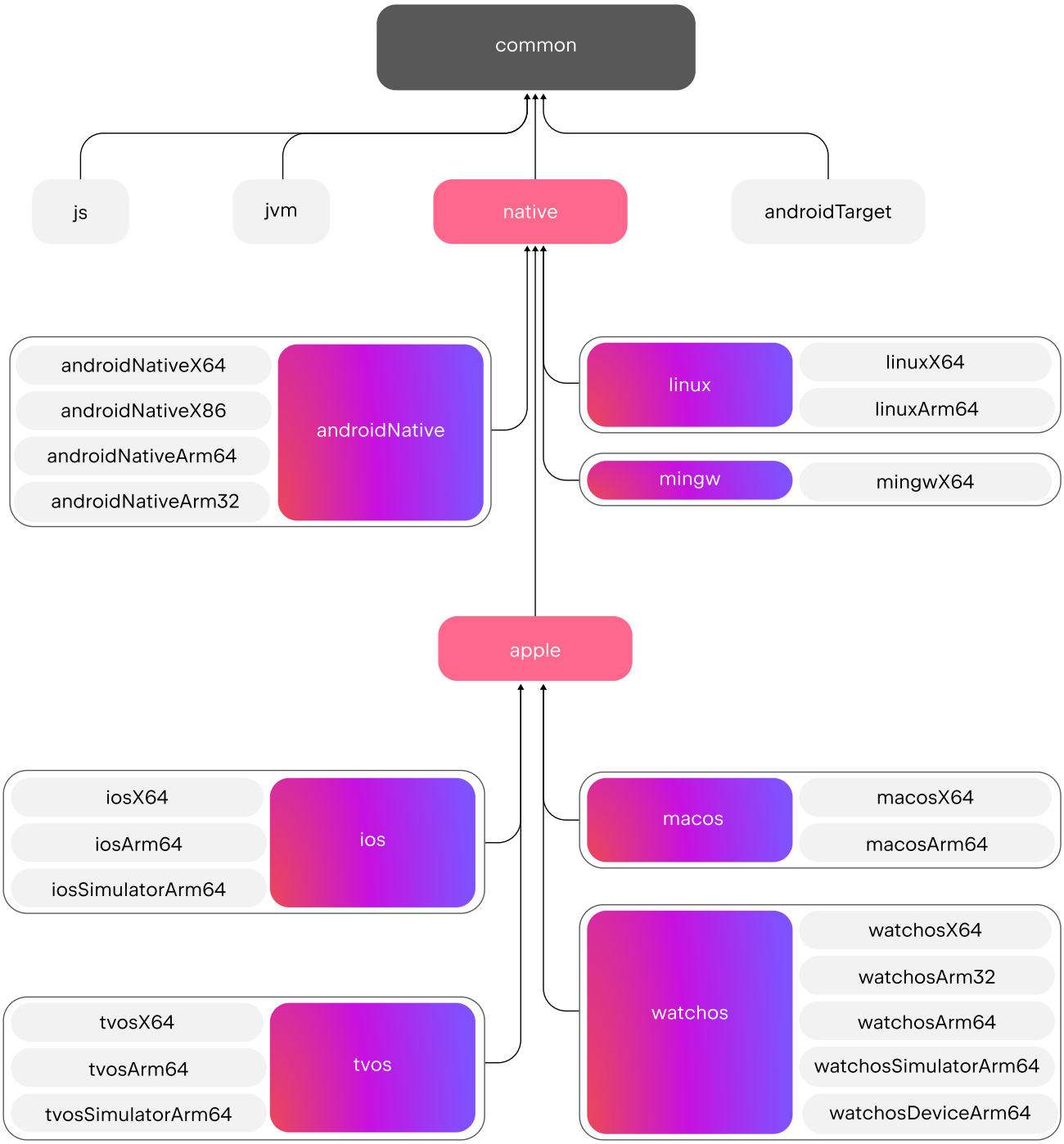
  One option you have here is to find different source sets for your purposes, either in the default hierarchy template or ones that have been manually created. Another is to opt out of the template completely.

  To opt out, add `kotlin.mpp.applyDefaultHierarchyTemplate=false` to your `gradle.properties` and configure all other source sets manually.

  We're currently working on an API for creating your own hierarchy templates to simplify the setup process in such cases.

See the full hierarchy template

When you declare the targets to which your project compiles, the plugin picks the shared source sets from the template accordingly and creates them in your project.



Default hierarchy template

> This example only shows the production part of the project, omitting the `Main` suffix (for example, using `common` instead of `commonMain`). However, everything is the same for `*Test` sources as well.

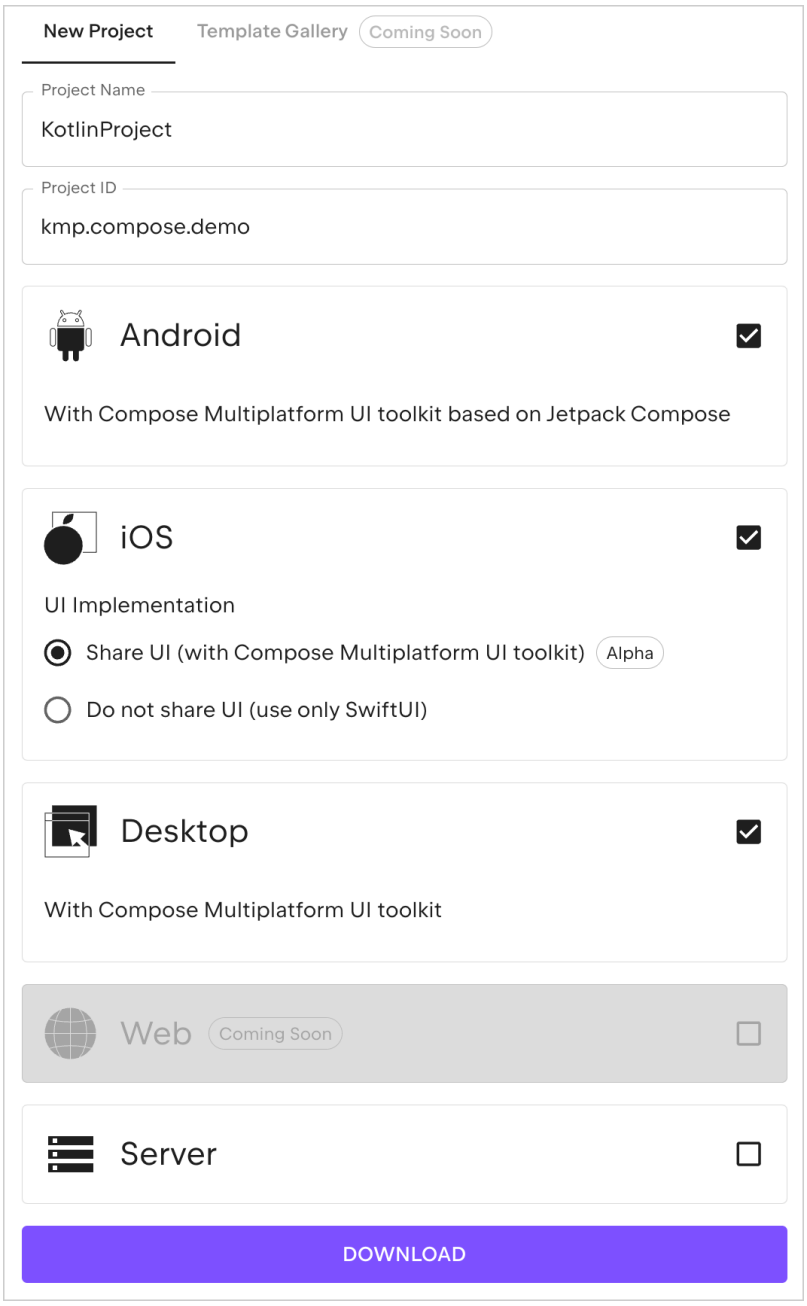
New project wizard

The JetBrains team is introducing a new way of creating cross-platform projects – the Kotlin Multiplatform web wizard.

This first implementation of the new Kotlin Multiplatform wizard covers the most popular Kotlin Multiplatform use cases. It incorporates all the feedback about previous project templates and makes the architecture as robust and reliable as possible.

The new wizard has a distributed architecture that allows us to have a unified backend and different frontends, with the web version being the first step. We're considering both implementing an IDE version and creating a command-line tool in the future. On the web, you always get the latest version of the wizard, while in IDEs you'll need to wait for the next release.

With the new wizard, project setup is easier than ever. You can tailor your projects to your needs by choosing the target platforms for mobile, server, and desktop development. We also plan to add web development in future releases.



Multiplatform web wizard

The new project wizard is now the preferred way to create cross-platform projects with Kotlin. Since 1.9.20, the Kotlin plugin no longer provides a **Kotlin Multiplatform** project wizard in IntelliJ IDEA.

The new wizard will guide you easily through the initial setup, making the onboarding process much smoother. If you encounter any issues, please report them to YouTrack to help us improve your experience with the wizard.



Create a project

Full support for the Gradle configuration cache in Kotlin Multiplatform

Previously, we introduced a preview of the Gradle configuration cache, which was available for Kotlin multiplatform libraries. With 1.9.20, the Kotlin Multiplatform plugin takes a step further.

It now supports the Gradle configuration cache in the Kotlin CocoaPods Gradle plugin, as well as in the integration tasks that are necessary for Xcode builds, like `embedAndSignAppleFrameworkForXcode`.

Now all multiplatform projects can take advantage of the improved build time. The Gradle configuration cache speeds up the build process by reusing the results of the configuration phase for subsequent builds. For more details and setup instructions, see the Gradle documentation.

Easier configuration of new standard library versions in Gradle

When you create a multiplatform project, a dependency for the standard library (`stdlib`) is added automatically to each source set. This is the easiest way to get started with your multiplatform projects.

Previously, if you wanted to configure a dependency on the standard library manually, you needed to configure it for each source set individually. From `kotlin-stdlib:1.9.20` onward, you only need to configure the dependency **once** in the `commonMain` root source set:

Standard library version 1.9.10 and earlier

```
kotlin {
    sourceSets {
        // For the common source set
        val commonMain by getting {
            dependencies {

implementation("org.jetbrains.kotlin:kotlin-
stdlib-common:1.9.10")
            }
        }

        // For the JVM source set
        val jvmMain by getting {
            dependencies {

implementation("org.jetbrains.kotlin:kotlin-
stdlib:1.9.10")
            }
        }

        // For the JS source set
        val jsMain by getting {
            dependencies {

implementation("org.jetbrains.kotlin:kotlin-
stdlib-js:1.9.10")
            }
        }
    }
}
```

Standard library version 1.9.20

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {

implementation("org.jetbrains.kotlin:kotlin-
stdlib:1.9.20")
            }
        }
    }
}
```

This change was made possible by including new information in the Gradle metadata of the standard library. This allows Gradle to automatically resolve the correct standard library artifacts for the other source sets.

Default support for third-party cinterop libraries

Kotlin 1.9.20 adds default support (rather than support by opt-in) for all cinterop dependencies in projects that have the Kotlin CocoaPods Gradle plugin applied.

This means you can now share more native code without being limited by platform–specific dependencies. For example, you can add dependencies on Pod libraries to the `iosMain` shared source set.

Previously, this only worked with platform-specific libraries shipped with a Kotlin/Native distribution (like Foundation, UIKit, and POSIX). All third-party Pod libraries are now available in shared source sets by default. You no longer need to specify a separate Gradle property to support them.

Support for Kotlin/Native compilation caches in Compose Multiplatform projects

This release resolves a compatibility issue with the Compose Multiplatform compiler plugin, which mostly affected Compose Multiplatform projects for iOS.

To work around this issue, you had to disable caching by using the `kotlin.native.cacheKind=none` Gradle property. However, this workaround came at a performance cost: It slowed down compilation time as caching didn't work in the Kotlin/Native compiler.

Now that the issue is fixed, you can remove `kotlin.native.cacheKind=none` from your `gradle.properties` file and enjoy the improved compilation times in your Compose Multiplatform projects.

For more tips on improving compilation times, see the Kotlin/Native documentation.

Compatibility guidelines

When configuring your projects, check the Kotlin Multiplatform Gradle plugin's compatibility with the available Gradle, Xcode, and Android Gradle plugin (AGP) versions:

Kotlin Multiplatform Gradle plugin	Gradle	Android Gradle plugin	Xcode
1.9.20	7.5 and later	7.4.2–8.2	15.0. See details below

As of this release, the recommended version of Xcode is 15.0. Libraries delivered with Xcode 15.0 are fully supported, and you can access them from anywhere in your Kotlin code.

However, XCode 14.3 should still work in the majority of cases. Keep in mind that if you use version 14.3 on your local machine, libraries delivered with Xcode 15 will be visible but not accessible.

Kotlin/Wasm

In 1.9.20, Kotlin Wasm reached the Alpha level of stability.

- Compatibility with Wasm GC phase 4 and final opcodes
- New `wasm-wasi` target, and the renaming of the `wasm` target to `wasm-js`
- Support for the WASI API in standard library
- Kotlin/Wasm API improvements

Kotlin Wasm is Alpha. It is subject to change at any time. Use it only for evaluation purposes.

We would appreciate your feedback on it in YouTrack.

Compatibility with Wasm GC phase 4 and final opcodes

Wasm GC moves to the final phase and it requires updates of opcodes – constant numbers used in the binary representation. Kotlin 1.9.20 supports the latest opcodes, so we strongly recommend that you update your Wasm projects to the latest version of Kotlin. We also recommend using the latest versions of browsers with the Wasm environment:

- Version 119 or newer for Chrome and Chromium–based browsers.
- Version 119 or newer for Firefox. Note that in Firefox 119, you need to turn on Wasm GC manually.

New wasm-wasi target, and the renaming of the wasm target to wasm-js

In this release, we're introducing a new target for Kotlin/Wasm – `wasm-wasi`. We're also renaming the `wasm` target to `wasm-js`. In the Gradle DSL, these targets are available as `wasmWasi {}` and `wasmJs {}`, respectively.

To use these targets in your project, update the `build.gradle.kts` file:

```
kotlin {
    wasmWasi {
        // ...
    }
    wasmJs {
        // ...
    }
}
```

The previously introduced `wasm {}` block has been deprecated in favor of `wasmJs {}`.

To migrate your existing Kotlin/Wasm project, do the following:

- In the `build.gradle.kts` file, rename the `wasm {}` block to `wasmJs {}`.
- In your project structure, rename the `wasmMain` directory to `wasmJsMain`.

Support for the WASI API in the standard library

In this release, we have included support for WASI, a system interface for the Wasm platform. WASI support makes it easier for you to use Kotlin/Wasm outside of browsers, for example in server–side applications, by offering a standardized set of APIs for accessing system resources. In addition, WASI provides capability–based security – another layer of security when accessing external resources.

To run Kotlin/Wasm applications, you need a VM that supports Wasm Garbage Collection (GC), for example, Node.js or Deno. Wasmtime, WasmEdge, and others are still working towards full Wasm GC support.

To import a WASI function, use the `@WasmImport` annotation:

```
import kotlin.wasm.WasmImport

@WasmImport("wasi_snapshot_preview1", "clock_time_get")
private external fun wasiRawClockTimeGet(clockId: Int, precision: Long, resultPtr: Int): Int
```

You can find a full example in our GitHub repository.



It isn't possible to use interoperability with JavaScript, while targeting `wasmWasi`.

Kotlin/Wasm API improvements

This release delivers several quality-of-life improvements to the Kotlin/Wasm API. For example, you're no longer required to return a value for DOM event listeners:

Before 1.9.20

```
fun main() {
    window.onload = {
        document.body?.sayHello()
        null
    }
}
```

In 1.9.20

```
fun main() {
    window.onload = { document.body?.sayHello() }
}
```

Gradle

Kotlin 1.9.20 is fully compatible with Gradle 6.8.3 through 8.1. You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

This version brings the following changes:

- Support for test fixtures to access internal declarations
- New property to configure paths to Konan directories
- New build report metrics for Kotlin/Native tasks

Support for test fixtures to access internal declarations

In Kotlin 1.9.20, if you use Gradle's `java-test-fixtures` plugin, then your test fixtures now have access to `internal` declarations within main source set classes. In addition, any test sources can also see any `internal` declarations within test fixtures classes.

New property to configure paths to Konan directories

In Kotlin 1.9.20, the `konan.data.dir` Gradle property is available to customize your path to the `~/.konan` directory so that you don't have to configure it through the environment variable `KONAN_DATA_DIR`.

Alternatively, you can use the `-Xkonan-data-dir` compiler option to configure your custom path to the `~/.konan` directory via the `cinterop` and `konanc` tools.

New build report metrics for Kotlin/Native tasks

In Kotlin 1.9.20, Gradle build reports now include metrics for Kotlin/Native tasks. Here is an example of a build report containing these metrics:

```
Total time for Kotlin tasks: 20.81 s (93.1 % of all tasks time)
Time   |% of Kotlin time|Task
15.24 s|73.2 %          |:compileCommonMainKotlinMetadata
5.57 s |26.8 %          |:compileNativeMainKotlinMetadata

Task ':compileCommonMainKotlinMetadata' finished in 15.24 s
Task info:
  Kotlin language version: 2.0
Time metrics:
  Total Gradle task time: 15.24 s
  Spent time before task action: 0.16 s
  Task action before worker execution: 0.21 s
  Run native in process: 2.70 s
    Run entry point: 2.64 s
Size metrics:
  Start time of task action: 2023-07-27T11:04:17

Task ':compileNativeMainKotlinMetadata' finished in 5.57 s
Task info:
  Kotlin language version: 2.0
Time metrics:
  Total Gradle task time: 5.57 s
  Spent time before task action: 0.04 s
  Task action before worker execution: 0.02 s
  Run native in process: 1.48 s
    Run entry point: 1.47 s
Size metrics:
  Start time of task action: 2023-07-27T11:04:32
```

In addition, the `kotlin.experimental.tryK2` build report now includes any Kotlin/Native tasks that were compiled and lists the language version used:

```
##### 'kotlin.experimental.tryK2' results #####
:lib:compileCommonMainKotlinMetadata: 2.0 language version
:lib:compileKotlinJvm: 2.0 language version
:lib:compileKotlinIosArm64: 2.0 language version
:lib:compileKotlinIosSimulatorArm64: 2.0 language version
:lib:compileKotlinLinuxX64: 2.0 language version
:lib:compileTestKotlinJvm: 2.0 language version
:lib:compileTestKotlinIosSimulatorArm64: 2.0 language version
:lib:compileTestKotlinLinuxX64: 2.0 language version
##### 100% (8/8) tasks have been compiled with Kotlin 2.0 #####
```

If you use Gradle 8.0, you might come across some problems with build reports, especially when Gradle configuration caching is enabled. This is a known issue, which is fixed in Gradle 8.1 and later.

Standard library

In Kotlin 1.9.20, the Kotlin/Native standard library becomes Stable, and there are some new features:

- Replacement of the Enum class values generic function
- Improved performance of HashMap operations in Kotlin/JS

Replacement of the Enum class values generic function

> This feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

In Kotlin 1.9.0, the `entries` property for enum classes became Stable. The `entries` property is a modern and performant replacement for the synthetic `values()` function. As part of Kotlin 1.9.20, there is a replacement for the generic `enumValues<T>()` function: `enumEntries<T>()`.

> The `enumValues<T>()` function is still supported, but we recommend that you use the `enumEntries<T>()` function instead because it has less performance impact. Every time you call `enumValues<T>()`, a new array is created, whereas whenever you call `enumEntries<T>()`, the same list is returned each time, which is far more efficient.

For example:

```
enum class RGB { RED, GREEN, BLUE }

@OptIn(ExperimentalStdlibApi::class)
inline fun <reified T : Enum<T>> printAllValues() {
    print(enumEntries<T>().joinToString { it.name })
}

printAllValues<RGB>()
// RED, GREEN, BLUE
```

How to enable the enumEntries function

To try this feature, opt in with `@OptIn(ExperimentalStdlibApi)` and use language version 1.9 or later. If you use the latest version of the Kotlin Gradle plugin, you don't need to specify the language version to test the feature.

The Kotlin/Native standard library becomes Stable

In Kotlin 1.9.0, we explained the actions we've taken to bring the Kotlin/Native standard library closer to our goal of stabilization. In Kotlin 1.9.20, we finally conclude this work and make the Kotlin/Native standard library Stable. Here are some highlights from this release:

- The `Vector128` class was moved from the `kotlin.native` package to the `kotlinx.cinterop` package.
- The opt-in requirement level for `ExperimentalNativeApi` and `NativeRuntimeApi` annotations, which were introduced as part of Kotlin 1.9.0, has been raised from `WARNING` to `ERROR`.
- Kotlin/Native collections now detect concurrent modifications, for example, in the `ArrayList` and `HashMap` collections.
- The `printStackTrace()` function from the `Throwable` class now prints to `STDERR` instead of `STDOUT`.

> The output format of `printStackTrace()` isn't Stable and is subject to change.

Improvements to the Atomics API

In Kotlin 1.9.0, we said that the Atomics API would be ready to become Stable when the Kotlin/Native standard library becomes Stable. Kotlin 1.9.20 includes the following additional changes:

- Experimental `AtomicIntArray`, `AtomicLongArray`, and `AtomicArray<T>` classes are introduced. These new classes are designed specifically to be consistent with Java's atomic arrays so that in the future, they can be included in the common standard library.

  > The `AtomicIntArray`, `AtomicLongArray`, and `AtomicArray<T>` classes are Experimental. They may be dropped or changed at any time. To try them, opt in with `@OptIn(ExperimentalStdlibApi)`. Use them only for evaluation purposes. We would appreciate your feedback in YouTrack.

- In the `kotlin.native.concurrent` package, the Atomics API that was deprecated in Kotlin 1.9.0 with deprecation level `WARNING` has had its deprecation level raised to `ERROR`.
- In the `kotlin.concurrent` package, member functions of the `AtomicInt` and `AtomicLong` classes that had deprecation level: `ERROR`, have been removed.
- All member functions of the `AtomicReference` class now use atomic intrinsic functions.

For more information on all of the changes in Kotlin 1.9.20, see our YouTrack ticket.

Improved performance of HashMap operations in Kotlin/JS

Kotlin 1.9.20 improves the performance of `HashMap` operations and reduces their memory footprint in Kotlin/JS. Internally, Kotlin/JS has changed its internal implementation to open addressing. This means that you should see performance improvements when you:

- Insert new elements into a `HashMap`.
- Search for existing elements in a `HashMap`.
- Iterate through keys or values in a `HashMap`.

Documentation updates

The Kotlin documentation has received some notable changes:

- The JVM Metadata API reference – Explore how you can parse metadata with Kotlin/JVM.
- Time measurement guide – Learn how to calculate and measure time in Kotlin.
- Improved Collections chapter in the tour of Kotlin – Learn the fundamentals of the Kotlin programming language with chapters including both theory and practice.
- Definitely non-nullable types – Learn about definitely non-nullable generic types.
- Improved Arrays page – Learn about arrays and when to use them.
- Expected and actual declarations in Kotlin Multiplatform – Learn about the Kotlin mechanism of expected and actual declarations in Kotlin Multiplatform.

Install Kotlin 1.9.20

Check the IDE version

IntelliJ IDEA 2023.1.x and 2023.2.x automatically suggest updating the Kotlin plugin to version 1.9.20. IntelliJ IDEA 2023.3 will include the Kotlin 1.9.20 plugin.

Android Studio Hedgehog (231) and Iguana (232) will support Kotlin 1.9.20 in their upcoming releases.

The new command-line compiler is available for download on the GitHub release page.

Configure Gradle settings

To download Kotlin artifacts and dependencies, update your `settings.gradle(.kts)` file to use the Maven Central repository:

```
pluginManagement {
    repositories {
        mavenCentral()
        gradlePluginPortal()
    }
}
```

If the repository is not specified, Gradle uses the sunset JCenter repository, which could lead to issues with Kotlin artifacts.

What's new in Kotlin 1.9.0

Release date: July 6, 2023

The Kotlin 1.9.0 release is out and the K2 compiler for the JVM is now in **Beta**. Additionally, here are some of the main highlights:

- New Kotlin K2 compiler updates
- Stable replacement of the enum class values function
- Stable `..<` operator for open-ended ranges
- New common function to get regex capture group by name
- New path utility to create parent directories
- Preview of the Gradle configuration cache in Kotlin Multiplatform
- Changes to Android target support in Kotlin Multiplatform
- Preview of custom memory allocator in Kotlin/Native
- Library linkage in Kotlin/Native
- Size-related optimizations in Kotlin/Wasm

You can also find a short overview of the updates in this video:

Watch video online.

For information about the Kotlin release cycle, see Kotlin release process.

IDE support

The Kotlin plugins that support 1.9.0 are available for:

IDE	Supported versions
IntelliJ IDEA	2022.3.x, 2023.1.x
Android Studio	Giraffe (223), Hedgehog (231)*

*The Kotlin 1.9.0 plugin will be included with Android Studio Giraffe (223) and Hedgehog (231) in their upcoming releases.

The Kotlin 1.9.0 plugin will be included with IntelliJ IDEA 2023.2 in the upcoming releases.

To download Kotlin artifacts and dependencies, configure your Gradle settings to use the Maven Central Repository.

New Kotlin K2 compiler updates

The Kotlin team at JetBrains continues to stabilize the K2 compiler, and the 1.9.0 release introduces further advancements. The K2 compiler for the JVM is now in **Beta**.

There's now also basic support for Kotlin/Native and multiplatform projects.

Compatibility of the kapt compiler plugin with the K2 compiler

You can use the kapt plugin in your project along with the K2 compiler, but with some restrictions. Despite setting `languageVersion` to `2.0`, the kapt compiler plugin still utilizes the old compiler.

If you execute the kapt compiler plugin within a project where `languageVersion` is set to `2.0`, kapt will automatically switch to `1.9` and disable specific version compatibility checks. This behavior is equivalent to including the following command arguments:

- `-Xskip-metadata-version-check`
- `-Xskip-prerelease-check`
- `-Xallow-unstable-dependencies`

These checks are exclusively disabled for kapt tasks. All other compilation tasks will continue to utilize the new K2 compiler.

If you encounter any issues when using kapt with the K2 compiler, please report them to our issue tracker.

Try the K2 compiler in your project

Starting with 1.9.0 and until the release of Kotlin 2.0, you can easily test the K2 compiler by adding the `kotlin.experimental.tryK2=true` Gradle property to your `gradle.properties` file. You can also run the following command:

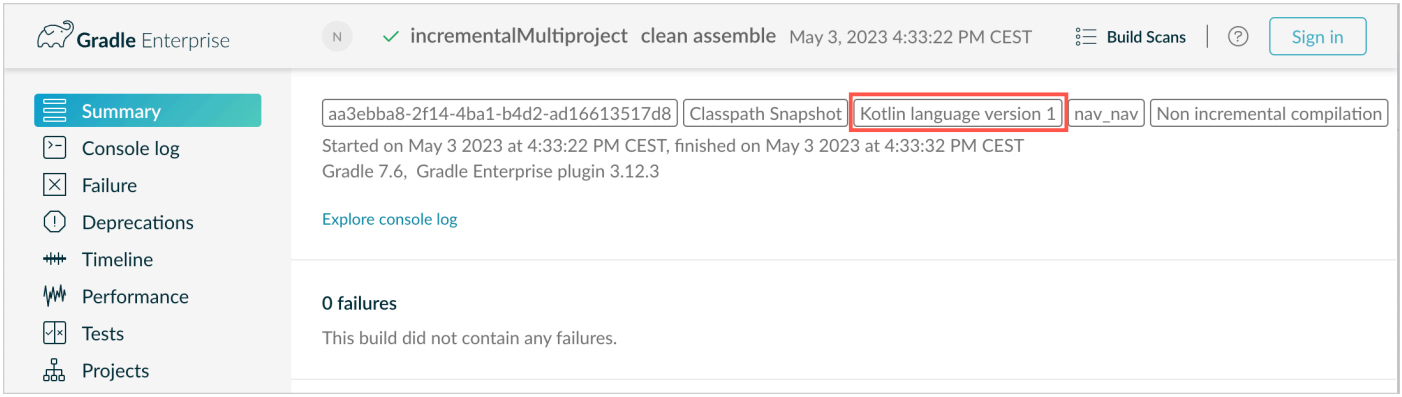
```
./gradlew assemble -Pkotlin.experimental.tryK2=true
```

This Gradle property automatically sets the language version to 2.0 and updates the build report with the number of Kotlin tasks compiled using the K2 compiler compared to the current compiler:

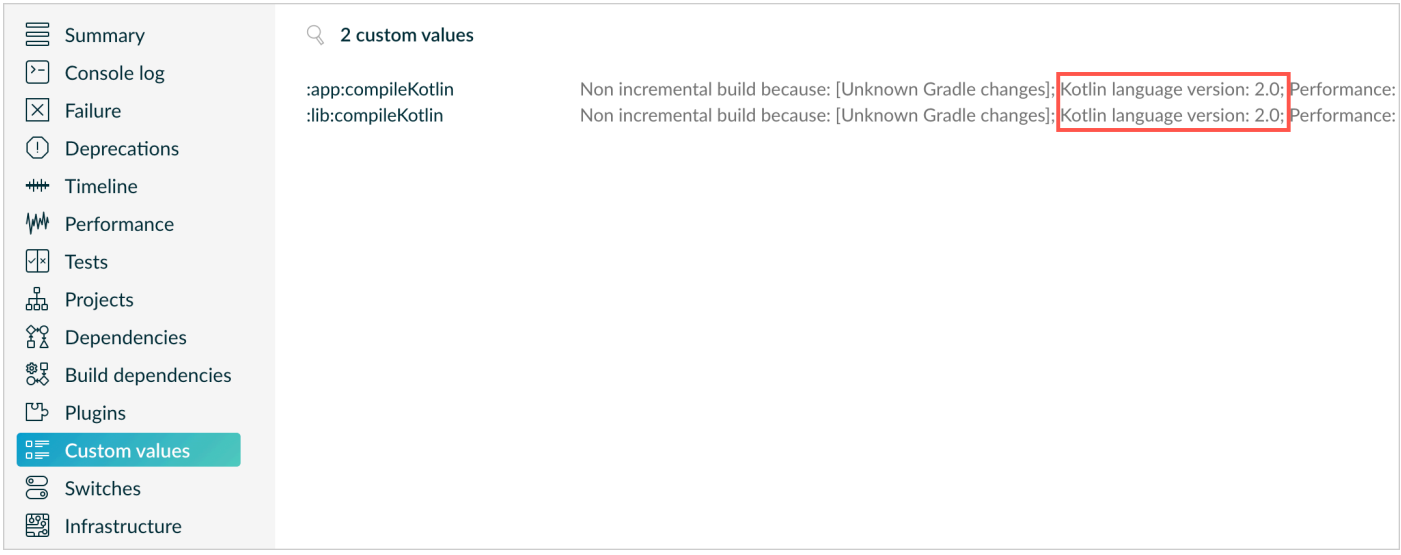
```
##### 'kotlin.experimental.tryK2' results (Kotlin/Native not checked) #####
:lib:compileKotlin: 2.0 language version
:app:compileKotlin: 2.0 language version
##### 100% (2/2) tasks have been compiled with Kotlin 2.0 #####
```

Gradle build reports

Gradle build reports now show whether the current or the K2 compiler was used to compile the code. In Kotlin 1.9.0, you can see this information in your Gradle build scans:



Gradle build scan - K1



Gradle build scan - K2

You can also find the Kotlin version used in the project right in the build report:

```
Task info:
  Kotlin language version: 1.9
```

If you use Gradle 8.0, you might come across some problems with build reports, especially when Gradle configuration caching is enabled. This is a known issue, fixed in Gradle 8.1 and later.

Current K2 compiler limitations

Enabling K2 in your Gradle project comes with certain limitations that can affect projects using Gradle versions below 8.3 in the following cases:

- Compilation of source code from `buildSrc`.
- Compilation of Gradle plugins in included builds.
- Compilation of other Gradle plugins if they are used in projects with Gradle versions below 8.3.
- Building Gradle plugin dependencies.

If you encounter any of the problems mentioned above, you can take the following steps to address them:

- Set the language version for `buildSrc`, any Gradle plugins, and their dependencies:

```
kotlin {
    compilerOptions {
        languageVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9)
        apiVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9)
    }
}
```

- Update the Gradle version in your project to 8.3 when it becomes available.

Leave your feedback on the new K2 compiler

We'd appreciate any feedback you may have!

- Provide your feedback directly to K2 developers Kotlin's Slack – get an invite and join the #k2-early-adopters channel.
- Report any problems you've faced with the new K2 compiler on our issue tracker.
- Enable the **Send usage statistics** option to allow JetBrains to collect anonymous data about K2 usage.

Language

In Kotlin 1.9.0, we're stabilizing some new language features that were introduced earlier:

- Replacement of the enum class values function
- Data object symmetry with data classes
- Support for secondary constructors with bodies in inline value classes

Stable replacement of the enum class values function

In 1.8.20, the `entries` property for enum classes was introduced as an Experimental feature. The `entries` property is a modern and performant replacement for the synthetic `values()` function. In 1.9.0, the `entries` property is Stable.

> The `values()` function is still supported, but we recommend that you use the `entries` property instead.

```
enum class Color(val colorName: String, val rgb: String) {
    RED("Red", "#FF0000"),
    ORANGE("Orange", "#FF7F00"),
    YELLOW("Yellow", "#FFFF00")
}

fun findByRgb(rgb: String): Color? = Color.entries.find { it.rgb == rgb }
```

For more information about the `entries` property for enum classes, see What's new in Kotlin 1.8.20.

Stable data objects for symmetry with data classes

Data object declarations, which were introduced in Kotlin 1.8.20, are now Stable. This includes the functions added for symmetry with data classes: `toString()`, `equals()`, and `hashCode()`.

This feature is particularly useful with `sealed` hierarchies (like a `sealed class` or `sealed interface` hierarchy), because `data object` declarations can be used conveniently alongside `data class` declarations. In this example, declaring `EndOfFile` as a `data object` instead of a plain `object` means that it automatically has a `toString()` function without the need to override it manually. This maintains symmetry with the accompanying data class definitions.

```
sealed interface ReadResult
data class Number(val number: Int) : ReadResult
data class Text(val text: String) : ReadResult
data object EndOfFile : ReadResult

fun main() {
    println(Number(7)) // Number(number=7)
    println(EndOfFile) // EndOfFile
}
```

For more information, see What's new in Kotlin 1.8.20.

Support for secondary constructors with bodies in inline value classes

Starting with Kotlin 1.9.0, the use of secondary constructors with bodies in inline value classes is available by default:

```
@JvmInline
value class Person(private val fullName: String) {
    // Allowed since Kotlin 1.4.30:
    init {
        check(fullName.isNotBlank()) {
            "Full name shouldn't be empty"
        }
    }
    // Allowed by default since Kotlin 1.9.0:
    constructor(name: String, lastName: String) : this("$name $lastName") {
        check(lastName.isNotBlank()) {
            "Last name shouldn't be empty"
        }
    }
}
```

Previously, Kotlin allowed only public primary constructors in inline classes. As a result, it was impossible to encapsulate underlying values or create an inline class that would represent some constrained values.

As Kotlin developed, these issues were fixed. Kotlin 1.4.30 lifted restrictions on `init` blocks and then Kotlin 1.8.20 came with a preview of secondary constructors with bodies. They are now available by default. Learn more about the development of Kotlin inline classes in this KEEP.

Kotlin/JVM

Starting with version 1.9.0, the compiler can generate classes with a bytecode version corresponding to JVM 20. In addition, the deprecation of the `JvmDefault` annotation and legacy `-Xjvm-default` modes continues.

Deprecation of JvmDefault annotation and legacy -Xjvm-default modes

Starting from Kotlin 1.5, the usage of the `JvmDefault` annotation has been deprecated in favor of the newer `-Xjvm-default` modes: `all` and `all-compatibility`. With the introduction of `JvmDefaultWithoutCompatibility` in Kotlin 1.4 and `JvmDefaultWithCompatibility` in Kotlin 1.6, these modes offer comprehensive control over the generation of `DefaultImpls` classes, ensuring seamless compatibility with older Kotlin code.

Consequently in Kotlin 1.9.0, the `JvmDefault` annotation no longer holds any significance and has been marked as deprecated, resulting in an error. It will eventually be removed from Kotlin.

Kotlin/Native

Among other improvements, this release brings further advancements to the Kotlin/Native memory manager that should enhance its robustness and performance:

- Preview of custom memory allocator
- Objective-C or Swift object deallocation hook on the main thread
- No object initialization when accessing constant values in Kotlin/Native
- Ability to configure standalone mode for iOS simulator tests
- Library linkage in Kotlin/Native

Preview of custom memory allocator

Kotlin 1.9.0 introduces the preview of a custom memory allocator. Its allocation system improves the runtime performance of the Kotlin/Native memory manager.

The current object allocation system in Kotlin/Native uses a general-purpose allocator that doesn't have the functionality for efficient garbage collection. To compensate, it maintains thread-local linked lists of all allocated objects before the garbage collector (GC) merges them into a single list, which can be iterated during sweeping. This approach comes with several performance downsides:

- The sweeping order lacks memory locality and often results in scattered memory access patterns, leading to potential performance issues.
- Linked lists require additional memory for each object, increasing memory usage, particularly when dealing with many small objects.
- The single list of allocated objects makes it challenging to parallelize sweeping, which can cause memory usage problems when mutator threads allocate objects faster than the GC thread can collect them.

To address these issues, Kotlin 1.9.0 introduces a preview of the custom allocator. It divides system memory into pages, allowing independent sweeping in consecutive order. Each allocation becomes a memory block within a page, and the page keeps track of block sizes. Different page types are optimized for various allocation sizes. The consecutive arrangement of memory blocks ensures efficient iteration through all allocated blocks.

When a thread allocates memory, it searches for a suitable page based on the allocation size. Threads maintain a set of pages for different size categories. Typically, the current page for a given size can accommodate the allocation. If not, the thread requests a different page from the shared allocation space. This page may already be available, require sweeping, or should be created first.

The new allocator allows having multiple independent allocation spaces simultaneously, which will allow the Kotlin team to experiment with different page layouts to improve performance even further.

For more information on the design of the new allocator, see this README.

How to enable

Add the `-Xallocator=custom` compiler option:

```
kotlin {
    macosX64("native") {
        binaries.executable()

        compilations.configureEach {
            compilerOptions.configure {
                freeCompilerArgs.add("-Xallocator=custom")
            }
        }
    }
}
```

Leave feedback

We would appreciate your feedback in YouTrack to improve the custom allocator.

Objective-C or Swift object deallocation hook on the main thread

Starting with Kotlin 1.9.0, the Objective-C or Swift object deallocation hook is called on the main thread if the object is passed to Kotlin there. The way the Kotlin/Native memory manager previously handled references to Objective-C objects could lead to memory leaks. We believe the new behavior should improve the robustness of the memory manager.

Consider an Objective-C object that is referenced in Kotlin code, for example, when passed as an argument, returned by a function, or retrieved from a collection. In this case, Kotlin creates its own object that holds the reference to the Objective-C object. When the Kotlin object gets deallocated, the Kotlin/Native runtime calls the `objc_release` function that releases that Objective-C reference.

Previously, the Kotlin/Native memory manager ran `objc_release` on a special GC thread. If it's the last object reference, the object gets deallocated. Issues could come up when Objective-C objects have custom deallocation hooks like the `dealloc` method in Objective-C or the `deinit` block in Swift, and these hooks expect to be called on a specific thread.

Since hooks for objects on the main thread usually expect to be called there, Kotlin/Native runtime now calls `objc_release` on the main thread as well. It should cover the cases when the Objective-C object was passed to Kotlin on the main thread, creating a Kotlin peer object there. This only works if the main dispatch queue is processed, which is the case for regular UI applications. When it's not the main queue or the object was passed to Kotlin on a thread other than main, the `objc_release` is called on a special GC thread as before.

How to opt out

In case you face issues, you can disable this behavior in your `gradle.properties` file with the following option:

```
kotlin.native.binary.objcDisposeOnMain=false
```

Don't hesitate to report such cases to our issue tracker.

No object initialization when accessing constant values in Kotlin/Native

Starting with Kotlin 1.9.0, the Kotlin/Native backend doesn't initialize objects when accessing `const val` fields:

```
object MyObject {
    init {
        println("side effect!")
    }

    const val y = 1
}

fun main() {
    println(MyObject.y) // No initialization at first
    val x = MyObject    // Initialization occurs
    println(x.y)
}
```

The behavior is now unified with Kotlin/JVM, where the implementation is consistent with Java and objects are never initialized in this case. You can also expect some performance improvements in your Kotlin/Native projects thanks to this change.

Ability to configure standalone mode for iOS simulator tests in Kotlin/Native

By default, when running iOS simulator tests for Kotlin/Native, the `--standalone` flag is used to avoid manual simulator booting and shutdown. In 1.9.0, you can now configure whether this flag is used in a Gradle task via the `standalone` property. By default, the `--standalone` flag is used so standalone mode is enabled.

Here is an example of how to disable standalone mode in your `build.gradle.kts` file:

```
tasks.withType<org.jetbrains.kotlin.gradle.targets.native.tasks.KotlinNativeSimulatorTest>
().configureEach {
    standalone.set(false)
}
```

If you disable standalone mode, you must boot the simulator manually. To boot your simulator from CLI, you can use the following command:

```
/usr/bin/xcrun simctl boot <DeviceId>
```

Library linkage in Kotlin/Native

Starting with Kotlin 1.9.0, the Kotlin/Native compiler treats linkage issues in Kotlin libraries the same way as Kotlin/JVM. You might face such issues if the author of one third-party Kotlin library makes an incompatible change in experimental APIs that another third-party Kotlin library consumes.

Now builds don't fail during compilation in case of linkage issues between third-party Kotlin libraries. Instead, you'll only encounter these errors in run time, exactly as on the JVM.

The Kotlin/Native compiler reports warnings every time it detects issues with library linkage. You can find such warnings in your compilation logs, for example:

```
No function found for symbol 'org.samples/MyRemovedClass.doSomething|3657632771909858561[0]'

Can not get instance of singleton 'MyEnumClass.REMOVED_ENTRY': No enum entry found for symbol
'org.samples/MyEnumClass.REMOVED_ENTRY|null[0]'

Function 'getMyRemovedClass' can not be called: Function uses unlinked class symbol
'org.samples/MyRemovedClass|null[0]'
```

You can further configure or even disable this behavior in your projects:

- If you don't want to see these warnings in your compilation logs, suppress them with the `-Xpartial-linkage-loglevel=INFO` compiler option.
- It's also possible to raise the severity of reported warnings to compilation errors with `-Xpartial-linkage-loglevel=ERROR`. In this case, the compilation fails and you'll see all the errors in the compilation log. Use this option to examine the linkage issues more closely.
- If you face unexpected problems with this feature, you can always opt out with the `-Xpartial-linkage=disable` compiler option. Don't hesitate to report such cases to our issue tracker.

```
// An example of passing compiler options via Gradle build file.
kotlin {
    macosX64("native") {
        binaries.executable()

        compilations.configureEach {
            compilerOptions.configure {
                // To suppress linkage warnings:
                freeCompilerArgs.add("-Xpartial-linkage-loglevel=INFO")

                // To raise linkage warnings to errors:
                freeCompilerArgs.add("-Xpartial-linkage-loglevel=ERROR")

                // To disable the feature completely:
                freeCompilerArgs.add("-Xpartial-linkage=disable")
            }
        }
    }
}
```

Compiler option for C interop implicit integer conversions

We have introduced a compiler option for C interop that allows you to use implicit integer conversions. After careful consideration, we've introduced this compiler option to prevent unintentional use as this feature still has room for improvement and our aim is to have an API of the highest quality.

In this code sample an implicit integer conversion allows `options = 0` even though `options` has unsigned type `UInt` and `0` is signed.

```
val today = NSDate()
val tomorrow = NSCalendar.currentCalendar.dateByAddingUnit(
    unit = NSCalendarUnitDay,
    value = 1,
    toDate = today,
    options = 0
)
```

To use implicit conversions with native interop libraries, use the `-XXLanguage:+ImplicitSignedToUnsignedIntegerConversion` compiler option.

You can configure this in your Gradle `build.gradle.kts` file:

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinNativeCompile>().configureEach {
    compilerOptions.freeCompilerArgs.addAll(
        "-XXLanguage:+ImplicitSignedToUnsignedIntegerConversion"
    )
}
```

Kotlin Multiplatform

Kotlin Multiplatform has received some notable updates in 1.9.0 designed to improve your developer experience:

- Changes to Android target support
- New Android source set layout enabled by default
- Preview of the Gradle configuration cache in multiplatform projects

Changes to Android target support

We continue our efforts to stabilize Kotlin Multiplatform. An essential step is to provide first-class support for the Android target. We're excited to announce that in the future, the Android team from Google will provide its own Gradle plugin to support Android in Kotlin Multiplatform.

To open the way for this new solution from Google, we're renaming the `android` block in the current Kotlin DSL in 1.9.0. Please change all the occurrences of the `android` block to `androidTarget` in your build scripts. This is a temporary change that is necessary to free the `android` name for the upcoming DSL from Google.

The Google plugin will be the preferred way of working with Android in multiplatform projects. When it's ready, we'll provide the necessary migration instructions so that you'll be able to use the short `android` name as before.

New Android source set layout enabled by default

Starting with Kotlin 1.9.0, the new Android source set layout is the default. It replaced the previous naming schema for directories, which was confusing in multiple ways. The new layout has a number of advantages:

- Simplified type semantics – The new Android source layout provides clear and consistent naming conventions that help to distinguish between different types of source sets.
- Improved source directory layout – With the new layout, the `SourceDirectories` arrangement becomes more coherent, making it easier to organize code and locate source files.
- Clear naming schema for Gradle configurations – The schema is now more consistent and predictable in both `KotlinSourceSets` and `AndroidSourceSets`.

The new layout requires the Android Gradle plugin version 7.0 or later and is supported in Android Studio 2022.3 and later. See our migration guide to make the necessary changes in your `build.gradle(.kts)` file.

Preview of the Gradle configuration cache

Kotlin 1.9.0 comes with support for the Gradle configuration cache in multiplatform libraries. If you're a library author, you can already benefit from the improved build performance.

The Gradle configuration cache speeds up the build process by reusing the results of the configuration phase for subsequent builds. The feature has become Stable since Gradle 8.1. To enable it, follow the instructions in the Gradle documentation.

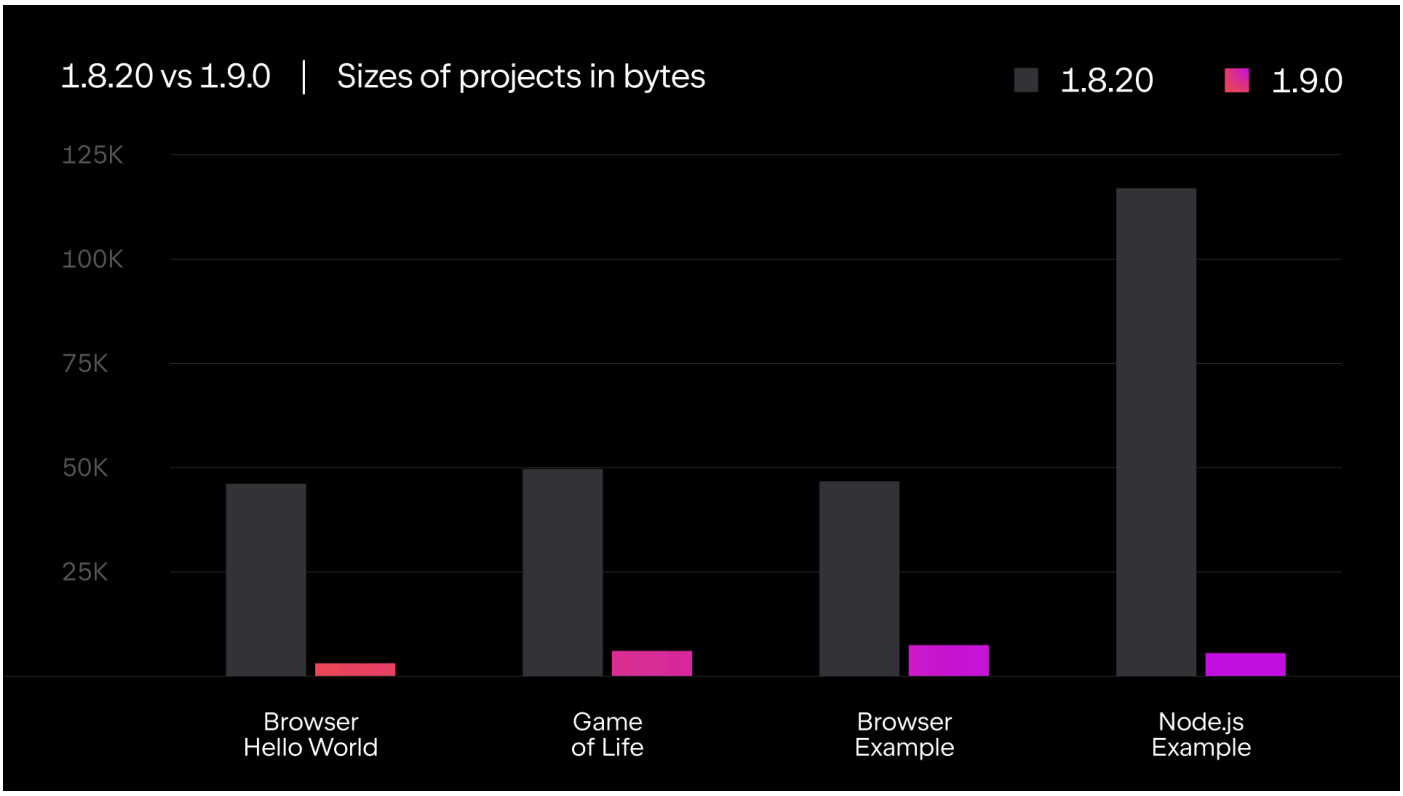
> The Kotlin Multiplatform plugin still doesn't support the Gradle configuration cache with Xcode integration tasks or the Kotlin CocoaPods Gradle plugin. We expect to add this feature in future Kotlin releases.

Kotlin/Wasm

The Kotlin team continues to experiment with the new Kotlin/Wasm target. This release introduces several performance and size-related optimizations, along with updates in JavaScript interop.

Size-related optimizations

Kotlin 1.9.0 introduces significant size improvements for WebAssembly (Wasm) projects. Comparing two "Hello World" projects, the code footprint for Wasm in Kotlin 1.9.0 is now over 10 times smaller than in Kotlin 1.8.20.



Kotlin/Wasm size-related optimizations

These size optimizations result in more efficient resource utilization and improved performance when targeting Wasm platforms with Kotlin code.

Updates in JavaScript interop

This Kotlin update introduces changes to the interoperability between Kotlin and JavaScript for Kotlin/Wasm. As Kotlin/Wasm is an Experimental feature, certain limitations apply to its interoperability.

Restriction of Dynamic types

Starting with version 1.9.0, Kotlin no longer supports the use of `Dynamic` types in Kotlin/Wasm. This is now deprecated in favor of the new universal `JsAny` type, which facilitates JavaScript interoperability.

For more details, see the Kotlin/Wasm interoperability with JavaScript documentation.

Restriction of non-external types

Kotlin/Wasm supports conversions for specific Kotlin static types when passing values to and from JavaScript. These supported types include:

- Primitives, such as signed numbers, `Boolean`, and `Char`.
- `String`.
- Function types.

Other types were passed without conversion as opaque references, leading to inconsistencies between JavaScript and Kotlin subtyping.

To address this, Kotlin restricts JavaScript interop to a well-supported set of types. Starting from Kotlin 1.9.0, only external, primitive, string, and function types are supported in Kotlin/Wasm JavaScript interop. Furthermore, a

separate explicit type called `JsReference` has been introduced to represent handles to Kotlin/Wasm objects that can be used in JavaScript interop.

For more details, refer to the Kotlin/Wasm interoperability with JavaScript documentation.

Kotlin/Wasm in Kotlin Playground

Kotlin Playground supports the Kotlin/Wasm target. You can write, run, and share your Kotlin code that targets the Kotlin/Wasm. Check it out!

> Using Kotlin/Wasm requires enabling experimental features in your browser.
>
> Learn more about how to enable these features.

```
import kotlin.time.*
import kotlin.time.measureTime

fun main() {
    println("Hello from Kotlin/Wasm!")
    computeAck(3, 10)
}

tailrec fun ack(m: Int, n: Int): Int = when {
    m == 0 -> n + 1
    n == 0 -> ack(m - 1, 1)
    else -> ack(m - 1, ack(m, n - 1))
}

fun computeAck(m: Int, n: Int) {
    var res = 0
    val t = measureTime {
        res = ack(m, n)
    }
    println()
    println("ack($m, $n) = ${res}")
    println("duration: ${t.inWholeNanoseconds / 1e6} ms")
}
```

Kotlin/JS

This release introduces updates for Kotlin/JS, including the removal of the old Kotlin/JS compiler, Kotlin/JS Gradle plugin deprecation and Experimental support for ES2015:

- Removal of the old Kotlin/JS compiler
- Deprecation of the Kotlin/JS Gradle plugin
- Deprecation of external enum
- Experimental support for ES2015 classes and modules
- Changed default destination of JS production distribution
- Extract org.w3c declarations from stdlib-js

> Starting from version 1.9.0, partial library linkage is also enabled for Kotlin/JS.

Removal of the old Kotlin/JS compiler

In Kotlin 1.8.0, we announced that the IR-based backend became Stable. Since then, not specifying the compiler has become an error, and using the old compiler leads to warnings.

In Kotlin 1.9.0, using the old backend results in an error. Please migrate to the IR compiler.

Deprecation of the Kotlin/JS Gradle plugin

Starting with Kotlin 1.9.0, the `kotlin-js` Gradle plugin is deprecated. We encourage you to use the `kotlin-multiplatform` Gradle plugin with the `js()` target instead.

The functionality of the Kotlin/JS Gradle plugin essentially duplicated the `kotlin-multiplatform` plugin and shared the same implementation under the hood. This overlap created confusion and increased maintenance load on the Kotlin team.

Refer to our Compatibility guide for Kotlin Multiplatform for migration instructions. If you find any issues that aren't covered in the guide, please report them to our issue tracker.

Deprecation of external enum

In Kotlin 1.9.0, the use of external enums will be deprecated due to issues with static enum members like `entries`, that can't exist outside Kotlin. We recommend using an external sealed class with object subclasses instead:

```
// Before
external enum class ExternalEnum { A, B }

// After
external sealed class ExternalEnum {
    object A: ExternalEnum
    object B: ExternalEnum
}
```

By switching to an external sealed class with object subclasses, you can achieve similar functionality to external enums while avoiding the problems associated with default methods.

Starting from Kotlin 1.9.0, the use of external enums will be marked as deprecated. We encourage you to update your code to utilize the suggested external sealed class implementation for compatibility and future maintenance.

Experimental support for ES2015 classes and modules

This release introduces Experimental support for ES2015 modules and generation of ES2015 classes:

- Modules offer a way to simplify your codebase and improve maintainability.
- Classes allow you to incorporate object-oriented programming (OOP) principles, resulting in cleaner and more intuitive code.

To enable these features, update your `build.gradle.kts` file accordingly:

```
// build.gradle.kts
kotlin {
    js(IR) {
        useEsModules() // Enables ES2015 modules
        browser()
    }
}

// Enables ES2015 classes generation
tasks.withType<KotlinJsCompile>().configureEach {
    kotlinOptions {
        useEsClasses = true
    }
}
```

Learn more about ES2015 (ECMAScript 2015, ES6) in the official documentation.

Changed default destination of JS production distribution

Prior to Kotlin 1.9.0, the distribution target directory was `build/distributions`. However, this is a common directory for Gradle archives. To resolve this issue, we've changed the default distribution target directory in Kotlin 1.9.0 to: `build/dist/<targetName>/<binaryName>`.

For example, `productionExecutable` was in `build/distributions`. In Kotlin 1.9.0, it's in `build/dist/js/productionExecutable`.

If you have a pipeline in place that uses the results of these builds, make sure to update the directory.

Extract org.w3c declarations from stdlib-js

Since Kotlin 1.9.0, the `stdlib-js` no longer includes `org.w3c` declarations. Instead, these declarations have been moved to a separate Gradle dependency. When you add the Kotlin Multiplatform Gradle plugin to your `build.gradle.kts` file, these declarations will be automatically included in your project, similar to the standard library.

There is no need for any manual action or migration. The necessary adjustments will be handled automatically.

Gradle

Kotlin 1.9.0 comes with new Gradle compiler options and a lot more:

- Removed classpath property
- New Gradle compiler options
- Project-level compiler options for Kotlin/JVM
- Compiler option for Kotlin/Native module name
- Separate compiler plugins for official Kotlin libraries
- Incremented minimum supported version
- kapt doesn't cause eager task creation
- Programmatic configuration of the JVM target validation mode

Removed classpath property

In Kotlin 1.7.0, we announced the start of a deprecation cycle for the `KotlinCompile` task's property: `classpath`. The deprecation level was raised to `ERROR` in Kotlin 1.8.0. In this release, we've finally removed the `classpath` property. All compile tasks should now use the `libraries` input for a list of libraries required for compilation.

New compiler options

The Kotlin Gradle plugin now provides new properties for opt-ins and the compiler's progressive mode.

- To opt in to new APIs, you can now use the `optIn` property and pass a list of strings like: `optIn.set(listOf(a, b, c))`.
- To enable progressive mode, use `progressiveMode.set(true)`.

Project-level compiler options for Kotlin/JVM

Starting with Kotlin 1.9.0, a new `compilerOptions` block is available inside the `kotlin` configuration block:

```
kotlin {
    compilerOptions {
        jvmTarget.set(JVM.Target_11)
    }
}
```

It makes configuring compiler options much easier. However, it is important to note some important details:

- This configuration only works on the project level.
- For the Android plugin, this block configures the same object as:

```
android {
    kotlinOptions {}
}
```

- The `android.kotlinOptions` and `kotlin.compilerOptions` configuration blocks override each other. The last (lowest) block in the build file always takes effect.
- If `moduleName` is configured on the project level, its value could be changed when passed to the compiler. It's not the case for the `main` compilation, but for other types, for example, test sources, the Kotlin Gradle plugin will add the `_test` suffix.
- The configuration inside the `tasks.withType<KotlinJvmCompile>().configureEach {}` (or `tasks.named<KotlinJvmCompile>("compileKotlin") { }`) overrides both `kotlin.compilerOptions` and `android.kotlinOptions`.

Compiler option for Kotlin/Native module name

The Kotlin/Native `module-name` compiler option is now easily available in the Kotlin Gradle plugin.

This option specifies a name for the compilation module and can also be used for adding a name prefix for declarations exported to Objective-C.

You can now set the module name directly in the `compilerOptions` block of your Gradle build files:

Kotlin

```kotlin
tasks.named<org.jetbrains.kotlin.gradle.tasks.KotlinNativeCompile>("compileKotlinLinuxX64") {
    compilerOptions {
        moduleName.set("my-module-name")
    }
}
```

Groovy

```groovy
tasks.named("compileKotlinLinuxX64", org.jetbrains.kotlin.gradle.tasks.KotlinNativeCompile.class) {
    compilerOptions {
        moduleName = "my-module-name"
    }
}
```

Separate compiler plugins for official Kotlin libraries

Kotlin 1.9.0 introduces separate compiler plugins for its official libraries. Previously, compiler plugins were embedded into their corresponding Gradle plugins. This could cause compatibility issues in case the compiler plugin was compiled against a Kotlin version higher than the Gradle build's Kotlin runtime version.

Now compiler plugins are added as separate dependencies, so you'll no longer face compatibility issues with older Gradle versions. Another major advantage of the new approach is that new compiler plugins can be used with other build systems like Bazel.

Here's the list of new compiler plugins we're now publishing to Maven Central:

- kotlin-atomicfu-compiler-plugin
- kotlin-allopen-compiler-plugin
- kotlin-lombok-compiler-plugin
- kotlin-noarg-compiler-plugin
- kotlin-sam-with-receiver-compiler-plugin
- kotlinx-serialization-compiler-plugin

Every plugin has its `-embeddable` counterpart, for example, `kotlin-allopen-compiler-plugin-embeddable` is designed for working with the `kotlin-compiler-embeddable` artifact, the default option for scripting artifacts.

Gradle adds these plugins as compiler arguments. You don't need to make any changes to your existing projects.

Incremented minimum supported version

Starting with Kotlin 1.9.0, the minimum supported Android Gradle plugin version is 4.2.2.

See the Kotlin Gradle plugin's compatibility with available Gradle versions in our documentation.

kapt doesn't cause eager task creation in Gradle

Prior to 1.9.0, the kapt compiler plugin caused eager task creation by requesting the configured instance of the Kotlin compilation task. This behavior has been fixed in Kotlin 1.9.0. If you use the default configuration for your `build.gradle.kts` file then your setup is not affected by this change.

If you use a custom configuration, your setup will be adversely affected. For example, if you have modified the `KotlinJvmCompile` task using Gradle's tasks API, you must similarly modify the `KaptGenerateStubs` task in your build script.

For example, if your script has the following configuration for the `KotlinJvmCompile` task:

```
tasks.named<KotlinJvmCompile>("compileKotlin") { // Your custom configuration }
```

In this case, you need to make sure that the same modification is included as part of the `KaptGenerateStubs` task:

```
tasks.named<KaptGenerateStubs>("kaptGenerateStubs") { // Your custom configuration }
```

For more information, see our YouTrack ticket.

Programmatic configuration of the JVM target validation mode

Before Kotlin 1.9.0, there was only one way to adjust the detection of JVM target incompatibility between Kotlin and Java. You had to set `kotlin.jvm.target.validation.mode=ERROR` in your `gradle.properties` for the whole project.

You can now also configure it on the task level in your `build.gradle.kts` file:

```
tasks.named<org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompile>("compileKotlin") {
    jvmTargetValidationMode.set(org.jetbrains.kotlin.gradle.dsl.jvm.JvmTargetValidationMode.WARNING)
}
```

Standard library

Kotlin 1.9.0 has some great improvements for the standard library:

- The `..<` operator and time API are Stable.
- The Kotlin/Native standard library has been thoroughly reviewed and updated
- The `@Volatile` annotation can be used on more platforms
- There's a **common** function to get a regex capture group by name
- The `HexFormat` class has been introduced to format and parse hexadecimals

Stable ..< operator for open-ended ranges

The new `..<` operator for open-ended ranges that was introduced in Kotlin 1.7.20 and became Stable in 1.8.0. In 1.9.0, the standard library API for working with open-ended ranges is also Stable.

Our research shows that the new `..<` operator makes it easier to understand when an open-ended range is declared. If you use the `until` infix function, it's easy to make the mistake of assuming that the upper bound is included.

Here is an example using the `until` function:

```
fun main() {
    for (number in 2 until 10) {
        if (number % 2 == 0) {
            print("$number ")
        }
    }
    // 2 4 6 8
}
```

And here is an example using the new `..<` operator:

```
fun main() {
    for (number in 2..<10) {
        if (number % 2 == 0) {
            print("$number ")
        }
    }
    // 2 4 6 8
}
```

From IntelliJ IDEA version 2023.1.1, a new code inspection is available that highlights when you can use the `..<` operator.

For more information about what you can do with this operator, see What's new in Kotlin 1.7.20.

Stable time API

Since 1.3.50, we have previewed a new time measurement API. The duration part of the API became Stable in 1.6.0. In 1.9.0, the remaining time measurement API is Stable.

The old time API provided the `measureTimeMillis` and `measureNanoTime` functions, which aren't intuitive to use. Although it is clear that they both measure time in different units, it isn't clear that `measureTimeMillis` uses a wall clock to measure time, whereas `measureNanoTime` uses a monotonic time source. The new time API resolves this and other issues to make the API more user friendly.

With the new time API, you can easily:

- Measure the time taken to execute some code using a monotonic time source with your desired time unit.
- Mark a moment in time.
- Compare and find the difference between two moments in time.
- Check how much time has passed since a specific moment in time.
- Check whether the current time has passed a specific moment in time.

Measure code execution time

To measure the time taken to execute a block of code, use the `measureTime` inline function.

To measure the time taken to execute a block of code **and** return the result of the block of code, use the `measureTimedValue` inline function.

By default, both functions use a monotonic time source. However, if you want to use an elapsed real-time source, you can. For example, on Android the default time source `System.nanoTime()` only counts time while the device is active. It loses track of time when the device enters deep sleep. To keep track of time while the device is in deep sleep, you can create a time source that uses `SystemClock.elapsedRealtimeNanos()` instead:

```
object RealtimeMonotonicTimeSource : AbstractLongTimeSource(DurationUnit.NANOSECONDS) {
    override fun read(): Long = SystemClock.elapsedRealtimeNanos()
}
```

Mark and measure differences in time

To mark a specific moment in time, use the `TimeSource` interface and the `markNow()` function to create a `TimeMark`. To measure differences between `TimeMarks` from the same time source, use the subtraction operator (`-`):

```
import kotlin.time.*

fun main() {
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    Thread.sleep(500) // Sleep 0.5 seconds.
    val mark2 = timeSource.markNow()

    repeat(4) { n ->
        val mark3 = timeSource.markNow()
        val elapsed1 = mark3 - mark1
        val elapsed2 = mark3 - mark2

        println("Measurement 1.${n + 1}: elapsed1=$elapsed1, elapsed2=$elapsed2, diff=${elapsed1 -
elapsed2}")
    }
    // It's also possible to compare time marks with each other.
    println(mark2 > mark1) // This is true, as mark2 was captured later than mark1.
}
```

To check if a deadline has passed or a timeout has been reached, use the `hasPassedNow()` and `hasNotPassedNow()` extension functions:

```
import kotlin.time.*
import kotlin.time.Duration.Companion.seconds

fun main() {
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    val fiveSeconds: Duration = 5.seconds
    val mark2 = mark1 + fiveSeconds

    // It hasn't been 5 seconds yet
    println(mark2.hasPassedNow())
    // false

    // Wait six seconds
    Thread.sleep(6000)
    println(mark2.hasPassedNow())
    // true
}
```

The Kotlin/Native standard library's journey towards stabilization

As our standard library for Kotlin/Native continues to grow, we decided that it was time for a complete review to ensure that it meets our high standards. As part of this, we carefully reviewed **every** existing public signature. For each signature, we considered whether it:

- Has a unique purpose.
- Is consistent with other Kotlin APIs.
- Has similar behavior to its counterpart for the JVM.
- Is future-proof.

Based on these considerations, we made one of the following decisions:

- Made it Stable.
- Made it Experimental.
- Marked it as `private`.
- Modified its behavior.
- Moved it to a different location.
- Deprecated it.
- Marked it as obsolete.

If an existing signature has been:

- Moved to another package, then the signature still exists in the original package but it's now deprecated with deprecation level: `WARNING`. IntelliJ IDEA will automatically suggest replacements upon code inspection.
- Deprecated, then it's been deprecated with deprecation level: `WARNING`.
- Marked as obsolete, then you can keep using it, but it will be replaced in future.

We won't list all of the results of the review here, but here are some of the highlights:

- We stabilized the Atomics API.
- We made `kotlinx.cinterop` Experimental and now require different opt-ins for the package to be used. For more information, see Explicit C-interoperability stability guarantees.
- We marked the `Worker` class and its related APIs as obsolete.
- We marked the `BitSet` class as obsolete.
- We marked all `public` APIs in the `kotlin.native.internal` package as `private` or moved them to other packages.

Explicit C-interoperability stability guarantees

To maintain the high quality of our API, we decided to make `kotlinx.cinterop` Experimental. Although `kotlinx.cinterop` has been thoroughly tried and tested, there is still room for improvement before we are satisfied enough to make it Stable. We recommend that you use this API for interoperability but that you try to confine its use to specific areas in your projects. This will make your migration easier once we begin evolving this API to make it Stable.

If you want to use C-like foreign APIs such as pointers, you must opt in with `@OptIn(ExperimentalForeignApi)`, otherwise your code won't compile.

To use the remainder of `kotlinx.cinterop`, which covers Objective-C/Swift interoperability, you must opt in with `@OptIn(BetaInteropApi)`. If you try to use this API without the opt-in, your code will compile but the compiler will raise warnings that provide a clear explanation of what behavior you can expect.

For more information about these annotations, see our source code for `Annotations.kt`.

For more information on **all** of the changes as part of this review, see our YouTrack ticket.

We'd appreciate any feedback you might have! You can provide your feedback directly by commenting on the ticket.

Stable @Volatile annotation

If you annotate a `var` property with `@Volatile`, then the backing field is marked so that any reads or writes to this field are atomic, and writes are always made visible to other threads.

Prior to 1.8.20, the `kotlin.jvm.Volatile` annotation was available in the common standard library. However, this annotation was only effective on the JVM. If you used it on other platforms, it was ignored, which led to errors.

In 1.8.20, we introduced an experimental common annotation, `kotlin.concurrent.Volatile`, which you could preview in both the JVM and Kotlin/Native.

In 1.9.0, `kotlin.concurrent.Volatile` is Stable. If you use `kotlin.jvm.Volatile` in your multiplatform projects, we recommend that you migrate to `kotlin.concurrent.Volatile`.

New common function to get regex capture group by name

Prior to 1.9.0, every platform had its own extension to get a regular expression capture group by its name from a regular expression match. However there was no common function. It wasn't possible to have a common function prior to Kotlin 1.8.0, because the standard library still supported JVM targets 1.6 and 1.7.

As of Kotlin 1.8.0, the standard library is compiled with JVM target 1.8. So in 1.9.0, there is now a **common** `groups` function that you can use to retrieve a group's contents by its name for a regular expression match. This is useful when you want to access the results of regular expression matches belonging to a particular capture group.

Here is an example with a regular expression containing three capture groups: `city`, `state`, and `areaCode`. You can use these group names to access the matched values:

```
fun main() {
    val regex = """\b(?<city>[A-Za-z\s]+),\s(?<state>[A-Z]{2}):\s(?<areaCode>[0-9]{3})\b""".toRegex()
    val input = "Coordinates: Austin, TX: 123"

    val match = regex.find(input)!!
    println(match.groups["city"]?.value)
    // Austin
    println(match.groups["state"]?.value)
    // TX
    println(match.groups["areaCode"]?.value)
    // 123
}
```

New path utility to create parent directories

In 1.9.0 there is a new `createParentDirectories()` extension function that you can use to create a new file with all the necessary parent directories. When you provide a file path to `createParentDirectories()` it checks whether the parent directories already exist. If they do, it does nothing. However, if they do not, it creates them for you.

`createParentDirectories()` is particularly useful when you are copying files. For example, you can use it in combination with the `copyToRecursively()` function:

```
sourcePath.copyToRecursively(
   destinationPath.createParentDirectories(),
   followLinks = false
)
```

New HexFormat class to format and parse hexadecimals

> The new `HexFormat` class and its related extension functions are Experimental, and to use them, you can opt in with `@OptIn(ExperimentalStdlibApi::class)` or the compiler argument `-opt-in=kotlin.ExperimentalStdlibApi`.

In 1.9.0, the `HexFormat` class and its related extension functions are provided as an Experimental feature that allows you to convert between numerical values and hexadecimal strings. Specifically, you can use the extension functions to convert between hexadecimal strings and `ByteArrays` or other numeric types (`Int`, `Short`, `Long`).

For example:

```
println(93.toHexString()) // "0000005d"
```

The `HexFormat` class includes formatting options that you can configure with the `HexFormat{}` builder.

If you are working with `ByteArrays` you have the following options, which are configurable by properties:

Option	Description
`upperCase`	Whether hexadecimal digits are upper or lower case. By default, lower case is assumed. `upperCase = false`.
`bytes.bytesPerLine`	The maximum number of bytes per line.
`bytes.bytesPerGroup`	The maximum number of bytes per group.
`bytes.bytesSeparator`	The separator between bytes. Nothing by default.
`bytes.bytesPrefix`	The string that immediately precedes a two-digit hexadecimal representation of each byte, nothing by default.
`bytes.bytesSuffix`	The string that immediately succeeds a two-digit hexadecimal representation of each byte, nothing by default.

For example:

```
val macAddress = "001b638445e6".hexToByteArray()

// Use HexFormat{} builder to separate the hexadecimal string by colons
println(macAddress.toHexString(HexFormat { bytes.byteSeparator = ":" }))
// "00:1b:63:84:45:e6"

// Use HexFormat{} builder to:
// * Make the hexadecimal string uppercase
// * Group the bytes in pairs
// * Separate by periods
val threeGroupFormat = HexFormat { upperCase = true; bytes.bytesPerGroup = 2; bytes.groupSeparator =
"." }

println(macAddress.toHexString(threeGroupFormat))
// "001B.6384.45E6"
```

If you are working with numeric types, you have the following options, which are configurable by properties:

Option	Description
`number.prefix`	The prefix of a hexadecimal string, nothing by default.
`number.suffix`	The suffix of a hexadecimal string, nothing by default.
`number.removeLeadingZeros`	Whether to remove leading zeros in a hexadecimal string. By default, no leading zeros are removed. `number.removeLeadingZeros = false`

For example:

```
// Use HexFormat{} builder to parse a hexadecimal that has prefix: "0x".
println("0x3a".hexToInt(HexFormat { number.prefix = "0x" })) // "58"
```

Documentation updates

The Kotlin documentation has received some notable changes:

- The tour of Kotlin – Learn the fundamentals of the Kotlin programming language with chapters including both theory and practice.
- Android source set layout – Learn about the new Android source set layout.
- Compatibility guide for Kotlin Multiplatform – Learn about the incompatible changes you might encounter while developing projects with Kotlin Multiplatform.
- Kotlin Wasm – Learn about Kotlin/Wasm and how you can use it in your Kotlin Multiplatform projects.

Install Kotlin 1.9.0

Check the IDE version

IntelliJ IDEA 2022.3.3 and 2023.1.1 automatically suggest updating the Kotlin plugin to version 1.9.0. IntelliJ IDEA 2023.2 will include the Kotlin 1.9.0 plugin.

Android Studio Giraffe (223) and Hedgehog (231) will support Kotlin 1.9.0 in their upcoming releases.

The new command-line compiler is available for download on the GitHub release page.

Configure Gradle settings

To download Kotlin artifacts and dependencies, update your `settings.gradle(.kts)` file to use the Maven Central repository:

```
pluginManagement {
    repositories {
        mavenCentral()
        gradlePluginPortal()
    }
}
```

If the repository is not specified, Gradle uses the sunset JCenter repository, which could lead to issues with Kotlin artifacts.

Compatibility guide for Kotlin 1.9.0

Kotlin 1.9.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of these changes in the Compatibility guide for Kotlin 1.9.0.

Compatibility guide for Kotlin 1.9.x

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.8 to Kotlin 1.9.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Remove language version 1.3

Issue: KT-61111

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 introduces language version 1.9 and removes support for language version 1.3.

Deprecation cycle:

- 1.6.0: report a warning
- 1.9.0: raise the warning to an error

Prohibit super constructor call when the super interface type is a function literal

Issue: KT-46344

Component: Core language

Incompatible change type: source

Short summary: If an interface inherits from a function literal type, Kotlin 1.9 prohibits super constructor calls because no such constructor exists.

Deprecation cycle:

- 1.7.0: report a warning (or an error in progressive mode)
- 1.9.0: raise the warning to an error

Prohibit cycles in annotation parameter types

Issue: KT-47932

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 prohibits the type of an annotation being used as one of its parameter types, either directly or indirectly. This prevents cycles from being created. However, you are allowed to have parameter types that are an `Array` or a `vararg` of the annotation type.

Deprecation cycle:

- 1.7.0: report a warning (or an error in progressive mode) on cycles in types of annotation parameters
- 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitCyclesInAnnotations` can be used to temporarily revert to pre-1.9 behavior

Prohibit use of @ExtensionFunctionType annotation on function types with no parameters

Issue: KT-43527

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 prohibits using the `@ExtensionFunctionType` annotation on function types with no parameters, or on types that aren't function types.

Deprecation cycle:

- 1.7.0: report a warning for annotations on types that aren't function types, report an error for annotations on types that **are** function types
- 1.9.0: raise the warning for function types to an error

Prohibit Java field type mismatch on assignment

Issue: KT-48994

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.9 reports a compiler error if it detects that the type of a value assigned to a Java field doesn't match the Java field's projected type.

Deprecation cycle:

- 1.6.0: report a warning (or an error in the progressive mode) when a projected Java field type doesn't match the assigned value type
- 1.9.0: raise the warning to an error, `-XXLanguage:-RefineTypeCheckingOnAssignmentsToJavaFields` can be used to temporarily revert to pre-1.9 behavior

No source code excerpts in platform-type nullability assertion exceptions

Issue: KT-57570

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: In Kotlin 1.9, exception messages for expression null checks do not include source code excerpts. Instead, the name of the method or field is displayed. If the expression is not a method or field, there is no additional information provided in the message.

Deprecation cycle:

- < 1.9.0: exception messages generated by expression null checks contain source code excerpts
- 1.9.0: exception messages generated by expression null checks contain method or field names only, `-XXLanguage:-NoSourceCodeInNotNullAssertionExceptions` can be used to temporarily revert to pre-1.9 behavior

Prohibit the delegation of super calls to an abstract superclass member

Issues: KT-45508, KT-49017, KT-38078

Component: Core language

Incompatible change type: source

Short summary: Kotlin will report a compile error when an explicit or implicit super call is delegated to an *abstract* member of the superclass, even if there's a default implementation in a super interface.

Deprecation cycle:

- 1.5.20: introduce a warning when non-abstract classes that do not override all abstract members are used
- 1.7.0: report a warning if a super call, in fact, accesses an abstract member from a superclass
- 1.7.0: report an error in all affected cases if the `-Xjvm-default=all` or `-Xjvm-default=all-compatibility` compatibility modes are enabled; report an error in the progressive mode
- 1.8.0: report an error in cases of declaring a concrete class with a non-overridden abstract method from the superclass, and super calls of `Any` methods are overridden as abstract in the superclass
- 1.9.0: report an error in all affected cases, including explicit super calls to an abstract method from the super class

Deprecate confusing grammar in when-with-subject

Issue: KT-48385

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 deprecated several confusing grammar constructs in `when` condition expressions.

Deprecation cycle:

- 1.6.20: introduce a deprecation warning on the affected expressions
- 1.8.0: raise this warning to an error, `-XXLanguage:-ProhibitConfusingSyntaxInWhenBranches` can be used to temporarily revert to the pre-1.8 behavior
- >= 2.1: repurpose some deprecated constructs for new language features

Prevent implicit coercions between different numeric types

Issue: KT-48645

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin will avoid converting numeric values automatically to a primitive numeric type where only a downcast to that type is needed semantically.

Deprecation cycle:

- < 1.5.30: the old behavior in all affected cases
- 1.5.30: fix the downcast behavior in generated property delegate accessors, `-Xuse-old-backend` can be used to temporarily revert to the pre-1.5.30 fix behavior
- >= 2.0: fix the downcast behavior in other affected cases

Prohibit upper bound violation in a generic type alias usage (a type parameter used in a generic type argument of a type argument of the aliased type)

Issue: KT-54066

Component: Core language

Incompatible change type: source

Short summary: Kotlin will prohibit using a type alias with type arguments that violate the upper bound restrictions of the corresponding type parameters of the aliased type in case when the typealias type parameter is used as a generic type argument of a type argument of the aliased type, for example, `typealias Alias<T> = Base<List<T>>`.

Deprecation cycle:

- 1.8.0: report a warning when a generic typealias usage has type arguments violating upper bound constraints of the corresponding type parameters of the aliased type
- 2.0.0: raise the warning to an error

Keep nullability when approximating local types in public signatures

Issue: KT-53982

Component: Core language

Incompatible change type: source, binary

Short summary: when a local or anonymous type is returned from an expression-body function without an explicitly specified return type, Kotlin compiler infers (or approximates) the return type using the known supertype of that type. During this, the compiler can infer a non-nullable type where the null value could in fact be returned.

Deprecation cycle:

- 1.8.0: approximate flexible types by flexible supertypes
- 1.8.0: report a warning when a declaration is inferred to have a non-nullable type that should be nullable, prompting users to specify the type explicitly
- 2.0.0: approximate nullable types by nullable supertypes, `-XXLanguage:-KeepNullabilityWhenApproximatingLocalType` can be used to temporarily revert to the pre-2.0 behavior

Do not propagate deprecation through overrides

Issue: KT-47902

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will no longer propagate deprecation from a deprecated member in the superclass to its overriding member in the subclass, thus providing an explicit mechanism for deprecating a member of the superclass while leaving it non-deprecated in the subclass.

Deprecation cycle:

- 1.6.20: reporting a warning with the message of the future behavior change and a prompt to either suppress this warning or explicitly write a `@Deprecated` annotation on an override of a deprecated member
- 1.9.0: stop propagating deprecation status to the overridden members. This change also takes effect immediately in the progressive mode

Prohibit using collection literals in annotation classes anywhere except their parameters declaration

Issue: KT-39041

Component: Core language

Incompatible change type: source

Short summary: Kotlin allows using collection literals in a restricted way - for passing arrays to parameters of annotation classes or specifying default values for these parameters. However besides that, Kotlin allowed using collections literals anywhere else inside an annotation class, for example, in its nested object. Kotlin 1.9 will prohibit using collection literals in annotation classes anywhere except their parameters' default values.

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) on array literals in nested objects in annotation classes
- 1.9.0: raise the warning to an error

Prohibit forward referencing of parameters in default value expressions

Issue: KT-25694

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit forward referencing of parameters in default value expressions of other parameters. This ensures that by the time the parameter is accessed in a default value expression, it would already have a value either passed to the function or initialized by its own default value expression.

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) when a parameter with default value is references in default value of another parameter that comes before it
- 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitIllegalValueParameterUsageInDefaultArguments` can be used to temporarily revert to the pre-1.9 behavior

Prohibit extension calls on inline functional parameters

Issue: KT-52502

Component: Core language

Incompatible change type: source

Short summary: while Kotlin allowed passing an inline functional parameter to another inline function as a receiver, it always resulted in compiler exceptions when compiling such code. Kotlin 1.9 will prohibit this, thus reporting an error instead of crashing the compiler.

Deprecation cycle:

- 1.7.20: report a warning (or an error in the progressive mode) for inline extension calls on inline functional parameters
- 1.9.0: raise the warning to an error

Prohibit calls to infix functions named suspend with an anonymous function argument

Issue: KT-49264

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will no longer allow calling infix functions named `suspend` that have a single argument of a functional type passed as an anonymous function literal.

Deprecation cycle:

- 1.7.20: report a warning on suspend infix calls with an anonymous function literal
- 1.9.0: raise the warning to an error, `-XXLanguage:-ModifierNonBuiltinSuspendFunError` can be used to temporarily revert to the pre-1.9 behavior
- TODO: Change how the `suspend fun` token sequence is interpreted by the parser

Prohibit using captured type parameters in inner classes against their variance

Issue: KT-50947

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit using type parameters of an outer class having `in` or `out` variance in an inner class of that class in positions violating that type parameters' declared variance.

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) when an outer class' type parameter usage position violates the variance rules of that parameter
- 1.9.0: raise the warning to an error, `-XXLanguage:-ReportTypeVarianceConflictOnQualifierArguments` can be used to temporarily revert to the pre-1.9 behavior

Prohibit recursive call of a function without explicit return type in compound assignment operators

Issue: KT-48546

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit calling a function without explicitly specified return type in an argument of a compound assignment operator inside that function's body, as it currently does in other expressions inside the body of that function.

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) when a function without explicitly specified return type is called recursively in that function's body in a compound assignment operator argument
- 1.9.0: raise the warning to an error

Prohibit unsound calls with expected @NotNull T and given Kotlin generic parameter with nullable bound

Issue: KT-36770

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit method calls where a value of a potentially nullable generic type is passed for a `@NotNull`-annotated parameter of a Java method.

Deprecation cycle:

- 1.5.20: report a warning when an unconstrained generic type parameter is passed where a non-nullable type is expected
- 1.9.0: report a type mismatch error instead of the warning above, `-XXLanguage:-ProhibitUsingNullableTypeParameterAgainstNotNullAnnotated` can be used to temporarily revert to the pre-1.8 behavior

Prohibit access to members of a companion of an enum class from entry initializers of this enum

Issue: KT-49110

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit all kinds of access to the companion object of an enum from an enum entry initializer.

Deprecation cycle:

- 1.6.20: report a warning (or an error in the progressive mode) on such companion member access
- 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitAccessToEnumCompanionMembersInEnumConstructorCall` can be used to temporarily revert to the pre-1.8 behavior

Deprecate and remove Enum.declaringClass synthetic property

Issue: KT-49653

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin allowed using the synthetic property `declaringClass` on `Enum` values produced from the method `getDeclaringClass()` of the underlying Java class `java.lang.Enum` even though this method is not available for Kotlin `Enum` type. Kotlin 1.9 will prohibit using this property, proposing to migrate to the extension property `declaringJavaClass` instead.

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) on `declaringClass` property usages, propose the migration to `declaringJavaClass` extension
- 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitEnumDeclaringClass` can be used to temporarily revert to the pre-1.9 behavior
- 2.0.0: remove `declaringClass` synthetic property

Deprecate enable and compatibility modes of the compiler option -Xjvm-default

Issues: KT-46329, KT-54746

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.9 prohibits using the `enable` and `compatibility` modes of the `-Xjvm-default` compiler option.

Deprecation cycle:

- 1.6.20: introduce a warning on the `enable` and `compatibility` modes of the `-Xjvm-default` compiler option
- 1.9.0: raise this warning to an error

Prohibit implicit inferring a type variable into an upper bound in the builder inference context

Issue: KT-47986

Component: Core language

Incompatible change type: source

Short summary: Kotlin 2.0 will prohibit inferring a type variable into the corresponding type parameter's upper bound in the absence of any use-site type information in the scope of builder inference lambda functions, the same way as it does currently in other contexts.

Deprecation cycle:

- 1.7.20: report a warning (or an error in the progressive mode) when a type parameter is inferred into declared upper bounds in the absence of use-site type information
- 2.0.0: raise the warning to an error

Standard library

Warn about potential overload resolution change when Range/Progression starts implementing Collection

Issue: KT-49276

Component: Core language / kotlin-stdlib

Incompatible change type: source

Short summary: it is planned to implement the `Collection` interface in the standard progressions and concrete ranges inherited from them in Kotlin 1.9. This could make a different overload selected in the overload resolution if there are two overloads of some method, one accepting an element and another accepting a collection. Kotlin will make this situation visible by reporting a warning or an error when such overloaded method is called with a range or progression argument.

Deprecation cycle:

- 1.6.20: report a warning when an overloaded method is called with the standard progression or its range inheritor as an argument if implementing the `Collection` interface by this progression/range leads to another overload being selected in this call in future
- 1.8.0: raise this warning to an error
- 2.1.0: stop reporting the error, implement `Collection` interface in progressions thus changing the overload resolution result in the affected cases

Migrate declarations from kotlin.dom and kotlin.browser packages to kotlinx.*

Issue: KT-39330

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: declarations from the `kotlin.dom` and `kotlin.browser` packages are moved to the corresponding `kotlinx.*` packages to prepare for extracting them from stdlib.

Deprecation cycle:

- 1.4.0: introduce the replacement API in `kotlinx.dom` and `kotlinx.browser` packages
- 1.4.0: deprecate the API in `kotlin.dom` and `kotlin.browser` packages and propose the new API above as a replacement
- 1.6.0: raise the deprecation level to an error
- 1.8.20: remove the deprecated functions from stdlib for JS-IR target
- >= 2.0: move the API in kotlinx.* packages to a separate library

Deprecate some JS-only API

Issue: KT-48587

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: a number of JS-only functions in stdlib are deprecated for removal. They include: `String.concat(String)`, `String.match(regex: String)`, `String.matches(regex: String)`, and the `sort` functions on arrays taking a comparison function, for example, `Array<out T>.sort(comparison: (a: T, b: T) -> Int)`.

Deprecation cycle:

- 1.6.0: deprecate the affected functions with a warning
- 1.9.0: raise the deprecation level to an error
- >=2.0: remove the deprecated functions from the public API

Tools

Remove enableEndorsedLibs flag from Gradle setup

Issue: KT-54098

Component: Gradle

Incompatible change type: source

Short summary: the `enableEndorsedLibs` flag is no longer supported in Gradle setup.

Deprecation cycle:

- < 1.9.0: `enableEndorsedLibs` flag is supported in Gradle setup
- 1.9.0: `enableEndorsedLibs` flag is **not** supported in Gradle setup

Remove Gradle conventions

Issue: KT-52976

Component: Gradle

Incompatible change type: source

Short summary: Gradle conventions were deprecated in Gradle 7.1 and have been removed in Gradle 8.

Deprecation cycle:

- 1.7.20: Gradle conventions deprecated
- 1.9.0: Gradle conventions removed

Remove classpath property of KotlinCompile task

Issue: KT-53748

Component: Gradle

Incompatible change type: source

Short summary: the `classpath` property of the `KotlinCompile` task is removed.

Deprecation cycle:

- 1.7.0: the `classpath` property is deprecated
- 1.8.0: raise the deprecation level to an error
- 1.9.0: remove the deprecated functions from the public API

Deprecate kotlin.internal.single.build.metrics.file property

Issue: KT-53357

Component: Gradle

Incompatible change type: source

Short summary: deprecate the `kotlin.internal.single.build.metrics.file` property used to define a single file for build reports. Use the property `kotlin.build.report.single_file` instead with `kotlin.build.report.output=single_file`.

Deprecation cycle:

- 1.8.0: raise the deprecation level to a warning
- >= 1.9: delete the property

What's new in Kotlin 1.8.20

Released: 25 April 2023

The Kotlin 1.8.20 release is out and here are some of its biggest highlights:

- New Kotlin K2 compiler updates
- New experimental Kotlin/Wasm target
- New JVM incremental compilation by default in Gradle
- Update for Kotlin/Native targets
- Preview of Gradle composite builds in Kotlin Multiplatform
- Improved output for Gradle errors in Xcode
- Experimental support for the AutoCloseable interface in the standard library
- Experimental support for Base64 encoding in the standard library

You can also find a short overview of the changes in this video:

Watch video online.

For information about the Kotlin release cycle, see Kotlin release process.

IDE support

The Kotlin plugins that support 1.8.20 are available for:

IDE	Supported versions
IntelliJ IDEA	2022.2.x, 2022.3.x, 2023.1.x

IDE	Supported versions
Android Studio	Flamingo (222)

To download Kotlin artifacts and dependencies properly, configure Gradle settings to use the Maven Central repository.

New Kotlin K2 compiler updates

The Kotlin team continues to stabilize the K2 compiler. As mentioned in the Kotlin 1.7.0 announcement, it's still in **Alpha**. This release introduces further improvements on the road to K2 Beta.

Starting with this 1.8.20 release, the Kotlin K2 compiler:

- Has a preview version of the serialization plugin.
- Provides Alpha support for the JS IR compiler.
- Introduces the future release of the new language version, Kotlin 2.0.

Learn more about the new compiler and its benefits in the following videos:

- What Everyone Must Know About The NEW Kotlin K2 Compiler
- The New Kotlin K2 Compiler: Expert Review

How to enable the Kotlin K2 compiler

To enable and test the Kotlin K2 compiler, use the new language version with the following compiler option:

```
-language-version 2.0
```

You can specify it in your `build.gradle(.kts)` file:

```
kotlin {
   sourceSets.all {
       languageSettings {
           languageVersion = "2.0"
       }
   }
}
```

The previous `-Xuse-k2` compiler option has been deprecated.

The Alpha version of the new K2 compiler only works with JVM and JS IR projects. It doesn't support Kotlin/Native or any of the multiplatform projects yet.

Leave your feedback on the new K2 compiler

We would appreciate any feedback you may have!

- Provide your feedback directly to K2 developers on Kotlin Slack – get an invite and join the #k2-early-adopters channel.

- Report any problems you faced with the new K2 compiler on our issue tracker.
- Enable the **Send usage statistics** option to allow JetBrains to collect anonymous data about K2 usage.

Language

As Kotlin continues to evolve, we're introducing preview versions for new language features in 1.8.20:

- A modern and performant replacement of the Enum class values function
- Data objects for symmetry with data classes
- Lifting restrictions on secondary constructors with bodies in inline classes

A modern and performant replacement of the Enum class values function

This feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Enum classes have a synthetic `values()` function, which returns an array of defined enum constants. However, using an array can lead to hidden performance issues in Kotlin and Java. In addition, most of the APIs use collections, which require eventual conversion. To fix these problems, we've introduced the `entries` property for Enum classes, which should be used instead of the `values()` function. When called, the `entries` property returns a pre-allocated immutable list of defined enum constants.

The `values()` function is still supported, but we recommend that you use the `entries` property instead.

```
enum class Color(val colorName: String, val rgb: String) {
    RED("Red", "#FF0000"),
    ORANGE("Orange", "#FF7F00"),
    YELLOW("Yellow", "#FFFF00")
}

@OptIn(ExperimentalStdlibApi::class)
fun findByRgb(rgb: String): Color? = Color.entries.find { it.rgb == rgb }
```

How to enable the entries property

To try this feature out, opt in with `@OptIn(ExperimentalStdlibApi)` and enable the `-language-version 1.9` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)` file:

Kotlin

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
    }
```

Starting with IntelliJ IDEA 2023.1, if you have opted in to this feature, the appropriate IDE inspection will notify you about converting from `values()` to `entries` and offer a quick-fix.

For more information on the proposal, see the KEEP note.

Preview of data objects for symmetry with data classes

Data objects allow you to declare objects with singleton semantics and a clean `toString()` representation. In this snippet, you can see how adding the `data` keyword to an object declaration improves the readability of its `toString()` output:

```
package org.example
object MyObject
data object MyDataObject

fun main() {
    println(MyObject) // org.example.MyObject@1f32e575
    println(MyDataObject) // MyDataObject
}
```

Especially for `sealed` hierarchies (like a `sealed class` or `sealed interface` hierarchy), `data objects` are an excellent fit because they can be used conveniently alongside `data class` declarations. In this snippet, declaring `EndOfFile` as a `data object` instead of a plain `object` means that it will get a pretty `toString` without the need to override it manually. This maintains symmetry with the accompanying data class definitions.

```
sealed interface ReadResult
data class Number(val number: Int) : ReadResult
data class Text(val text: String) : ReadResult
data object EndOfFile : ReadResult

fun main() {
    println(Number(7)) // Number(number=7)
    println(EndOfFile) // EndOfFile
}
```

Semantics of data objects

Since their first preview version in Kotlin 1.7.20, the semantics of data objects have been refined. The compiler now automatically generates a number of convenience functions for them:

toString

The `toString()` function of a data object returns the simple name of the object:

```
data object MyDataObject {
    val x: Int = 3
}

fun main() {
    println(MyDataObject) // MyDataObject
}
```

equals and hashCode

The `equals()` function for a `data object` ensures that all objects that have the type of your `data object` are considered equal. In most cases, you will only have a single instance of your data object at runtime (after all, a `data object` declares a singleton). However, in the edge case where another object of the same type is generated at runtime (for example, via platform reflection through `java.lang.reflect`, or by using a JVM serialization library that uses this API under the hood), this ensures that the objects are treated as equal.

Make sure to only compare `data objects` structurally (using the `==` operator) and never by reference (the `===` operator). This helps avoid pitfalls when more than one instance of a data object exists at runtime. The following snippet illustrates this specific edge case:

```
import java.lang.reflect.Constructor

data object MySingleton

fun main() {
    val evilTwin = createInstanceViaReflection()

    println(MySingleton) // MySingleton
    println(evilTwin) // MySingleton

    // Even when a library forcefully creates a second instance of MySingleton, its `equals` method
returns true:
    println(MySingleton == evilTwin) // true

    // Do not compare data objects via ===.
    println(MySingleton === evilTwin) // false
}

fun createInstanceViaReflection(): MySingleton {
    // Kotlin reflection does not permit the instantiation of data objects.
    // This creates a new MySingleton instance "by force" (i.e., Java platform reflection)
    // Don't do this yourself!
    return (MySingleton.javaClass.declaredConstructors[0].apply { isAccessible = true } as
Constructor<MySingleton>).newInstance()
}
```

The behavior of the generated `hashCode()` function is consistent with that of the `equals()` function, so that all runtime instances of a `data object` have the same hash code.

No copy and componentN functions for data objects

While `data object` and `data class` declarations are often used together and have some similarities, there are some functions that are not generated for a `data object`:

Because a `data object` declaration is intended to be used as a singleton object, no `copy()` function is generated. The singleton pattern restricts the instantiation of a class to a single instance, and allowing copies of the instance to be created would violate that restriction.

Also, unlike a `data class`, a `data object` does not have any data properties. Since attempting to destructure such an object would not make sense, no `componentN()` functions are generated.

We would appreciate your feedback on this feature in YouTrack.

How to enable the data objects preview

To try this feature out, enable the `-language-version 1.9` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)` file:

Kotlin

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
    }
```

Preview of lifting restriction on secondary constructors with bodies in inline classes

This feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 1.8.20 lifts restrictions on the use of secondary constructors with bodies in inline classes.

Inline classes used to allow only a public primary constructor without `init` blocks or secondary constructors to have clear initialization semantics. As a result, it was impossible to encapsulate underlying values or create an inline class that would represent some constrained values.

These issues were fixed when Kotlin 1.4.30 lifted restrictions on `init` blocks. Now we're taking it a step further and allowing secondary constructors with bodies in preview mode:

```
@JvmInline
value class Person(private val fullName: String) {
    // Allowed since Kotlin 1.4.30:
    init {
        check(fullName.isNotBlank()) {
            "Full name shouldn't be empty"
        }
    }

    // Preview available since Kotlin 1.8.20:
    constructor(name: String, lastName: String) : this("$name $lastName") {
        check(lastName.isNotBlank()) {
            "Last name shouldn't be empty"
        }
    }
}
```

How to enable secondary constructors with bodies

To try this feature out, enable the `-language-version 1.9` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)`:

Kotlin

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
    }
```

We encourage you to try this feature out and submit all reports in YouTrack to help us make it the default in Kotlin 1.9.0.

Learn more about the development of Kotlin inline classes in this KEEP.

New Kotlin/Wasm target

Kotlin/Wasm (Kotlin WebAssembly) goes Experimental in this release. The Kotlin team finds WebAssembly to be a promising technology and wants to find better ways for you to use it and get all of the benefits of Kotlin.

WebAssembly binary format is independent of the platform because it runs using its own virtual machine. Almost all modern browsers already support WebAssembly 1.0. To set up the environment to run WebAssembly, you only need to enable an experimental garbage collection mode that Kotlin/Wasm targets. You can find detailed instructions here: How to enable Kotlin/Wasm.

We want to highlight the following advantages of the new Kotlin/Wasm target:

- Faster compilation speed compared to the `wasm32` Kotlin/Native target, since Kotlin/Wasm doesn't have to use LLVM.
- Easier interoperability with JS and integration with browsers compared to the `wasm32` target, thanks to the Wasm garbage collection.
- Potentially faster application startup compared to Kotlin/JS and JavaScript because Wasm has a compact and easy-to-parse bytecode.
- Improved application runtime performance compared to Kotlin/JS and JavaScript because Wasm is a statically typed language.

Starting with the 1.8.20 release, you can use Kotlin/Wasm in your experimental projects. We provide the Kotlin standard library (`stdlib`) and test library (`kotlin.test`) for Kotlin/Wasm out of the box. IDE support will be added in future releases.

Learn more about Kotlin/Wasm in this YouTube video.

How to enable Kotlin/Wasm

To enable and test Kotlin/Wasm, update your `build.gradle.kts` file:

```
plugins {
    kotlin("multiplatform") version "1.8.20"
}

kotlin {
    wasm {
        binaries.executable()
        browser {
        }
    }
    sourceSets {
        val commonMain by getting
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test"))
            }
        }
        val wasmMain by getting
        val wasmTest by getting
    }
}
```

Check out the GitHub repository with Kotlin/Wasm examples.

To run a Kotlin/Wasm project, you need to update the settings of the target environment:

Chrome

- For version 109:

  Run the application with the `--js-flags=--experimental-wasm-gc` command line argument.
- For version 110 or later:
  1. Go to `chrome://flags/#enable-webassembly-garbage-collection` in your browser.
  2. Enable **WebAssembly Garbage Collection**.
  3. Relaunch your browser.

Firefox

For version 109 or later:

1. Go to `about:config` in your browser.
2. Enable `javascript.options.wasm_function_references` and `javascript.options.wasm_gc` options.
3. Relaunch your browser.

Edge

For version 109 or later:

Run the application with the `--js-flags=--experimental-wasm-gc` command line argument.

Leave your feedback on Kotlin/Wasm

We would appreciate any feedback you may have!

- Provide your feedback directly to developers in Kotlin Slack – get an invite and join the #webassembly channel.
- Report any problems you faced with Kotlin/Wasm on this YouTrack issue.

Kotlin/JVM

Kotlin 1.8.20 introduces a preview of Java synthetic property references and support for the JVM IR backend in the kapt stub generating task by default.

Preview of Java synthetic property references

> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 1.8.20 introduces the ability to create references to Java synthetic properties, for example, for such Java code:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Kotlin has always allowed you to write `person.age`, where `age` is a synthetic property. Now, you can also create references to `Person::age` and `person::age`. All the same works for `name`, as well.

```
val persons = listOf(Person("Jack", 11), Person("Sofie", 12), Person("Peter", 11))
    persons
        // Call a reference to Java synthetic property:
        .sortedBy(Person::age)
        // Call Java getter via the Kotlin property syntax:
        .forEach { person -> println(person.name) }
```

How to enable Java synthetic property references

To try this feature out, enable the `-language-version 1.9` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)`:

Kotlin

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
    }
```

Support for the JVM IR backend in kapt stub generating task by default

In Kotlin 1.7.20, we introduced support for the JVM IR backend in the kapt stub generating task. Starting with this release, this support works by default. You no longer need to specify `kapt.use.jvm.ir=true` in your `gradle.properties` to enable it. We would appreciate your feedback on this feature in YouTrack.

Kotlin/Native

Kotlin 1.8.20 includes changes to supported Kotlin/Native targets, interoperability with Objective-C, and improvements to the CocoaPods Gradle plugin, among other updates:

- Update for Kotlin/Native targets
- Deprecation of the legacy memory manager
- Support for Objective-C headers with @import directives
- Support for link-only mode in the Cocoapods Gradle plugin
- Import Objective-C extensions as class members in UIKit
- Reimplementation of compiler cache management in the compiler
- Deprecation of `useLibraries()` in Cocoapods Gradle plugin

Update for Kotlin/Native targets

The Kotlin team decided to revisit the list of targets supported by Kotlin/Native, split them into tiers, and deprecate some of them starting with Kotlin 1.8.20. See the Kotlin/Native target support section for the full list of supported and deprecated targets.

The following targets have been deprecated with Kotlin 1.8.20 and will be removed in 1.9.20:

- `iosArm32`
- `watchosX86`
- `wasm32`
- `mingwX86`
- `linuxArm32Hfp`
- `linuxMips32`
- `linuxMipsel32`

As for the remaining targets, there are now three tiers of support depending on how well a target is supported and tested in the Kotlin/Native compiler. A target can be moved to a different tier. For example, we'll do our best to provide full support for `iosArm64` in the future, as it is important for Kotlin Multiplatform.

If you're a library author, these target tiers can help you decide which targets to test on CI tools and which ones to skip. The Kotlin team will use the same approach when developing official Kotlin libraries, like kotlinx.coroutines.

Check out our blog post to learn more about the reasons for these changes.

Deprecation of the legacy memory manager

Starting with 1.8.20, the legacy memory manager is deprecated and will be removed in 1.9.20. The new memory manager was enabled by default in 1.7.20 and has been receiving further stability updates and performance improvements.

If you're still using the legacy memory manager, remove the `kotlin.native.binary.memoryModel=strict` option from your `gradle.properties` and follow our Migration guide to make the necessary changes.

The new memory manager doesn't support the `wasm32` target. This target is also deprecated starting with this release and will be removed in 1.9.20.

Support for Objective-C headers with @import directives

> This feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin/Native can now import Objective-C headers with `@import` directives. This feature is useful for consuming Swift libraries that have auto-generated Objective-C headers or classes of CocoaPods dependencies written in Swift.

Previously, the cinterop tool failed to analyze headers that depended on Objective-C modules via the `@import` directive. The reason was that it lacked support for the `-fmodules` option.

Starting with Kotlin 1.8.20, you can use Objective-C headers with `@import`. To do so, pass the `-fmodules` option to the compiler in the definition file as `compilerOpts`. If you use CocoaPods integration, specify the cinterop option in the configuration block of the `pod()` function like this:

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("PodName") {
            extraOpts = listOf("-compiler-option", "-fmodules")
        }
    }
}
```

This was a highly awaited feature, and we welcome your feedback about it in YouTrack to help us make it the default in future releases.

Support for the link-only mode in Cocoapods Gradle plugin

With Kotlin 1.8.20, you can use Pod dependencies with dynamic frameworks only for linking, without generating cinterop bindings. This may come in handy when cinterop bindings are already generated.

Consider a project with 2 modules, a library and an app. The library depends on a Pod but doesn't produce a framework, only a `.klib`. The app depends on the library and produces a dynamic framework. In this case, you need to link this framework with the Pods that the library depends on, but you don't need cinterop bindings because they are already generated for the library.

To enable the feature, use the `linkOnly` option or a builder property when adding a dependency on a Pod:

```
cocoapods {
    summary = "CocoaPods test library"
    homepage = "https://github.com/JetBrains/kotlin"

    pod("Alamofire", linkOnly = true) {
        version = "5.7.0"
    }
}
```

If you use this option with static frameworks, it will remove the Pod dependency entirely because Pods are not used for static framework linking.

Import Objective-C extensions as class members in UIKit

Since Xcode 14.1, some methods from Objective-C classes have been moved to category members. That led to the generation of a different Kotlin API, and these methods were imported as Kotlin extensions instead of methods.

You may have experienced issues resulting from this when overriding methods using UIKit. For example, it became impossible to override `drawRect()` or `layoutSubviews()` methods when subclassing a UIView in Kotlin.

Since 1.8.20, category members that are declared in the same headers as NSView and UIView classes are imported as members of these classes. This means that the methods subclassing from NSView and UIView can be easily overridden, like any other method.

If everything goes well, we're planning to enable this behavior by default for all of the Objective-C classes.

Reimplementation of compiler cache management in the compiler

To speed up the evolution of compiler caches, we've moved compiler cache management from the Kotlin Gradle plugin to the Kotlin/Native compiler. This unblocks work on several important improvements, including those to do with compilation times and compiler cache flexibility.

If you encounter some problem and need to return to the old behavior, use the `kotlin.native.cacheOrchestration=gradle` Gradle property.

We would appreciate your feedback on this in YouTrack.

Deprecation of useLibraries() in Cocoapods Gradle plugin

Kotlin 1.8.20 starts the deprecation cycle of the `useLibraries()` function used in the CocoaPods integration for static libraries.

We introduced the `useLibraries()` function to allow dependencies on Pods containing static libraries. With time, this case has become very rare. Most of the Pods are distributed by sources, and Objective-C frameworks or XCFrameworks are a common choice for binary distribution.

Since this function is unpopular and it creates issues that complicate the development of the Kotlin CocoaPods Gradle plugin, we've decided to deprecate it.

For more information on frameworks and XCFrameworks, see Build final native binaries.

Kotlin Multiplatform

Kotlin 1.8.20 strives to improve the developer experience with the following updates to Kotlin Multiplatform:

- New approach for setting up source set hierarchy
- Preview of Gradle composite builds support in Kotlin Multiplatform
- Improved output for Gradle errors in Xcode

New approach to source set hierarchy

> The new approach to source set hierarchy is Experimental. It may be changed in future Kotlin releases without prior notice. Opt-in is required (see the details below). We would appreciate your feedback in YouTrack.

Kotlin 1.8.20 offers a new way of setting up source set hierarchy in your multiplatform projects - the default target hierarchy. The new approach is intended to replace target shortcuts like `ios`, which have their design flaws.

The idea behind the default target hierarchy is simple: You explicitly declare all the targets to which your project compiles, and the Kotlin Gradle plugin automatically creates shared source sets based on the specified targets.

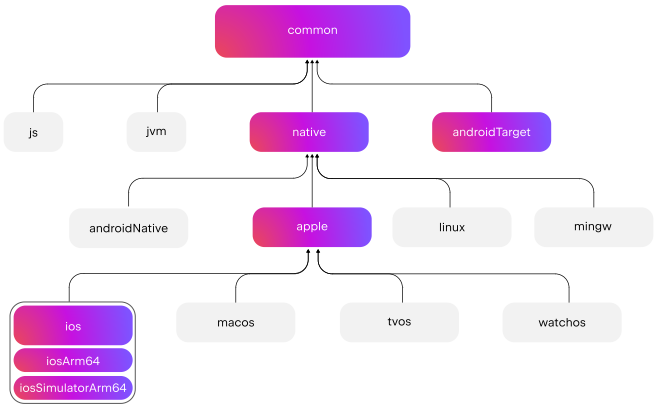
Set up your project

Consider this example of a simple multiplatform mobile app:

```
@OptIn(ExperimentalKotlinGradlePluginApi::class)
kotlin {
    // Enable the default target hierarchy:
    targetHierarchy.default()

    android()
    iosArm64()
    iosSimulatorArm64()
}
```

You can think of the default target hierarchy as a template for all possible targets and their shared source sets. When you declare the final targets `android`, `iosArm64`, and `iosSimulatorArm64` in your code, the Kotlin Gradle plugin finds suitable shared source sets from the template and creates them for you. The resulting hierarchy looks like this:



An example of using the default target hierarchy

Green source sets are actually created and present in the project, while gray ones from the default template are ignored. As you can see, the Kotlin Gradle plugin hasn't created the `watchos` source set, for example, because there are no watchOS targets in the project.

If you add a watchOS target, such as `watchosArm64`, the `watchos` source set is created, and the code from the `apple`, `native`, and `common` source sets is compiled to `watchosArm64`, as well.

You can find the complete scheme for the default target hierarchy in the documentation.

In this example, the `apple` and `native` source sets compile only to the `iosArm64` and `iosSimulatorArm64` targets. Therefore, despite their names, they have access to the full iOS API. This might be counter-intuitive for source sets like `native`, as you may expect that only APIs available on all native targets are accessible in this source set. This behavior may change in the future.

Why replace shortcuts

Creating source sets hierarchies can be verbose, error-prone, and unfriendly for beginners. Our previous solution was to introduce shortcuts like `ios` that create a part of the hierarchy for you. However, working with shortcuts proved they have a big design flaw: they're difficult to change.

Take the `ios` shortcut, for example. It creates only the `iosArm64` and `iosX64` targets, which can be confusing and may lead to issues when working on an M1-based host that requires the `iosSimulatorArm64` target as well. However, adding the `iosSimulatorArm64` target can be a very disruptive change for user projects:

- All dependencies used in the `iosMain` source set have to support the `iosSimulatorArm64` target; otherwise, the dependency resolution fails.
- Some native APIs used in `iosMain` may disappear when adding a new target (though this is unlikely in the case of `iosSimulatorArm64`).
- In some cases, such as when writing a small pet project on your Intel-based MacBook, you might not even need this change.

It became clear that shortcuts didn't solve the problem of configuring hierarchies, which is why we stopped adding new shortcuts at some point.

The default target hierarchy may look similar to shortcuts at first glance, but they have a crucial distinction: **users have to explicitly specify the set of targets**. This set defines how your project is compiled and published and how it participates in dependency resolution. Since this set is fixed, changes to the default configuration from the Kotlin Gradle plugin should cause significantly less distress in the ecosystem, and it will be much easier to provide tooling-assisted migration.

How to enable the default hierarchy

This new feature is Experimental. For Kotlin Gradle build scripts, you need to opt in with `@OptIn(ExperimentalKotlinGradlePluginApi::class)`.

For more information, see Hierarchical project structure.

Leave feedback

This is a significant change to multiplatform projects. We would appreciate your feedback to help make it even better.

Preview of Gradle composite builds support in Kotlin Multiplatform

> This feature has been supported in Gradle builds since Kotlin Gradle Plugin 1.8.20. For IDE support, use IntelliJ IDEA 2023.1 Beta 2 (231.8109.2) or later and the Kotlin Gradle plugin 1.8.20 with any Kotlin IDE plugin.

Starting with 1.8.20, Kotlin Multiplatform supports Gradle composite builds. Composite builds allow you to include builds of separate projects or parts of the same project into a single build.

Due to some technical challenges, using Gradle composite builds with Kotlin Multiplatform was only partially supported. Kotlin 1.8.20 contains a preview of the improved support that should work with a larger variety of projects. To try it out, add the following option to your `gradle.properties`:

```
kotlin.mpp.import.enableKgpDependencyResolution=true
```

This option enables a preview of the new import mode. Besides the support for composite builds, it provides a smoother import experience in multiplatform projects, as we've included major bug fixes and improvements to make the import more stable.

Known issues

It's still a preview version that needs further stabilization, and you might encounter some issues with import along the way. Here are some known issues we're planning to fix before the final release of Kotlin 1.8.20:

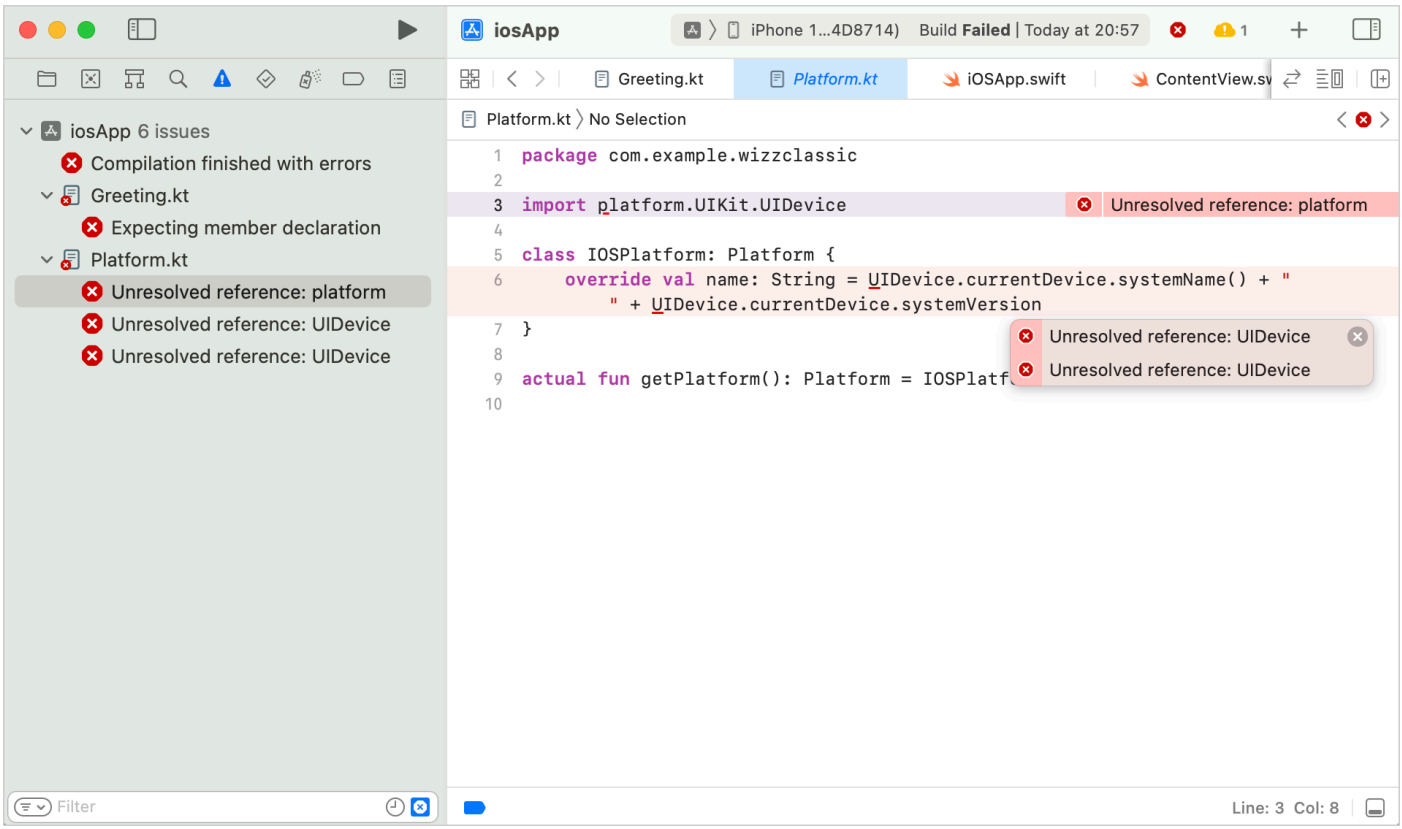
- There's no Kotlin 1.8.20 plugin available for IntelliJ IDEA 2023.1 EAP yet. Despite that, you can still set the Kotlin Gradle plugin version to 1.8.20 and try out composite builds in this IDE.
- If your projects include builds with a specified `rootProject.name`, composite builds may fail to resolve the Kotlin metadata. For the workaround and details, see this Youtrack issue.

We encourage you to try it out and submit all reports on YouTrack to help us make it the default in Kotlin 1.9.0.

Improved output for Gradle errors in Xcode

If you had issues building your multiplatform projects in Xcode, you might have encountered a "Command PhaseScriptExecution failed with a nonzero exit code" error. This message signals that the Gradle invocation has failed, but it's not very helpful when trying to detect the problem.

Starting with Kotlin 1.8.20, Xcode can parse the output from the Kotlin/Native compiler. Furthermore, in case the Gradle build fails, you'll see an additional error message from the root cause exception in Xcode. In most cases, it'll help to identify the root problem.



Improved output for Gradle errors in Xcode

The new behavior is enabled by default for the standard Gradle tasks for Xcode integration, like `embedAndSignAppleFrameworkForXcode` that can connect the iOS framework from your multiplatform project to the iOS application in Xcode. It can also be enabled (or disabled) with the `kotlin.native.useXcodeMessageStyle` Gradle property.

Kotlin/JavaScript

Kotlin 1.8.20 changes the ways TypeScript definitions can be generated. It also includes a change designed to improve your debugging experience:

- Removal of Dukat integration from the Gradle plugin
- Kotlin variable and function names in source maps
- Opt in for generation of TypeScript definition files

Removal of Dukat integration from Gradle plugin

In Kotlin 1.8.20, we've removed our Experimental Dukat integration from the Kotlin/JavaScript Gradle plugin. The Dukat integration supported the automatic conversion of TypeScript declaration files (`.d.ts`) into Kotlin external declarations.

You can still convert TypeScript declaration files (`.d.ts`) into Kotlin external declarations by using our Dukat tool instead.



The Dukat tool is Experimental. It may be dropped or changed at any time.

Kotlin variable and function names in source maps

To help with debugging, we've introduced the ability to add the names that you declared in Kotlin code for variables and functions into your source maps. Prior to 1.8.20, these weren't available in source maps, so in the debugger, you always saw the variable and function names of the generated JavaScript.

You can configure what is added by using `sourceMapNamesPolicy` in your Gradle file `build.gradle.kts` , or the `-source-map-names-policy` compiler option. The table below lists the possible settings:

Setting	Description	Example output
`simple-names`	Variable names and simple function names are added. (Default)	`main`
`fully-qualified-names`	Variable names and fully qualified function names are added.	`com.example.kjs.playground.main`
`no`	No variable or function names are added.	N/A

See below for an example configuration in a `build.gradle.kts` file:

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.Kotlin2JsCompile>().configureEach {

compilercompileOptions.sourceMapNamesPolicy.set(org.jetbrains.kotlin.gradle.dsl.JsSourceMapNamesPolicy.S
// or SOURCE_MAP_NAMES_POLICY_NO, or SOURCE_MAP_NAMES_POLICY_SIMPLE_NAMES
}
```

Debugging tools like those provided in Chromium-based browsers can pick up the original Kotlin names from your source map to improve the readability of your stack trace. Happy debugging!

The addition of variable and function names in source maps is Experimental. It may be dropped or changed at any time.

Opt in for generation of TypeScript definition files

Previously, if you had a project that produced executable files (`binaries.executable()`), the Kotlin/JS IR compiler collected any top-level declarations marked with `@JsExport` and automatically generated TypeScript definitions in a `.d.ts` file.

As this isn't useful for every project, we've changed the behavior in Kotlin 1.8.20. If you want to generate TypeScript definitions, you have to explicitly configure this in your Gradle build file. Add `generateTypeScriptDefinitions()` to your `build.gradle.kts.file` in the `js` section. For example:

```
kotlin {
    js {
        binaries.executable()
        browser {
        }
        generateTypeScriptDefinitions()
    }
}
```

The generation of TypeScript definitions (`d.ts`) is Experimental. It may be dropped or changed at any time.

Gradle

Kotlin 1.8.20 is fully compatible with Gradle 6.8 through 7.6 except for some special cases in the Multiplatform plugin. You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

This version brings the following changes:

- New alignment of Gradle plugins' versions
- New JVM incremental compilation by default in Gradle
- Precise backup of compilation tasks' outputs
- Lazy Kotlin/JVM task creation for all Gradle versions
- Non-default location of compile tasks' destinationDirectory
- Ability to opt-out from reporting compiler arguments to an HTTP statistics service

New Gradle plugins versions alignment

Gradle provides a way to ensure dependencies that must work together are always aligned in their versions. Kotlin 1.8.20 adopted this approach, too. It works by default so that you don't need to change or update your configuration to enable it. In addition, you no longer need to resort to this workaround for resolving Kotlin Gradle plugins' transitive dependencies.

We would appreciate your feedback on this feature in YouTrack.

New JVM incremental compilation by default in Gradle

The new approach to incremental compilation, which has been available since Kotlin 1.7.0, now works by default. You no longer need to specify `kotlin.incremental.useClasspathSnapshot=true` in your `gradle.properties` to enable it.

We would appreciate your feedback on this. You can file an issue in YouTrack.

Precise backup of compilation tasks' outputs

> Precise backup of compilation tasks' outputs is Experimental. To use it, add `kotlin.compiler.preciseCompilationResultsBackup=true` to `gradle.properties`. We would appreciate your feedback on it in YouTrack.

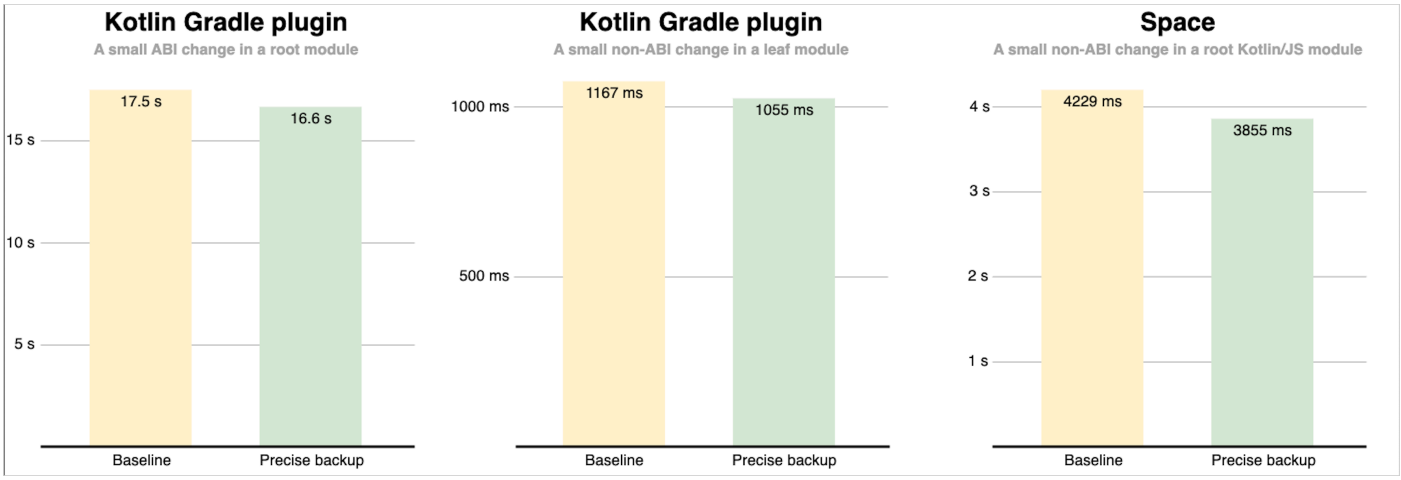
Starting with Kotlin 1.8.20, you can enable precise backup, whereby only those classes that Kotlin recompiles in the incremental compilation will be backed up. Both full and precise backups help to run builds incrementally again after compilation errors. Precise backup also saves build time compared to full backup. Full backup may take **noticeable** build time in large projects or if many tasks are making backups, especially if a project is located on a slow HDD.

This optimization is Experimental. You can enable it by adding the `kotlin.compiler.preciseCompilationResultsBackup` Gradle property to the `gradle.properties` file:

```
kotlin.compiler.preciseCompilationResultsBackup=true
```

Example of precise backup usage in JetBrains

In the following charts, you can see examples of using precise backup compared to full backup:



Comparison of full and precise backups

The first and second charts show how precise backup in the Kotlin project affects building the Kotlin Gradle plugin:

1. After making a small ABI change – adding a new public method – to a module that lots of modules depend on.
2. After making a small non-ABI change – adding a private function – to a module that no other modules depend on.

The third chart shows how precise backup in the Space project affects building a web frontend after a small non-ABI change – adding a private function – to a Kotlin/JS module that lots of modules depend on.

These measurements were performed on a computer with the Apple M1 Max CPU; different computers will yield slightly different results. The factors affecting performance include but are not limited to:

- How warm the Kotlin daemon and the Gradle daemon are.
- How fast or slow the disk is.
- The CPU model and how busy it is.
- Which modules are affected by the changes and how big these modules are.
- Whether the changes are ABI or non-ABI.

Evaluating optimizations with build reports

To estimate the impact of the optimization on your computer for your project and your scenarios, you can use Kotlin build reports. Enable reports in the text file format by adding the following property to your `gradle.properties` file:

```
kotlin.build.report.output=file
```

Here is an example of a relevant part of the report before enabling precise backup:

```
Task ':kotlin-gradle-plugin:compileCommonKotlin' finished in 0.59 s
<...>
Time metrics:
 Total Gradle task time: 0.59 s
 Task action before worker execution: 0.24 s
  Backup output: 0.22 s // Pay attention to this number
<...>
```

And here is an example of a relevant part of the report after enabling precise backup:

```
Task ':kotlin-gradle-plugin:compileCommonKotlin' finished in 0.46 s
<...>
Time metrics:
 Total Gradle task time: 0.46 s
 Task action before worker execution: 0.07 s
  Backup output: 0.05 s // The time has reduced
 Run compilation in Gradle worker: 0.32 s
  Clear jar cache: 0.00 s
  Precise backup output: 0.00 s // Related to precise backup
  Cleaning up the backup stash: 0.00 s // Related to precise backup
<...>
```

Lazy Kotlin/JVM tasks creation for all Gradle versions

For projects with the `org.jetbrains.kotlin.gradle.jvm` plugin on Gradle 7.3+, the Kotlin Gradle plugin no longer creates and configures the task `compileKotlin` eagerly. On lower Gradle versions, it simply registers all the tasks and doesn't configure them on a dry run. The same behavior is now in place when using Gradle 7.3+.

Non-default location of compile tasks' destinationDirectory

Update your build script with some additional code if you do one of the following:

- Override the Kotlin/JVM `KotlinJvmCompile` / `KotlinCompile` task's `destinationDirectory` location.
- Use a deprecated Kotlin/JS/Non-IR variant and override the `Kotlin2JsCompile` task's `destinationDirectory`.

You need to explicitly add `sourceSets.main.kotlin.classesDirectories` to `sourceSets.main.outputs` in your JAR file:

```
tasks.jar(type: Jar) {
    from sourceSets.main.outputs
    from sourceSets.main.kotlin.classesDirectories
}
```

Ability to opt-out from reporting compiler arguments to an HTTP statistics service

You can now control whether the Kotlin Gradle plugin should include compiler arguments in HTTP build reports. Sometimes, you might not need the plugin to report these arguments. If a project contains many modules, its compiler arguments in the report can be very heavy and not that helpful. There is now a way to disable it and thus save memory. In your `gradle.properties` or `local.properties`, use the `kotlin.build.report.include_compiler_arguments=(true|false)` property.

We would appreciate your feedback on this feature on YouTrack.

Standard library

Kotlin 1.8.20 adds a variety of new features, including some that are particularly useful for Kotlin/Native development:

- Support for the AutoCloseable interface
- Support for Base64 encoding and decoding
- Support for @Volatile in Kotlin/Native
- Bug fix for stack overflow when using regex in Kotlin/Native

Support for the AutoCloseable interface

> The new `AutoCloseable` interface is Experimental, and to use it you need to opt in with `@OptIn(ExperimentalStdlibApi::class)` or the compiler argument `-opt-in=kotlin.ExperimentalStdlibApi`.

The `AutoCloseable` interface has been added to the common standard library so that you can use one common interface for all libraries to close resources. In Kotlin/JVM, the `AutoCloseable` interface is an alias for `java.lang.AutoClosable`.

In addition, the extension function `use()` is now included, which executes a given block function on the selected resource and then closes it down correctly, whether an exception is thrown or not.

There is no public class in the common standard library that implements the `AutoCloseable` interface. In the example below, we define the `XMLWriter` interface and assume that there is a resource that implements it. For example, this resource could be a class that opens a file, writes XML content, and then closes it.

```
interface XMLWriter : AutoCloseable {
    fun document(encoding: String, version: String, content: XMLWriter.() -> Unit)
    fun element(name: String, content: XMLWriter.() -> Unit)
    fun attribute(name: String, value: String)
    fun text(value: String)
}

fun writeBooksTo(writer: XMLWriter) {
    writer.use { xml ->
        xml.document(encoding = "UTF-8", version = "1.0") {
            element("bookstore") {
                element("book") {
                    attribute("category", "fiction")
                    element("title") { text("Harry Potter and the Prisoner of Azkaban") }
                    element("author") { text("J. K. Rowling") }
                    element("year") { text("1999") }
                    element("price") { text("29.99") }
                }
                element("book") {
                    attribute("category", "programming")
                    element("title") { text("Kotlin in Action") }
                    element("author") { text("Dmitry Jemerov") }
                    element("author") { text("Svetlana Isakova") }
                    element("year") { text("2017") }
                    element("price") { text("25.19") }
                }
            }
        }
    }
}
```

Support for Base64 encoding

The new encoding and decoding functionality is Experimental, and to use it, you need to opt in with `@OptIn(ExperimentalEncodingApi::class)` or the compiler argument `-opt-in=kotlin.io.encoding.ExperimentalEncodingApi`.

We've added support for Base64 encoding and decoding. We provide 3 class instances, each using different encoding schemes and displaying different behaviors. Use the `Base64.Default` instance for the standard Base64 encoding scheme.

Use the `Base64.UrlSafe` instance for the "URL and Filename safe" encoding scheme.

Use the `Base64.Mime` instance for the MIME encoding scheme. When you use the `Base64.Mime` instance, all encoding functions insert a line separator every 76 characters. In the case of decoding, any illegal characters are skipped and don't throw an exception.

The `Base64.Default` instance is the companion object of the `Base64` class. As a result, you can call its functions via `Base64.encode()` and `Base64.decode()` instead of `Base64.Default.encode()` and `Base64.Default.decode()`.

```
val foBytes = "fo".map { it.code.toByte() }.toByteArray()
Base64.Default.encode(foBytes) // "Zm8="
// Alternatively:
// Base64.encode(foBytes)

val foobarBytes = "foobar".map { it.code.toByte() }.toByteArray()
Base64.UrlSafe.encode(foobarBytes) // "Zm9vYmFy"

Base64.Default.decode("Zm8=") // foBytes
// Alternatively:
// Base64.decode("Zm8=")

Base64.UrlSafe.decode("Zm9vYmFy") // foobarBytes
```

You can use additional functions to encode or decode bytes into an existing buffer, as well as to append the encoding result to a provided `Appendable` type object.

In Kotlin/JVM, we've also added the extension functions `encodingWith()` and `decodingWith()` to enable you to perform Base64 encoding and decoding with input and output streams.

Support for @Volatile in Kotlin/Native

`@Volatile` in Kotlin/Native is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

If you annotate a `var` property with `@Volatile`, then the backing field is marked so that any reads or writes to this field are atomic, and writes are always made visible to other threads.

Prior to 1.8.20, the `kotlin.jvm.Volatile` annotation was available in the common standard library. However, this annotation is only effective in the JVM. If you use it in Kotlin/Native, it is ignored, which can lead to errors.

In 1.8.20, we've introduced a common annotation, `kotlin.concurrent.Volatile`, that you can use in both the JVM and Kotlin/Native.

How to enable

To try this feature out, opt in with `@OptIn(ExperimentalStdlibApi)` and enable the `-language-version 1.9` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)` file:

Kotlin

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
    }
```

Bug fix for stack overflow when using regex in Kotlin/Native

In previous versions of Kotlin, a crash could occur if your regex input contained a large number of characters, even when the regex pattern was very simple. In 1.8.20, this issue has been resolved. For more information, see KT-46211.

Serialization updates

Kotlin 1.8.20 comes with Alpha support for the Kotlin K2 compiler and prohibits serializer customization via companion object.

Prototype serialization compiler plugin for Kotlin K2 compiler

Support for the serialization compiler plugin for K2 is in Alpha. To use it, enable the Kotlin K2 compiler.

Starting with 1.8.20, the serialization compiler plugin works with the Kotlin K2 compiler. Give it a try and share your feedback with us!

Prohibit implicit serializer customization via companion object

Currently, it is possible to declare a class as serializable with the `@Serializable` annotation and, at the same time, declare a custom serializer with the `@Serializer` annotation on its companion object.

For example:

```
import kotlinx.serialization.*

@Serializable
class Foo(val a: Int) {
    @Serializer(Foo::class)
    companion object {
        // Custom implementation of KSerializer<Foo>
    }
}
```

In this case, it's not clear from the `@Serializable` annotation which serializer is used. In actual fact, class `Foo` has a custom serializer.

To prevent this kind of confusion, in Kotlin 1.8.20 we've introduced a compiler warning for when this scenario is detected. The warning includes a possible migration path to resolve this issue.

If you use such constructs in your code, we recommend updating them to the below:

```
import kotlinx.serialization.*

@Serializable(Foo.Companion::class)
class Foo(val a: Int) {
    // Doesn't matter if you use @Serializer(Foo::class) or not
    companion object: KSerializer<Foo> {
        // Custom implementation of KSerializer<Foo>
    }
}
```

With this approach, it is clear that the `Foo` class uses the custom serializer declared in the companion object. For more information, see our YouTrack ticket.

> In Kotlin 2.0, we plan to promote the compile warning to a compiler error. We recommend that you migrate your code if you see this warning.

Documentation updates

The Kotlin documentation has received some notable changes:

- Get started with Spring Boot and Kotlin – create a simple application with a database and learn more about the features of Spring Boot and Kotlin.
- Scope functions – learn how to simplify your code with useful scope functions from the standard library.
- CocoaPods integration – set up an environment to work with CocoaPods.

Install Kotlin 1.8.20

Check the IDE version

IntelliJ IDEA 2022.2 and 2022.3 automatically suggest updating the Kotlin plugin to version 1.8.20. IntelliJ IDEA 2023.1 has the built-in Kotlin plugin 1.8.20.

Android Studio Flamingo (222) and Giraffe (223) will support Kotlin 1.8.20 in the next releases.

The new command-line compiler is available for download on the GitHub release page.

Configure Gradle settings

To download Kotlin artifacts and dependencies properly, update your `settings.gradle(.kts)` file to use the Maven Central repository:

```
pluginManagement {
    repositories {
        mavenCentral()
        gradlePluginPortal()
    }
}
```

If the repository is not specified, Gradle uses the sunset JCenter repository that could lead to issues with Kotlin artifacts.

What's new in Kotlin 1.8.0

Released: 28 December 2022

The Kotlin 1.8.0 release is out and here are some of its biggest highlights:

- New experimental functions for JVM: recursively copy or delete directory content
- Improved kotlin-reflect performance
- New -Xdebug compiler option for better debugging experience
- `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` merged into `kotlin-stdlib`
- Improved Objective-C/Swift interoperability
- Compatibility with Gradle 7.3

For information about the Kotlin release cycle, see Kotlin release process.

IDE support

The Kotlin plugin that supports 1.8.0 is available for:

IDE	Supported versions
IntelliJ IDEA	2021.3, 2022.1, 2022.2
Android Studio	Electric Eel (221), Flamingo (222)

You can update your projects to Kotlin 1.8.0 in IntelliJ IDEA 2022.3 without updating the IDE plugin.

To migrate existing projects to Kotlin 1.8.0 in IntelliJ IDEA 2022.3, change the Kotlin version to `1.8.0` and reimport your Gradle or Maven project.

Kotlin/JVM

Starting with version 1.8.0, the compiler can generate classes with a bytecode version corresponding to JVM 19. The new language version also includes:

- A compiler option for switching off the generation of JVM annotation targets
- A new `-Xdebug` compiler option for disabling optimizations
- The removal of the old backend
- Support for Lombok's @Builder annotation

Ability to not generate TYPE_USE and TYPE_PARAMETER annotation targets

If a Kotlin annotation has `TYPE` among its Kotlin targets, the annotation maps to `java.lang.annotation.ElementType.TYPE_USE` in its list of Java annotation targets. This is just like how the

`TYPE_PARAMETER` Kotlin target maps to the `java.lang.annotation.ElementType.TYPE_PARAMETER` Java target. This is an issue for Android clients with API levels less than 26, which don't have these targets in the API.

Starting with Kotlin 1.8.0, you can use the new compiler option `-Xno-new-java-annotation-targets` to avoid generating the `TYPE_USE` and `TYPE_PARAMETER` annotation targets.

A new compiler option for disabling optimizations

Kotlin 1.8.0 adds a new `-Xdebug` compiler option, which disables optimizations for a better debugging experience. For now, the option disables the "was optimized out" feature for coroutines. In the future, after we add more optimizations, this option will disable them, too.

The "was optimized out" feature optimizes variables when you use suspend functions. However, it is difficult to debug code with optimized variables because you don't see their values.

Never use this option in production: Disabling this feature via `-Xdebug` can cause memory leaks.

Removal of the old backend

In Kotlin 1.5.0, we announced that the IR-based backend became Stable. That meant that the old backend from Kotlin 1.4.* was deprecated. In Kotlin 1.8.0, we've removed the old backend completely. By extension, we've removed the compiler option `-Xuse-old-backend` and the Gradle `useOldBackend` option.

Support for Lombok's @Builder annotation

The community has added so many votes for the Kotlin Lombok: Support generated builders (@Builder) YouTrack issue that we just had to support the @Builder annotation.

We don't yet have plans to support the `@SuperBuilder` or `@Tolerate` annotations, but we'll reconsider if enough people vote for the @SuperBuilder and @Tolerate issues.

Learn how to configure the Lombok compiler plugin.

Kotlin/Native

Kotlin 1.8.0 includes changes to Objective-C and Swift interoperability, support for Xcode 14.1, and improvements to the CocoaPods Gradle plugin:

- Support for Xcode 14.1
- Improved Objective-C/Swift interoperability
- Dynamic frameworks by default in the CocoaPods Gradle plugin

Support for Xcode 14.1

The Kotlin/Native compiler now supports the latest stable Xcode version, 14.1. The compatibility improvements include the following changes:

- There's a new `watchosDeviceArm64` preset for the watchOS target that supports Apple watchOS on ARM64 platforms.
- The Kotlin CocoaPods Gradle plugin no longer has bitcode embedding for Apple frameworks by default.

- Platform libraries were updated to reflect the changes to Objective-C frameworks for Apple targets.

Improved Objective-C/Swift interoperability

To make Kotlin more interoperable with Objective-C and Swift, three new annotations were added:

- `@ObjCName` allows you to specify a more idiomatic name in Swift or Objective-C, instead of renaming the Kotlin declaration.

  The annotation instructs the Kotlin compiler to use a custom Objective-C and Swift name for this class, property, parameter, or function:

  ```
  @ObjCName(swiftName = "MySwiftArray")
  class MyKotlinArray {
      @ObjCName("index")
      fun indexOf(@ObjCName("of") element: String): Int = TODO()
  }

  // Usage with the ObjCName annotations
  let array = MySwiftArray()
  let index = array.index(of: "element")
  ```

- `@HiddenFromObjC` allows you to hide a Kotlin declaration from Objective-C.

  The annotation instructs the Kotlin compiler not to export a function or property to Objective-C and, consequently, Swift. This can make your Kotlin code more Objective-C/Swift-friendly.

- `@ShouldRefineInSwift` is useful for replacing a Kotlin declaration with a wrapper written in Swift.

  The annotation instructs the Kotlin compiler to mark a function or property as `swift_private` in the generated Objective-C API. Such declarations get the `__` prefix, which makes them invisible to Swift code.

  You can still use these declarations in your Swift code to create a Swift-friendly API, but they won't be suggested by Xcode's autocompletion, for example.

  For more information on refining Objective-C declarations in Swift, see the official Apple documentation.

> The new annotations require opt-in.

The Kotlin team is very grateful to Rick Clephas for implementing these annotations.

Dynamic frameworks by default in the CocoaPods Gradle plugin

Starting with Kotlin 1.8.0, Kotlin frameworks registered by the CocoaPods Gradle plugin are linked dynamically by default. The previous static implementation was inconsistent with the behavior of the Kotlin Gradle plugin.

```
kotlin {
    cocoapods {
        framework {
            baseName = "MyFramework"
            isStatic = false // Now dynamic by default
        }
    }
}
```

If you have an existing project with a static linking type and you upgrade to Kotlin 1.8.0 (or change the linking type explicitly), you may encounter an error with the project's execution. To fix it, close your Xcode project and run `pod install` in the Podfile directory.

For more information, see the CocoaPods Gradle plugin DSL reference.

Kotlin Multiplatform: A new Android source set layout

Kotlin 1.8.0 introduces a new Android source set layout that replaces the previous naming schema for directories, which is confusing in multiple ways.

Consider an example of two `androidTest` directories created in the current layout. One is for `KotlinSourceSets` and the other is for `AndroidSourceSets`:

- They have different semantics: Kotlin's `androidTest` belongs to the `unitTest` type, whereas Android's belongs to the `integrationTest` type.
- They create a confusing `SourceDirectories` layout, as `src/androidTest/kotlin` has a `UnitTest` and `src/androidTest/java` has an `InstrumentedTest`.
- Both `KotlinSourceSets` and `AndroidSourceSets` use a similar naming schema for Gradle configurations, so the resulting configurations of `androidTest` for both Kotlin's and Android's source sets are the same: `androidTestImplementation`, `androidTestApi`, `androidTestRuntimeOnly`, and `androidTestCompileOnly`.

To address these and other existing issues, we've introduced a new Android source set layout. Here are some of the key differences between the two layouts:

KotlinSourceSet naming schema

Current source set layout	New source set layout
`targetName` + `AndroidSourceSet.name`	`targetName` + `AndroidVariantType`

`{AndroidSourceSet.name}` maps to `{KotlinSourceSet.name}` as follows:

	Current source set layout	New source set layout
main	androidMain	androidMain
test	androidTest	androidUnitTest
androidTest	androidAndroidTest	androidInstrumentedTest

SourceDirectories

Current source set layout	New source set layout
The layout adds additional `/kotlin` SourceDirectories	`src/{AndroidSourceSet.name}/kotlin`, `src/{KotlinSourceSet.name}/kotlin`

`{AndroidSourceSet.name}` maps to `{SourceDirectories included}` as follows:

	Current source set layout	New source set layout
main	src/androidMain/kotlin, src/main/kotlin, src/main/java	src/androidMain/kotlin, src/main/kotlin, src/main/java
test	src/androidTest/kotlin, src/test/kotlin, src/test/java	src/androidUnitTest/kotlin, src/test/kotlin, src/test/java
androidTest	src/androidAndroidTest/kotlin, src/androidTest/java	src/androidInstrumentedTest/kotlin, src/androidTest/java, src/androidTest/kotlin

The location of the AndroidManifest.xml file

Current source set layout	New source set layout
src/{AndroidSourceSet.name}/AndroidManifest.xml	src/{KotlinSourceSet.name}/AndroidManifest.xml

`{AndroidSourceSet.name}` maps to `{AndroidManifest.xml location}` as follows:

	Current source set layout	New source set layout
main	src/main/AndroidManifest.xml	src/androidMain/AndroidManifest.xml
debug	src/debug/AndroidManifest.xml	src/androidDebug/AndroidManifest.xml

The relation between Android and common tests

The new Android source set layout changes the relation between Android-instrumented tests (renamed to `androidInstrumentedTest` in the new layout) and common tests.

Previously, there was a default `dependsOn` relation between `androidAndroidTest` and `commonTest`. In practice, it meant the following:

- The code in `commonTest` was available in `androidAndroidTest`.
- `expect` declarations in `commonTest` had to have corresponding `actual` implementations in `androidAndroidTest`.

- Tests declared in `commonTest` were also running as Android instrumented tests.

In the new Android source set layout, the `dependsOn` relation is not added by default. If you prefer the previous behavior, manually declare this relation in your `build.gradle.kts` file:

```
kotlin {
    // ...
    sourceSets {
        val commonTest by getting
        val androidInstrumentedTest by getting {
            dependsOn(commonTest)
        }
    }
}
```

Support for Android flavors

Previously, the Kotlin Gradle plugin eagerly created source sets that correspond to Android source sets with `debug` and `release` build types or custom flavors like `demo` and `full`. It made them accessible by constructions like `val androidDebug by getting { ... }`.

In the new Android source set layout, those source sets are created in the `afterEvaluate` phase. It makes such expressions invalid, leading to errors like `org.gradle.api.UnknownDomainObjectException: KotlinSourceSet with name 'androidDebug' not found`.

To work around that, use the new `invokeWhenCreated()` API in your `build.gradle.kts` file:

```
kotlin {
    // ...
    sourceSets.invokeWhenCreated("androidFreeDebug") {
        // ...
    }
}
```

Configuration and setup

The new layout will become the default in future releases. You can enable it now with the following Gradle option:

```
kotlin.mpp.androidSourceSetLayoutVersion=2
```

The new layout requires Android Gradle plugin 7.0 or later and is supported in Android Studio 2022.3 and later.

The usage of the previous Android-style directories is now discouraged. Kotlin 1.8.0 marks the start of the deprecation cycle, introducing a warning for the current layout. You can suppress the warning with the following Gradle property:

```
kotlin.mpp.androidSourceSetLayoutVersion1.nowarn=true
```

Kotlin/JS

Kotlin 1.8.0 stabilizes the JS IR compiler backend and brings new features to JavaScript-related Gradle build scripts:

- Stable JS IR compiler backend
- New settings for reporting that yarn.lock has been updated
- Add test targets for browsers via Gradle properties
- New approach to adding CSS support to your project

Stable JS IR compiler backend

Starting with this release, the Kotlin/JS intermediate representation (IR-based) compiler backend is Stable. It took a while to unify infrastructure for all three backends, but they now work with the same IR for Kotlin code.

As a consequence of the stable JS IR compiler backend, the old one is deprecated from now on.

Incremental compilation is enabled by default along with the stable JS IR compiler.

If you still use the old compiler, switch your project to the new backend.

New settings for reporting that yarn.lock has been updated

If you use the `yarn` package manager, there are three new special Gradle settings that could notify you if the `yarn.lock` file has been updated. You can use these settings when you want to be notified if `yarn.lock` has been changed silently during the CI build process.

These three new Gradle properties are:

- `YarnLockMismatchReport`, which specifies how changes to the `yarn.lock` file are reported. You can use one of the following values:
  - `FAIL` fails the corresponding Gradle task. This is the default.
  - `WARNING` writes the information about changes in the warning log.
  - `NONE` disables reporting.
- `reportNewYarnLock`, which reports about the recently created `yarn.lock` file explicitly. By default, this option is disabled: it's a common practice to generate a new `yarn.lock` file at the first start. You can use this option to ensure that the file has been committed to your repository.
- `yarnLockAutoReplace`, which replaces `yarn.lock` automatically every time the Gradle task is run.

To use these options, update your build script file `build.gradle.kts` as follows:

```
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnLockMismatchReport
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension

rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin::class.java) {
    rootProject.the<YarnRootExtension>().yarnLockMismatchReport =
        YarnLockMismatchReport.WARNING // NONE | FAIL
    rootProject.the<YarnRootExtension>().reportNewYarnLock = false // true
    rootProject.the<YarnRootExtension>().yarnLockAutoReplace = false // true
}
```

Add test targets for browsers via Gradle properties

Starting with Kotlin 1.8.0, you can set test targets for different browsers right in the Gradle properties file. Doing so shrinks the size of the build script file as you no longer need to write all targets in `build.gradle.kts`.

You can use this property to define a list of browsers for all modules, and then add specific browsers in the build scripts of particular modules.

For example, the following line in your Gradle property file will run the test in Firefox and Safari for all modules:

```
kotlin.js.browser.karma.browsers=firefox,safari
```

See the full list of available values for the property on GitHub.

The Kotlin team is very grateful to Martynas Petuška for implementing this feature.

New approach to adding CSS support to your project

This release provides a new approach to adding CSS support to your project. We assume that this will affect a lot of projects, so don't forget to update your Gradle build script files as described below.

Before Kotlin 1.8.0, the `cssSupport.enabled` property was used to add CSS support:

```
browser {
    commonWebpackConfig {
        cssSupport.enabled = true
    }
}
```

Now you should use the `enabled.set()` method in the `cssSupport {}` block:

```
browser {
    commonWebpackConfig {
        cssSupport {
            enabled.set(true)
        }
    }
}
```

Gradle

Kotlin 1.8.0 **fully** supports Gradle versions 7.2 and 7.3. You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

This version brings lots of changes:

- Exposing Kotlin compiler options as Gradle lazy properties
- Bumping the minimum supported versions
- Ability to disable the Kotlin daemon fallback strategy
- Usage of the latest kotlin-stdlib version in transitive dependencies
- Obligatory check for JVM target compatibility equality of related Kotlin and Java compile tasks
- Resolution of Kotlin Gradle plugins' transitive dependencies

- Deprecations and removals

Exposing Kotlin compiler options as Gradle lazy properties

To expose available Kotlin compiler options as Gradle lazy properties and to integrate them better into the Kotlin tasks, we made lots of changes:

- Compile tasks have the new `compilerOptions` input, which is similar to the existing `kotlinOptions` but uses `Property` from the Gradle Properties API as the return type:

```
tasks.named("compileKotlin", org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompile::class.java) {
    compilerOptions {
        useK2.set(true)
    }
}
```

- The Kotlin tools tasks `KotlinJsDce` and `KotlinNativeLink` have the new `toolOptions` input, which is similar to the existing `kotlinOptions` input.
- New inputs have the `@Nested` Gradle annotation. Every property inside the inputs has a related Gradle annotation, such as `@Input` or `@Internal`.
- The Kotlin Gradle plugin API artifact has two new interfaces:
  - `org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask`, which has the `compilerOptions` input and the `compileOptions()` method. All Kotlin compilation tasks implement this interface.
  - `org.jetbrains.kotlin.gradle.tasks.KotlinToolTask`, which has the `toolOptions` input and the `toolOptions()` method. All Kotlin tool tasks – `KotlinJsDce`, `KotlinNativeLink`, and `KotlinNativeLinkArtifactTask` – implement this interface.
- Some `compilerOptions` use the new types instead of the `String` type:
  - `JvmTarget`
  - `KotlinVersion` (for the `apiVersion` and the `languageVersion` inputs)
  - `JsMainFunctionExecutionMode`
  - `JsModuleKind`
  - `JsSourceMapEmbedMode`

  For example, you can use `compilerOptions.jvmTarget.set(JvmTarget.JVM_11)` instead of `kotlinOptions.jvmTarget = "11"`.

  The `kotlinOptions` types didn't change, and they are internally converted to `compilerOptions` types.
- The Kotlin Gradle plugin API is binary-compatible with previous releases. There are, however, some source and ABI-breaking changes in the `kotlin-gradle-plugin` artifact. Most of these changes involve additional generic parameters to some internal types. One important change is that the `KotlinNativeLink` task no longer inherits the `AbstractKotlinNativeCompile` task.
- `KotlinJsCompilerOptions.outputFile` and the related `KotlinJsOptions.outputFile` options are deprecated. Use the `Kotlin2JsCompile.outputFileProperty` task input instead.

The Kotlin Gradle plugin still adds the `KotlinJvmOptions` DSL to the Android extension:

```
android {
    kotlinOptions {
        jvmTarget = "11"
    }
}
```

This will be changed in the scope of this issue, when the `compilerOptions` DSL will be added to a module level.

Limitations

The `kotlinOptions` task input and the `kotlinOptions{...}` task DSL are in support mode and will be deprecated in upcoming releases. Improvements will be made only to `compilerOptions` and `toolOptions`.

Calling any setter or getter on `kotlinOptions` delegates to the related property in the `compilerOptions`. This introduces the following limitations:

- `compilerOptions` and `kotlinOptions` cannot be changed in the task execution phase (see one exception in the paragraph below).
- `freeCompilerArgs` returns an immutable `List<String>`, which means that, for example, `kotlinOptions.freeCompilerArgs.remove("something")` will fail.

Several plugins, including `kotlin-dsl` and the Android Gradle plugin (AGP) with Jetpack Compose enabled, try to modify the `freeCompilerArgs` attribute in the task execution phase. We've added a workaround for them in Kotlin 1.8.0. This workaround allows any build script or plugin to modify `kotlinOptions.freeCompilerArgs` in the execution phase but produces a warning in the build log. To disable this warning, use the new Gradle property `kotlin.options.suppressFreeCompilerArgsModificationWarning=true`. Gradle is going to add fixes for the `kotlin-dsl` plugin and AGP with Jetpack Compose enabled.

Bumping the minimum supported versions

Starting with Kotlin 1.8.0, the minimum supported Gradle version is 6.8.3 and the minimum supported Android Gradle plugin version is 4.1.3.

See the Kotlin Gradle plugin compatibility with available Gradle versions in our documentation

Ability to disable the Kotlin daemon fallback strategy

There is a new Gradle property `kotlin.daemon.useFallbackStrategy`, whose default value is `true`. When the value is `false`, builds fail on problems with the daemon's startup or communication. There is also a new `useDaemonFallbackStrategy` property in Kotlin compile tasks, which takes priority over the Gradle property if you use both. If there is insufficient memory to run the compilation, you can see a message about it in the logs.

The Kotlin compiler's fallback strategy is to run a compilation outside the Kotlin daemon if the daemon somehow fails. If the Gradle daemon is on, the compiler uses the "In process" strategy. If the Gradle daemon is off, the compiler uses the "Out of process" strategy. For more information, see Compiler execution strategy. Note that silent fallback to another strategy can consume a lot of system resources or lead to non-deterministic builds; see this YouTrack issue for more details.

Usage of the latest kotlin-stdlib version in transitive dependencies

If you explicitly write Kotlin version 1.8.0 or higher in your dependencies, for example: `implementation("org.jetbrains.kotlin:kotlin-stdlib:1.8.0")`, then the Kotlin Gradle Plugin will use that Kotlin version for transitive `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` dependencies. This is done to avoid class duplication from different stdlib versions (learn more about merging `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` into `kotlin-stdlib`). You can disable this behavior with the `kotlin.stdlib.jdk.variants.version.alignment` Gradle property:

```
kotlin.stdlib.jdk.variants.version.alignment=false
```

If you run into issues with version alignment, align all versions via the Kotlin BOM by declaring a platform dependency on `kotlin-bom` in your build script:

```
implementation(platform("org.jetbrains.kotlin:kotlin-bom:1.8.0"))
```

Learn about other cases and our suggested solutions in the documentation.

Obligatory check for JVM targets of related Kotlin and Java compile tasks

This section applies to your JVM project even if your source files are only in Kotlin and you don't use Java.

Starting from this release, the default value for the `kotlin.jvm.target.validation.mode` property is `error` for projects on Gradle 8.0+ (this version of Gradle has not been released yet), and the plugin will fail the build in the event of JVM target incompatibility.

The shift of the default value from `warning` to `error` is a preparation step for a smooth migration to Gradle 8.0. **We encourage you to set this property to `error`** and configure a toolchain or align JVM versions manually.

Learn more about what can go wrong if you don't check the targets' compatibility.

Resolution of Kotlin Gradle plugins' transitive dependencies

In Kotlin 1.7.0, we introduced support for Gradle plugin variants. Because of these plugin variants, a build classpath can have different versions of the Kotlin Gradle plugins that depend on different versions of some dependency, usually `kotlin-gradle-plugin-api`. This can lead to a resolution problem, and we would like to propose the following workaround, using the `kotlin-dsl` plugin as an example.

The `kotlin-dsl` plugin in Gradle 7.6 depends on the `org.jetbrains.kotlin.plugin.sam.with.receiver:1.7.10` plugin, which depends on `kotlin-gradle-plugin-api:1.7.10`. If you add the `org.jetbrains.kotlin.gradle.jvm:1.8.0` plugin, this `kotlin-gradle-plugin-api:1.7.10` transitive dependency may lead to a dependency resolution error because of a mismatch between the versions (`1.8.0` and `1.7.10`) and the variant attributes' `org.gradle.plugin.api-version` values. As a workaround, add this constraint to align the versions. This workaround may be needed until we implement the Kotlin Gradle Plugin libraries alignment platform, which is in the plans:

```
dependencies {
    constraints {
        implementation("org.jetbrains.kotlin:kotlin-sam-with-receiver:1.8.0")
    }
}
```

This constraint forces the `org.jetbrains.kotlin:kotlin-sam-with-receiver:1.8.0` version to be used in the build classpath for transitive dependencies. Learn more about one similar case in the Gradle issue tracker.

Deprecations and removals

In Kotlin 1.8.0, the deprecation cycle continues for the following properties and methods:

- In the notes for Kotlin 1.7.0 that the `KotlinCompile` task still had the deprecated Kotlin property `classpath`, which would be removed in future releases. Now, we've changed the deprecation level to `error` for the `KotlinCompile` task's `classpath` property. All compile tasks use the `libraries` input for a list of libraries required for compilation.
- We removed the `kapt.use.worker.api` property that allowed running kapt via the Gradle Workers API. By default, kapt has been using Gradle workers since Kotlin 1.3.70, and we recommend sticking to this method.
- In Kotlin 1.7.0, we announced the start of a deprecation cycle for the `kotlin.compiler.execution.strategy` property. In this release, we removed this property. Learn how to define a Kotlin compiler execution strategy in other ways.

Standard library

Kotlin 1.8.0:

- Updates JVM compilation target.
- Stabilizes a number of functions – TimeUnit conversion between Java and Kotlin, `cbrt()`, Java `Optionals` extension functions.
- Provides a preview for comparable and subtractable `TimeMarks`.
- Includes experimental extension functions for `java.nio.file.path`.
- Presents improved kotlin-reflect performance.

Updated JVM compilation target

In Kotlin 1.8.0, the standard libraries (`kotlin-stdlib`, `kotlin-reflect`, and `kotlin-script-*`) are compiled with JVM target 1.8. Previously, the standard libraries were compiled with JVM target 1.6.

Kotlin 1.8.0 no longer supports JVM targets 1.6 and 1.7. As a result, you no longer need to declare `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` separately in build scripts because the contents of these artifacts have been merged into `kotlin-stdlib`.

> If you have explicitly declared `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` as dependencies in your build scripts, then you should replace them with `kotlin-stdlib`.

Note that mixing different versions of stdlib artifacts could lead to class duplication or to missing classes. To avoid that, the Kotlin Gradle plugin can help you align stdlib versions.

cbrt()

The `cbrt()` function, which allows you to compute the real cube root of a `double` or `float`, is now Stable.

```
import kotlin.math.*

fun main() {
    val num = 27
    val negNum = -num

    println("The cube root of ${num.toDouble()} is: " +
            cbrt(num.toDouble()))
    println("The cube root of ${negNum.toDouble()} is: " +
            cbrt(negNum.toDouble()))
}
```

TimeUnit conversion between Java and Kotlin

The `toTimeUnit()` and `toDurationUnit()` functions in `kotlin.time` are now Stable. Introduced as Experimental in Kotlin 1.6.0, these functions improve interoperability between Kotlin and Java. You can now easily convert between Java `java.util.concurrent.TimeUnit` and Kotlin `kotlin.time.DurationUnit`. These functions are supported on the JVM only.

```
import kotlin.time.*

// For use from Java
fun wait(timeout: Long, unit: TimeUnit) {
    val duration: Duration = timeout.toDuration(unit.toDurationUnit())
    ...
}
```

Comparable and subtractable TimeMarks

> The new functionality of `TimeMarks` is Experimental, and to use it you need to opt in by using `@OptIn(ExperimentalTime::class)` or `@ExperimentalTime`.

Before Kotlin 1.8.0, if you wanted to calculate the time difference between multiple `TimeMarks` and **now**, you could only call `elapsedNow()` on one `TimeMark` at a time. This made it difficult to compare the results because the two `elapsedNow()` function calls couldn't be executed at exactly the same time.

To solve this, in Kotlin 1.8.0 you can subtract and compare `TimeMarks` from the same time source. Now you can create a new `TimeMark` instance to represent **now** and subtract other `TimeMarks` from it. This way, the results that you collect from these calculations are guaranteed to be relative to each other.

```
import kotlin.time.*
fun main() {
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    Thread.sleep(500) // Sleep 0.5 seconds
    val mark2 = timeSource.markNow()

    // Before 1.8.0
    repeat(4) { n ->
        val elapsed1 = mark1.elapsedNow()
        val elapsed2 = mark2.elapsedNow()

        // Difference between elapsed1 and elapsed2 can vary depending
        // on how much time passes between the two elapsedNow() calls
        println("Measurement 1.${n + 1}: elapsed1=$elapsed1, " +
                "elapsed2=$elapsed2, diff=${elapsed1 - elapsed2}")
    }
    println()

    // Since 1.8.0
    repeat(4) { n ->
        val mark3 = timeSource.markNow()
        val elapsed1 = mark3 - mark1
        val elapsed2 = mark3 - mark2

        // Now the elapsed times are calculated relative to mark3,
        // which is a fixed value
        println("Measurement 2.${n + 1}: elapsed1=$elapsed1, " +
                "elapsed2=$elapsed2, diff=${elapsed1 - elapsed2}")
    }
    // It's also possible to compare time marks with each other
    // This is true, as mark2 was captured later than mark1
    println(mark2 > mark1)
}
```

This new functionality is particularly useful in animation calculations where you want to calculate the difference between, or compare, multiple `TimeMarks` representing different frames.

Recursive copying or deletion of directories

> These new functions for `java.nio.file.path` are Experimental. To use them, you need to opt in with `@OptIn(kotlin.io.path.ExperimentalPathApi::class)` or `@kotlin.io.path.ExperimentalPathApi`. Alternatively, you can use the compiler option `-opt-in=kotlin.io.path.ExperimentalPathApi`.

We have introduced two new extension functions for `java.nio.file.Path`, `copyToRecursively()` and `deleteRecursively()`, which allow you to recursively:

- Copy a directory and its contents to another destination.
- Delete a directory and its contents.

These functions can be very useful as part of a backup process.

Error handling

Using `copyToRecursively()`, you can define what should happen if an exception occurs while copying, by overloading the `onError` lambda function:

```
sourceRoot.copyToRecursively(destinationRoot, followLinks = false,
    onError = { source, target, exception →
        logger.logError(exception, "Failed to copy $source to $target")
        OnErrorResult.TERMINATE
    })
```

When you use `deleteRecursively()`, if an exception occurs while deleting a file or folder, then the file or folder is skipped. Once the deletion has completed, `deleteRecursively()` throws an `IOException` containing all the exceptions that occurred as suppressed exceptions.

File overwrite

If `copyToRecursively()` finds that a file already exists in the destination directory, then an exception occurs. If you want to overwrite the file instead, use the overload that has `overwrite` as an argument and set it to `true`:

```
fun setUpEnvironment(projectDirectory: Path, fixtureName: String) {
    fixturesRoot.resolve(COMMON_FIXTURE_NAME)
        .copyToRecursively(projectDirectory, followLinks = false)
    fixturesRoot.resolve(fixtureName)
        .copyToRecursively(projectDirectory, followLinks = false,
            overwrite = true) // patches the common fixture
}
```

Custom copying action

To define your own custom logic for copying, use the overload that has `copyAction` as an additional argument. By using `copyAction` you can provide a lambda function, for example, with your preferred actions:

```
sourceRoot.copyToRecursively(destinationRoot, followLinks = false) { source, target →
    if (source.name.startsWith(".")) {
        CopyActionResult.SKIP_SUBTREE
    } else {
        source.copyToIgnoringExistingDirectory(target, followLinks = false)
        CopyActionResult.CONTINUE
    }
}
```

For more information on these extension functions, see our API reference.

Java Optionals extension functions

The extension functions that were introduced in Kotlin 1.7.0 are now Stable. These functions simplify working with Optional classes in Java. They can be used to unwrap and convert `Optional` objects on the JVM, and to make working with Java APIs more concise. For more information, see What's new in Kotlin 1.7.0.

Improved kotlin-reflect performance

Taking advantage of the fact that `kotlin-reflect` is now compiled with JVM target 1.8, we migrated our internal cache mechanism to Java's `ClassValue`. Previously we only cached `KClass`, but we now also cache `KType` and `KDeclarationContainer`. These changes have led to significant performance improvements when invoking `typeOf()`.

Documentation updates

The Kotlin documentation has received some notable changes:

Revamped and new pages

- Gradle overview – learn how to configure and build a Kotlin project with the Gradle build system, available compiler options, compilation, and caches in the Kotlin Gradle plugin.
- Nullability in Java and Kotlin – see the differences between Java's and Kotlin's approaches to handling possibly nullable variables.
- Lincheck guide – learn how to set up and use the Lincheck framework for testing concurrent algorithms on the JVM.

New and updated tutorials

- Get started with Gradle and Kotlin/JVM – create a console application using IntelliJ IDEA and Gradle.
- Create a multiplatform app using Ktor and SQLDelight – create a mobile application for iOS and Android using Kotlin Multiplatform Mobile.
- Get started with Kotlin Multiplatform – learn about cross-platform mobile development with Kotlin and create an app that works on both Android and iOS.

Install Kotlin 1.8.0

IntelliJ IDEA 2021.3, 2022.1, and 2022.2 automatically suggest updating the Kotlin plugin to version 1.8.0. IntelliJ IDEA 2022.3 will have the 1.8.0 version of the Kotlin plugin bundled in an upcoming minor update.

> To migrate existing projects to Kotlin 1.8.0 in IntelliJ IDEA 2022.3, change the Kotlin version to `1.8.0` and reimport your Gradle or Maven project.

For Android Studio Electric Eel (221) and Flamingo (222), version 1.8.0 of the Kotlin plugin will be delivered with the upcoming Android Studios updates. The new command-line compiler is available for download on the GitHub release page.

Compatibility guide for Kotlin 1.8.0

Kotlin 1.8.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of these changes in the Compatibility guide for Kotlin 1.8.0.

Compatibility guide for Kotlin 1.8.x

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.7 to Kotlin 1.8.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Prohibit the delegation of super calls to an abstract superclass member

Issues: KT-45508, KT-49017, KT-38078

Component: Core language

Incompatible change type: source

Short summary: Kotlin will report a compile error when an explicit or implicit super call is delegated to an *abstract* member of the superclass, even if there's a default implementation in a super interface

Deprecation cycle:

- 1.5.20: introduce a warning when non-abstract classes that do not override all abstract members are used
- 1.7.0: report a warning if a super call, in fact, accesses an abstract member from a superclass
- 1.7.0: report an error in all affected cases if the `-Xjvm-default=all` or `-Xjvm-default=all-compatibility` compatibility modes are enabled; report an error in the progressive mode
- 1.8.0: report an error in cases of declaring a concrete class with a non-overridden abstract method from the superclass, and super calls of `Any` methods are overridden as abstract in the superclass
- 1.9.0: report an error in all affected cases, including explicit super calls to an abstract method from the super class

Deprecate confusing grammar in when-with-subject

Issue: KT-48385

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 deprecated several confusing grammar constructs in `when` condition expressions

Deprecation cycle:

- 1.6.20: introduce a deprecation warning on the affected expressions
- 1.8.0: raise this warning to an error, `-XXLanguage:-ProhibitConfusingSyntaxInWhenBranches` can be used to temporarily revert to the pre-1.8 behavior
- >= 1.9: repurpose some deprecated constructs for new language features

Prevent implicit coercions between different numeric types

Issue: KT-48645

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin will avoid converting numeric values automatically to a primitive numeric type where only a downcast to that type is needed semantically

Deprecation cycle:

- < 1.5.30: the old behavior in all affected cases
- 1.5.30: fix the downcast behavior in generated property delegate accessors, `-Xuse-old-backend` can be used to temporarily revert to the pre-1.5.30 fix behavior
- >= 1.9: fix the downcast behavior in other affected cases

Make private constructors of sealed classes really private

Issue: KT-44866

Component: Core language

Incompatible change type: source

Short summary: after relaxing restrictions on where the inheritors of sealed classes could be declared in the project structure, the default visibility of sealed class constructors became protected. However, until 1.8, Kotlin still allowed calling explicitly declared private constructors of sealed classes outside those classes' scopes

Deprecation cycle:

- 1.6.20: report a warning (or an error in the progressive mode) when a private constructor of a sealed class is called outside that class
- 1.8.0: use default visibility rules for private constructors (a call to a private constructor can be resolved only if this call is inside the corresponding class), the old behavior can be brought back temporarily by specifying the `-XXLanguage:-UseConsistentRulesForPrivateConstructorsOfSealedClasses` compiler argument

Prohibit using operator == on incompatible numeric types in builder inference context

Issue: KT-45508

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit using the operator `==` on incompatible numeric types, for example, `Int` and `Long`, in scopes of builder inference lambda functions, the same way as it currently does in other contexts

Deprecation cycle:

- 1.6.20: report a warning (or an error in the progressive mode) when the operator `==` is used on incompatible numeric types
- 1.8.0: raise the warning to an error, `-XXLanguage:-ProperEqualityChecksInBuilderInferenceCalls` can be used to temporarily revert to the pre-1.8 behavior

Prohibit if without else and non-exhaustive when in right hand side of elvis operator

Issue: KT-44705

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit using a non-exhaustive `when` or the `if` expression without an `else` branch on the right hand side of the Elvis operator (`?:`). Previously, it was allowed if the Elvis operator's result was not used as an expression

Deprecation cycle:

- 1.6.20: report a warning (or an error in the progressive mode) on such non-exhaustive if and when expressions
- 1.8.0: raise this warning to an error, `-XXLanguage:-ProhibitNonExhaustiveIfInRhsOfElvis` can be used to temporarily revert to the pre-1.8 behavior

Prohibit upper bound violation in a generic type alias usage (one type parameter used in several type arguments of the aliased type)

Issues: KT-29168

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit using a type alias with type arguments that violate the upper bound restrictions of the corresponding type parameters of the aliased type in case when one typealias type parameter is used in several type arguments of the aliased type, for example, `typealias Alias<T> = Base<T, T>`

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) on usages of a type alias with type arguments violating upper bound constraints of the corresponding type parameters of the aliased type
- 1.8.0: raise this warning to an error, `-XXLanguage:-ReportMissingUpperBoundsViolatedErrorOnAbbreviationAtSupertypes` can be used to temporarily revert to the pre-1.8 behavior

Prohibit upper bound violation in a generic type alias usage (a type parameter used in a generic type argument of a type argument of the aliased type)

Issue: KT-54066

Component: Core language

Incompatible change type: source

Short summary: Kotlin will prohibit using a type alias with type arguments that violate the upper bound restrictions of the corresponding type parameters of the aliased type in case when the typealias type parameter is used as a generic type argument of a type argument of the aliased type, for example, `typealias Alias<T> = Base<List<T>>`

Deprecation cycle:

- 1.8.0: report a warning when a generic typealias usage has type arguments violating upper bound constraints of the corresponding type parameters of the aliased type
- >=1.10: raise the warning to an error

Prohibit using a type parameter declared for an extension property inside delegate

Issue: KT-24643

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit delegating extension properties on a generic type to generic types that use the type parameter of the receiver in an unsafe way

Deprecation cycle:

- 1.6.0: report a warning (or an error in the progressive mode) when delegating an extension property to a type that uses type parameters inferred from the delegated property's type arguments in a particular way
- 1.8.0: raise the warning to an error, `-XXLanguage:-ForbidUsingExtensionPropertyTypeParameterInDelegate` can be used to temporarily revert to the pre-1.8 behavior

Forbid @Synchronized annotation on suspend functions

Issue: KT-48516

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit placing the `@Synchronized` annotation on suspend functions because a suspending call should not be allowed to happen inside a synchronized block

Deprecation cycle:

- 1.6.0: report a warning on suspend functions annotated with the `@Synchronized` annotation, the warning is reported as an error in the progressive mode
- 1.8.0: raise the warning to an error, `-XXLanguage:-SynchronizedSuspendError` can be used to temporarily revert to the pre-1.8 behavior

Prohibit using spread operator for passing arguments to non-vararg parameters

Issue: KT-48162

Component: Core language

Incompatible change type: source

Short summary: Kotlin allowed passing arrays with the spread operator (`*`) to non-vararg array parameters in certain conditions. Since Kotlin 1.8, this will be prohibited

Deprecation cycle:

- 1.6.0: report a warning (or an error in the progressive mode) on using the spread operator where a non-vararg array parameter is expected
- 1.8.0: raise the warning to an error, `-XXLanguage:-ReportNonVarargSpreadOnGenericCalls` can be used to temporarily revert to the pre-1.8 behavior

Prohibit null-safety violation in lambdas passed to functions overloaded by lambda return type

Issue: KT-49658

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit returning `null` from lambdas passed to functions overloaded by those lambdas' return types when overloads don't allow a nullable return type. Previously, it was allowed when `null` was returned from one of the branches of the `when` operator

Deprecation cycle:

- 1.6.20: report a type mismatch warning (or an error in the progressive mode)
- 1.8.0: raise the warning to an error, `-XXLanguage:-DontLoseDiagnosticsDuringOverloadResolutionByReturnType` can be used to temporarily revert to the pre-1.8 behavior

Keep nullability when approximating local types in public signatures

Issue: KT-53982

Component: Core language

Incompatible change type: source, binary

Short summary: when a local or anonymous type is returned from an expression-body function without an explicitly specified return type, Kotlin compiler infers (or approximates) the return type using the known supertype of that type. During this, the compiler can infer a non-nullable type where the null value could in fact be returned

Deprecation cycle:

- 1.8.0: approximate flexible types by flexible supertypes
- 1.8.0: report a warning when a declaration is inferred to have a non-nullable type that should be nullable, prompting users to specify the type explicitly
- 1.9.0: approximate nullable types by nullable supertypes, `-XXLanguage:-KeepNullabilityWhenApproximatingLocalType` can be used to temporarily revert to the pre-1.9 behavior

Do not propagate deprecation through overrides

Issue: KT-47902

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will no longer propagate deprecation from a deprecated member in the superclass to its overriding member in the subclass, thus providing an explicit mechanism for deprecating a member of the superclass while leaving it non-deprecated in the subclass

Deprecation cycle:

- 1.6.20: reporting a warning with the message of the future behavior change and a prompt to either suppress this warning or explicitly write a `@Deprecated` annotation on an override of a deprecated member
- 1.9.0: stop propagating deprecation status to the overridden members. This change also takes effect immediately in the progressive mode

Prohibit implicit inferring a type variable into an upper bound in the builder inference context

Issue: KT-47986

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit inferring a type variable into the corresponding type parameter's upper bound in the absence of any use-site type information in the scope of builder inference lambda functions, the same way as it does currently in other contexts

Deprecation cycle:

- 1.7.20: report a warning (or an error in the progressive mode) when a type parameter is inferred into declared upper bounds in the absence of use-site type information
- 1.9.0: raise the warning to an error, `-XXLanguage:-ForbidInferringPostponedTypeVariableIntoDeclaredUpperBound` can be used to temporarily revert to the pre-1.9 behavior

Prohibit using collection literals in annotation classes anywhere except their parameters declaration

Issue: KT-39041

Component: Core language

Incompatible change type: source

Short summary: Kotlin allows using collection literals in a restricted way - for passing arrays to parameters of annotation classes or specifying default values for these parameters. However besides that, Kotlin allowed using collections literals anywhere else inside an annotation class, for example, in its nested object. Kotlin 1.9 will prohibit using collection literals in annotation classes anywhere except their parameters' default values.

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) on array literals in nested objects in annotation classes
- 1.9.0: raise the warning to an error

Prohibit forward referencing of parameters with default values in default value expressions

Issue: KT-25694

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit forward referencing of parameters with default values in default value expressions of other parameters. This ensures that by the time the parameter is accessed in a default value expression, it would already have a value either passed to the function or initialized by its own default value expression

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) when a parameter with default value is references in default value of another parameter that comes before it
- 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitIllegalValueParameterUsageInDefaultArguments` can be used to temporarily revert to the pre-1.9 behavior

Prohibit extension calls on inline functional parameters

Issue: KT-52502

Component: Core language

Incompatible change type: source

Short summary: while Kotlin allowed passing an inline functional parameter to another inline function as a receiver, it always resulted in compiler exceptions when compiling such code. Kotlin 1.9 will prohibit this, thus reporting an error instead of crashing the compiler

Deprecation cycle:

- 1.7.20: report a warning (or an error in the progressive mode) for inline extension calls on inline functional parameters
- 1.9.0: raise the warning to an error

Prohibit calls to infix functions named suspend with an anonymous function argument

Issue: KT-49264

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will no longer allow calling infix functions named `suspend` that have a single argument of a functional type passed as an anonymous function literal

Deprecation cycle:

- 1.7.20: report a warning on suspend infix calls with an anonymous function literal
- 1.9.0: raise the warning to an error, `-XXLanguage:-ModifierNonBuiltinSuspendFunError` can be used to temporarily revert to the pre-1.9 behavior
- >=1.10: Change how the `suspend fun` token sequence is interpreted by the parser

Prohibit using captured type parameters in inner classes against their variance

Issue: KT-50947

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit using type parameters of an outer class having `in` or `out` variance in an inner class of that class in positions violating that type parameters' declared variance

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) when an outer class' type parameter usage position violates the variance rules of that parameter
- 1.9.0: raise the warning to an error, `-XXLanguage:-ReportTypeVarianceConflictOnQualifierArguments` can be used to temporarily revert to the pre-1.9 behavior

Prohibit recursive call of a function without explicit return type in compound assignment operators

Issue: KT-48546

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit calling a function without explicitly specified return type in an argument of a compound assignment operator inside that function's body, as it currently does in other expressions inside the body of that function

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) when a function without explicitly specified return type is called recursively in that function's body in a compound assignment operator argument
- 1.9.0: raise the warning to an error

Prohibit unsound calls with expected @NotNull T and given Kotlin generic parameter with nullable bound

Issue: KT-36770

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit method calls where a value of a potentially nullable generic type is passed for a `@NotNull`-annotated parameter of a Java method

Deprecation cycle:

- 1.5.20: report a warning when an unconstrained generic type parameter is passed where a non-nullable type is expected
- 1.9.0: report a type mismatch error instead of the warning above, `-XXLanguage:-ProhibitUsingNullableTypeParameterAgainstNotNullAnnotated` can be used to temporarily revert to the pre-1.8 behavior

Prohibit access to members of a companion of an enum class from entry initializers of this enum

Issue: KT-49110

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.9 will prohibit all kinds of access to the companion object of an enum from an enum entry initializer

Deprecation cycle:

- 1.6.20: report a warning (or an error in the progressive mode) on such companion member access
- 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitAccessToEnumCompanionMembersInEnumConstructorCall` can be used to temporarily revert to the pre-1.8 behavior

Deprecate and remove Enum.declaringClass synthetic property

Issue: KT-49653

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin allowed using the synthetic property `declaringClass` on `Enum` values produced from the method `getDeclaringClass()` of the underlying Java class `java.lang.Enum` even though this method is not available for Kotlin `Enum` type. Kotlin 1.9 will prohibit using this property, proposing to migrate to the extension property `declaringJavaClass` instead

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) on `declaringClass` property usages, propose the migration to `declaringJavaClass` extension
- 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitEnumDeclaringClass` can be used to temporarily revert to the pre-1.9 behavior
- >=1.10: remove `declaringClass` synthetic property

Deprecate the enable and the compatibility modes of the compiler option -Xjvm-default

Issue: KT-46329

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6.20 warns about the usage of the `enable` and `compatibility` modes of the `-Xjvm-default` compiler option

Deprecation cycle:

- 1.6.20: introduce a warning on the `enable` and `compatibility` modes of the `-Xjvm-default` compiler option
- >= 1.9: raise this warning to an error

Standard library

Warn about potential overload resolution change when Range/Progression starts implementing Collection

Issue: KT-49276

Component: Core language / kotlin-stdlib

Incompatible change type: source

Short summary: it is planned to implement the `Collection` interface in the standard progressions and concrete ranges inherited from them in Kotlin 1.9. This could make a different overload selected in the overload resolution if there are two overloads of some method, one accepting an element and another accepting a collection. Kotlin will make this situation visible by reporting a warning or an error when such overloaded method is called with a range or progression argument

Deprecation cycle:

- 1.6.20: report a warning when an overloaded method is called with the standard progression or its range inheritor as an argument if implementing the `Collection` interface by this progression/range leads to another overload being selected in this call in future
- 1.8.0: raise this warning to an error
- 1.9.0: stop reporting the error, implement `Collection` interface in progressions thus changing the overload resolution result in the affected cases

Migrate declarations from kotlin.dom and kotlin.browser packages to kotlinx.*

Issue: KT-39330

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: declarations from the `kotlin.dom` and `kotlin.browser` packages are moved to the corresponding `kotlinx.*` packages to prepare for extracting them from stdlib

Deprecation cycle:

- 1.4.0: introduce the replacement API in `kotlinx.dom` and `kotlinx.browser` packages
- 1.4.0: deprecate the API in `kotlin.dom` and `kotlin.browser` packages and propose the new API above as a replacement
- 1.6.0: raise the deprecation level to an error
- 1.8.20: remove the deprecated functions from stdlib for JS-IR target
- >= 1.9: move the API in kotlinx.* packages to a separate library

Deprecate some JS-only API

Issue: KT-48587

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: a number of JS-only functions in stdlib are deprecated for removal. They include: `String.concat(String)`, `String.match(regex: String)`, `String.matches(regex: String)`, and the `sort` functions on arrays taking a comparison function, for example, `Array<out T>.sort(comparison: (a: T, b: T) -> Int)`

Deprecation cycle:

- 1.6.0: deprecate the affected functions with a warning
- 1.9.0: raise the deprecation level to an error
- >=1.10.0: remove the deprecated functions from the public API

Tools

Raise deprecation level of classpath property of KotlinCompile task

Issue: KT-51679

Component: Gradle

Incompatible change type: source

Short summary: the `classpath` property of the `KotlinCompile` task is deprecated

Deprecation cycle:

- 1.7.0: the `classpath` property is deprecated
- 1.8.0: raise the deprecation level to an error
- >=1.9.0: remove the deprecated functions from the public API

Remove kapt.use.worker.api Gradle property

Issue: KT-48827

Component: Gradle

Incompatible change type: behavioral

Short summary: remove the `kapt.use.worker.api` property that allowed to run kapt via Gradle Workers API (default: true)

Deprecation cycle:

- 1.6.20: raise the deprecation level to a warning
- 1.8.0: remove this property

Remove kotlin.compiler.execution.strategy system property

Issue: KT-51831

Component: Gradle

Incompatible change type: behavioral

Short summary: remove the `kotlin.compiler.execution.strategy` system property used to choose a compiler execution strategy. Use the Gradle property `kotlin.compiler.execution.strategy` or the compile task property `compilerExecutionStrategy` instead

Deprecation cycle:

- 1.7.0: raise the deprecation level to a warning
- 1.8.0: remove the property

Changes in compiler options

Issues: KT-27301, KT-48532

Component: Gradle

Incompatible change type: source, binary

Short summary: this change might affect Gradle plugins authors. In `kotlin-gradle-plugin`, there are additional generic parameters to some internal types (you should add generic types or `*`). `KotlinNativeLink` task does not inherit the `AbstractKotlinNativeCompile` task anymore. `KotlinJsCompilerOptions.outputFile` and the related `KotlinJsOptions.outputFile` options are deprecated. Use the `Kotlin2JsCompile.outputFileProperty` task input instead. The `kotlinOptions` task input and the `kotlinOptions{...}` task DSL are in a support mode and will be deprecated in upcoming releases. `compilerOptions` and `kotlinOptions` can not be changed on a task execution phase (see one exception in What's new in Kotlin 1.8). `freeCompilerArgs` returns an immutable `List<String>` – `kotlinOptions.freeCompilerArgs.remove("something")` will fail. The `useOldBackend` property that allowed to use the old JVM backend is removed

Deprecation cycle:

- 1.8.0: `KotlinNativeLink` task does not inherit the `AbstractKotlinNativeCompile`. `KotlinJsCompilerOptions.outputFile` and the related `KotlinJsOptions.outputFile` options are deprecated. The `useOldBackend` property that allowed to use the old JVM backend is removed.

Deprecate kotlin.internal.single.build.metrics.file property

Issue: KT-53357

Component: Gradle

Incompatible change type: source

Short summary: deprecate the `kotlin.internal.single.build.metrics.file` property used to define a single file for build reports. Use the property `kotlin.build.report.single_file` instead with `kotlin.build.report.output=single_file`

Deprecation cycle:

- 1.8.0: raise the deprecation level to a warning >= 1.9: delete the property

What's new in Kotlin 1.7.20

Released: 29 September 2022

The Kotlin 1.7.20 release is out! Here are some highlights from this release:

- The new Kotlin K2 compiler supports `all-open`, SAM with receiver, Lombok, and other compiler plugins
- We introduced the preview of the `..<` operator for creating open-ended ranges
- The new Kotlin/Native memory manager is now enabled by default
- We introduced a new experimental feature for JVM: inline classes with a generic underlying type

You can also find a short overview of the changes in this video:

Watch video online.

For information about the Kotlin release cycle, see Kotlin release process.

Support for Kotlin K2 compiler plugins

The Kotlin team continues to stabilize the K2 compiler. K2 is still in **Alpha** (as announced in the Kotlin 1.7.0 release), but it now supports several compiler plugins. You can follow this YouTrack issue to get updates from the Kotlin team on the new compiler.

Starting with this 1.7.20 release, the Kotlin K2 compiler supports the following plugins:

- `all-open`
- `no-arg`
- SAM with receiver
- Lombok
- AtomicFU
- `jvm-abi-gen`

The Alpha version of the new K2 compiler only works with JVM projects. It doesn't support Kotlin/JS, Kotlin/Native, or other multiplatform projects.

Learn more about the new compiler and its benefits in the following videos:

- The Road to the New Kotlin Compiler
- K2 Compiler: a Top-Down View

How to enable the Kotlin K2 compiler

To enable the Kotlin K2 compiler and test it, use the following compiler option:

```
-Xuse-k2
```

You can specify it in your `build.gradle(.kts)` file:

Kotlin

```
tasks.withType<KotlinCompile> {
    kotlinOptions.useK2 = true
}
```

Groovy

```
compileKotlin {
    kotlinOptions.useK2 = true
}
```

Check out the performance boost on your JVM projects and compare it with the results of the old compiler.

Leave your feedback on the new K2 compiler

We really appreciate your feedback in any form:

- Provide your feedback directly to K2 developers in Kotlin Slack: get an invite and join the #k2-early-adopters channel.
- Report any problems you faced with the new K2 compiler to our issue tracker.
- Enable the **Send usage statistics** option to allow JetBrains collecting anonymous data about K2 usage.

Language

Kotlin 1.7.20 introduces preview versions for new language features, as well as puts restrictions on builder type inference:

- Preview of the ..< operator for creating open-ended ranges
- New data object declarations
- Builder type inference restrictions

Preview of the ..< operator for creating open-ended ranges

The new operator is Experimental, and it has limited support in the IDE.

This release introduces the new `..<` operator. Kotlin has the `..` operator to express a range of values. The new `..<` operator acts like the `until` function and helps you define the open-ended range.

Watch video online.

Our research shows that this new operator does a better job at expressing open-ended ranges and making it clear that the upper bound is not included.

Here is an example of using the `..<` operator in a `when` expression:

```
when (value) {
    in 0.0..<0.25 -> // First quarter
    in 0.25..<0.5 -> // Second quarter
    in 0.5..<0.75 -> // Third quarter
    in 0.75..1.0 ->  // Last quarter  <- Note closed range here
}
```

Standard library API changes

The following new types and operations will be introduced in the `kotlin.ranges` packages in the common Kotlin standard library:

New OpenEndRange<T> interface

The new interface to represent open-ended ranges is very similar to the existing `ClosedRange<T>` interface:

```
interface OpenEndRange<T : Comparable<T>> {
    // Lower bound
    val start: T
    // Upper bound, not included in the range
    val endExclusive: T
    operator fun contains(value: T): Boolean = value ≥ start && value < endExclusive
    fun isEmpty(): Boolean = start ≥ endExclusive
}
```

Implementing OpenEndRange in the existing iterable ranges

When developers need to get a range with an excluded upper bound, they currently use the `until` function to effectively produce a closed iterable range with the same values. To make these ranges acceptable in the new API that takes `OpenEndRange<T>`, we want to implement that interface in the existing iterable ranges: `IntRange`, `LongRange`, `CharRange`, `UIntRange`, and `ULongRange`. So they will simultaneously implement both the `ClosedRange<T>` and `OpenEndRange<T>` interfaces.

```
class IntRange : IntProgression(...), ClosedRange<Int>, OpenEndRange<Int> {
    override val start: Int
    override val endInclusive: Int
    override val endExclusive: Int
}
```

rangeUntil operators for the standard types

The `rangeUntil` operators will be provided for the same types and combinations currently defined by the `rangeTo` operator. We provide them as extension functions for prototype purposes, but for consistency, we plan to make them members later before stabilizing the open-ended ranges API.

How to enable the ..< operator

To use the `..<` operator or to implement that operator convention for your own types, enable the `-language-version 1.8` compiler option.

The new API elements introduced to support the open-ended ranges of the standard types require an opt-in, as usual for an experimental stdlib API: `@OptIn(ExperimentalStdlibApi::class)`. Alternatively, you could use the `-opt-in=kotlin.ExperimentalStdlibApi` compiler option.

Read more about the new operator in this KEEP document.

Improved string representations for singletons and sealed class hierarchies with data objects

Data objects are Experimental, and have limited support in the IDE at the moment.

This release introduces a new type of `object` declaration for you to use: `data object`. Data object behaves conceptually identical to a regular `object` declaration but comes with a clean `toString` representation out of the box.

Watch video online.

```
package org.example
object MyObject
data object MyDataObject

fun main() {
    println(MyObject) // org.example.MyObject@1f32e575
    println(MyDataObject) // MyDataObject
}
```

This makes `data object` declarations perfect for sealed class hierarchies, where you may use them alongside `data class` declarations. In this snippet, declaring `EndOfFile` as a `data object` instead of a plain `object` means that it will get a pretty `toString` without the need to override it manually, maintaining symmetry with the accompanying `data class` definitions:

```
sealed class ReadResult {
    data class Number(val value: Int) : ReadResult()
    data class Text(val value: String) : ReadResult()
    data object EndOfFile : ReadResult()
}

fun main() {
    println(ReadResult.Number(1)) // Number(value=1)
    println(ReadResult.Text("Foo")) // Text(value=Foo)
    println(ReadResult.EndOfFile) // EndOfFile
}
```

How to enable data objects

To use data object declarations in your code, enable the `-language-version 1.9` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)`:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompile>().configureEach {
    // ...
    kotlinOptions.languageVersion = "1.9"
}
```

Groovy

```
compileKotlin {
    // ...
    kotlinOptions.languageVersion = '1.9'
}
```

Read more about data objects, and share your feedback on their implementation in the respective KEEP document.

New builder type inference restrictions

Kotlin 1.7.20 places some major restrictions on the use of builder type inference that could affect your code. These restrictions apply to code containing builder lambda functions, where it's impossible to derive the parameter without analyzing the lambda itself. The parameter is used as an argument. Now, the compiler will always show an error for such code and ask you to specify the type explicitly.

This is a breaking change, but our research shows that these cases are very rare, and the restrictions shouldn't affect your code. If they do, consider the following cases:

- Builder inference with extension that hides members.

  If your code contains an extension function with the same name that will be used during the builder inference, the compiler will show you an error:

  ```
  class Data {
      fun doSmth() {} // 1
  }

  fun <T> T.doSmth() {} // 2

  fun test() {
      buildList {
          this.add(Data())
          this.get(0).doSmth() // Resolves to 2 and leads to error
      }
  }
  ```

  To fix the code, you should specify the type explicitly:

  ```
  class Data {
      fun doSmth() {} // 1
  }

  fun <T> T.doSmth() {} // 2

  fun test() {
      buildList<Data> { // Type argument!
          this.add(Data())
          this.get(0).doSmth() // Resolves to 1
      }
  }
  ```

- Builder inference with multiple lambdas and the type arguments are not specified explicitly.

  If there are two or more lambda blocks in builder inference, they affect the type. To prevent an error, the compiler requires you to specify the type:

```
fun <T: Any> buildList(
    first: MutableList<T>.() -> Unit,
    second: MutableList<T>.() -> Unit
): List<T> {
    val list = mutableListOf<T>()
    list.first()
    list.second()
    return list
}

fun main() {
    buildList(
        first = { // this: MutableList<String>
            add("")
        },
        second = { // this: MutableList<Int>
            val i: Int = get(0)
            println(i)
        }
    )
}
```

To fix the error, you should specify the type explicitly and fix the type mismatch:

```
fun main() {
    buildList<Int>(
        first = { // this: MutableList<Int>
            add(0)
        },
        second = { // this: MutableList<Int>
            val i: Int = get(0)
            println(i)
        }
    )
}
```

If you haven't found your case mentioned above, file an issue to our team.

See this YouTrack issue for more information about this builder inference update.

Kotlin/JVM

Kotlin 1.7.20 introduces generic inline classes, adds more bytecode optimizations for delegated properties, and supports IR in the kapt stub generating task, making it possible to use all the newest Kotlin features with kapt:

- Generic inline classes
- More optimized cases of delegated properties
- Support for the JVM IR backend in kapt stub generating task

Generic inline classes

Generic inline classes is an Experimental feature. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 1.7.20 allows the underlying type of JVM inline classes to be a type parameter. The compiler maps it to `Any?` or, generally, to the upper bound of the type parameter.

Watch video online.

Consider the following example:

```
@JvmInline
value class UserId<T>(val value: T)

fun compute(s: UserId<String>) {} // Compiler generates fun compute-<hashcode>(s: Any?)
```

The function accepts the inline class as a parameter. The parameter is mapped to the upper bound, not the type argument.

To enable this feature, use the `-language-version 1.8` compiler option.

We would appreciate your feedback on this feature in YouTrack.

More optimized cases of delegated properties

In Kotlin 1.6.0, we optimized the case of delegating to a property by omitting the `$delegate` field and generating immediate access to the referenced property. In 1.7.20, we've implemented this optimization for more cases. The `$delegate` field will now be omitted if a delegate is:

- A named object:

```
object NamedObject {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String = ...
}

val s: String by NamedObject
```

- A final `val` property with a backing field and a default getter in the same module:

```
val impl: ReadOnlyProperty<Any?, String> = ...

class A {
    val s: String by impl
}
```

- A constant expression, an enum entry, `this`, or `null`. Here's an example of `this`:

```
class A {
    operator fun getValue(thisRef: Any?, property: KProperty<*>) ...

    val s by this
}
```

Learn more about delegated properties.

We would appreciate your feedback on this feature in YouTrack.

Support for the JVM IR backend in kapt stub generating task

> Support for the JVM IR backend in the kapt stub generating task is an Experimental feature. It may be changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes.

Before 1.7.20, the kapt stub generating task used the old backend, and repeatable annotations didn't work with kapt. With Kotlin 1.7.20, we've added support for the JVM IR backend in the kapt stub generating task. This makes it possible to use all the newest Kotlin features with kapt, including repeatable annotations.

To use the IR backend in kapt, add the following option to your `gradle.properties` file:

```
kapt.use.jvm.ir=true
```

We would appreciate your feedback on this feature in YouTrack.

Kotlin/Native

Kotlin 1.7.20 comes with the new Kotlin/Native memory manager enabled by default and gives you the option to customize the `Info.plist` file:

- The new default memory manager
- Customizing the Info.plist file

The new Kotlin/Native memory manager enabled by default

This release brings further stability and performance improvements to the new memory manager, allowing us to promote the new memory manager to Beta.

The previous memory manager complicated writing concurrent and asynchronous code, including issues with implementing the `kotlinx.coroutines` library. This blocked the adoption of Kotlin Multiplatform Mobile because concurrency limitations created problems with sharing Kotlin code between iOS and Android platforms. The new memory manager finally paves the way to promote Kotlin Multiplatform Mobile to Beta.

The new memory manager also supports the compiler cache that makes compilation times comparable to previous releases. For more on the benefits of the new memory manager, see our original blog post for the preview version. You can find more technical details in the documentation.

Configuration and setup

Starting with Kotlin 1.7.20, the new memory manager is the default. Not much additional setup is required.

If you've already turned it on manually, you can remove the `kotlin.native.binary.memoryModel=experimental` option from your `gradle.properties` or `binaryOptions["memoryModel"] = "experimental"` from the `build.gradle(.kts)` file.

If necessary, you can switch back to the legacy memory manager with the `kotlin.native.binary.memoryModel=strict` option in your `gradle.properties`. However, compiler cache support is no longer available for the legacy memory manager, so compilation times might worsen.

Freezing

In the new memory manager, freezing is deprecated. Don't use it unless you need your code to work with the legacy manager (where freezing is still required). This may be helpful for library authors that need to maintain support for the legacy memory manager or developers who want to have a fallback if they encounter issues with the new memory manager.

In such cases, you can temporarily support code for both new and legacy memory managers. To ignore deprecation warnings, do one of the following:

- Annotate usages of the deprecated API with `@OptIn(FreezingIsDeprecated::class)`.
- Apply `languageSettings.optIn("kotlin.native.FreezingIsDeprecated")` to all the Kotlin source sets in Gradle.
- Pass the compiler flag `-opt-in=kotlin.native.FreezingIsDeprecated`.

Calling Kotlin suspending functions from Swift/Objective-C

The new memory manager still restricts calling Kotlin `suspend` functions from Swift and Objective-C from threads other than the main one, but you can lift it with a new Gradle option.

This restriction was originally introduced in the legacy memory manager due to cases where the code dispatched a continuation to be resumed on the original thread. If this thread didn't have a supported event loop, the task would never run, and the coroutine would never be resumed.

In certain cases, this restriction is no longer required, but a check of all the necessary conditions can't be easily implemented. Because of this, we decided to keep it in the new memory manager while introducing an option for you to disable it. For this, add the following option to your `gradle.properties`:

```
kotlin.native.binary.objcExportSuspendFunctionLaunchThreadRestriction=none
```

Do not add this option if you use the `native-mt` version of `kotlinx.coroutines` or other libraries that have the same "dispatch to the original thread" approach.

The Kotlin team is very grateful to Ahmed El-Helw for implementing this option.

Leave your feedback

This is a significant change to our ecosystem. We would appreciate your feedback to help make it even better.

Try the new memory manager on your projects and share feedback in our issue tracker, YouTrack.

Customizing the Info.plist file

When producing a framework, the Kotlin/Native compiler generates the information property list file, `Info.plist`. Previously, it was cumbersome to customize its contents. With Kotlin 1.7.20, you can directly set the following properties:

Property	Binary option
`CFBundleIdentifier`	`bundleId`

Property	Binary option
`CFBundleShortVersionString`	`bundleShortVersionString`
`CFBundleVersion`	`bundleVersion`

To do that, use the corresponding binary option. Pass the `-Xbinary=$option=$value` compiler flag or set the `binaryOption(option, value)` Gradle DSL for the necessary framework.

The Kotlin team is very grateful to Mads Ager for implementing this feature.

Kotlin/JS

Kotlin/JS has received some enhancements that improve the developer experience and boost performance:

- Klib generation is faster in both incremental and clean builds, thanks to efficiency improvements for the loading of dependencies.
- Incremental compilation for development binaries has been reworked, resulting in major improvements in clean build scenarios, faster incremental builds, and stability fixes.
- We've improved `.d.ts` generation for nested objects, sealed classes, and parameters with default values in constructors.

Gradle

The updates for the Kotlin Gradle plugin are focused on compatibility with the new Gradle features and the latest Gradle versions.

Kotlin 1.7.20 contains changes to support Gradle 7.1. Deprecated methods and properties were removed or replaced, reducing the number of deprecation warnings produced by the Kotlin Gradle plugin and unblocking future support for Gradle 8.0.

There are, however, some potentially breaking changes that may need your attention:

Target configuration

- `org.jetbrains.kotlin.gradle.dsl.SingleTargetExtension` now has a generic parameter, `SingleTargetExtension<T : KotlinTarget>`.
- The `kotlin.targets.fromPreset()` convention has been deprecated. Instead, you can still use `kotlin.targets { fromPreset() }`, but we recommend setting up targets explicitly.
- Target accessors auto-generated by Gradle are no longer available inside the `kotlin.targets { }` block. Please use the `findByName("targetName")` method instead.

  Note that such accessors are still available in the case of `kotlin.targets`, for example, `kotlin.targets.linuxX64`.

Source directories configuration

The Kotlin Gradle plugin now adds Kotlin `SourceDirectorySet` as a `kotlin` extension to Java's `SourceSet` group. This makes it possible to configure source directories in the `build.gradle.kts` file similarly to how they are configured in Java, Groovy, and Scala:

```
sourceSets {
    main {
        kotlin {
            java.setSrcDirs(listOf("src/java"))
            kotlin.setSrcDirs(listOf("src/kotlin"))
        }
    }
}
```

You no longer need to use a deprecated Gradle convention and specify the source directories for Kotlin.

Remember that you can also use the `kotlin` extension to access `KotlinSourceSet`:

```
kotlin {
    sourceSets {
        main {
        // ...
        }
    }
}
```

New method for JVM toolchain configuration

This release provides a new `jvmToolchain()` method for enabling the JVM toolchain feature. If you don't need any additional configuration fields, such as `implementation` or `vendor`, you can use this method from the Kotlin extension:

```
kotlin {
    jvmToolchain(17)
}
```

This simplifies the Kotlin project setup process without any additional configuration. Before this release, you could specify the JDK version only in the following way:

```
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(17))
    }
}
```

Standard library

Kotlin 1.7.20 offers new extension functions for the `java.nio.file.Path` class, which allows you to walk through a file tree:

- `walk()` lazily traverses the file tree rooted at the specified path.
- `fileVisitor()` makes it possible to create a `FileVisitor` separately. `FileVisitor` defines actions on directories and files when traversing them.

- `visitFileTree(fileVisitor: FileVisitor, ...)` consumes a ready `FileVisitor` and uses `java.nio.file.Files.walkFileTree()` under the hood.
- `visitFileTree(..., builderAction: FileVisitorBuilder.() -> Unit)` creates a `FileVisitor` with the `builderAction` and calls the `visitFileTree(fileVisitor, ...)` function.
- `FileVisitResult`, return type of `FileVisitor`, has the `CONTINUE` default value that continues the processing of the file.

> The new extension functions for `java.nio.file.Path` are Experimental. They may be changed at any time. Opt-in is required (see details below), and you should use them only for evaluation purposes.

Here are some things you can do with these new extension functions:

- Explicitly create a `FileVisitor` and then use:

```
val cleanVisitor = fileVisitor {
    onPreVisitDirectory { directory, attributes ->
        // Some logic on visiting directories
        FileVisitResult.CONTINUE
    }

    onVisitFile { file, attributes ->
        // Some logic on visiting files
        FileVisitResult.CONTINUE
    }
}

// Some logic may go here

projectDirectory.visitFileTree(cleanVisitor)
```

- Create a `FileVisitor` with the `builderAction` and use it immediately:

```
projectDirectory.visitFileTree {
// Definition of the builderAction:
    onPreVisitDirectory { directory, attributes ->
        // Some logic on visiting directories
        FileVisitResult.CONTINUE
    }

    onVisitFile { file, attributes ->
        // Some logic on visiting files
        FileVisitResult.CONTINUE
    }
}
```

- Traverse a file tree rooted at the specified path with the `walk()` function:

```
@OptIn(kotlin.io.path.ExperimentalPathApi::class)
fun traverseFileTree() {
    val cleanVisitor = fileVisitor {
        onPreVisitDirectory { directory, _ ->
            if (directory.name == "build") {
                directory.toFile().deleteRecursively()
                FileVisitResult.SKIP_SUBTREE
            } else {
                FileVisitResult.CONTINUE
            }
        }

        onVisitFile { file, _ ->
            if (file.extension == "class") {
                file.deleteExisting()
            }
            FileVisitResult.CONTINUE
        }
    }

    val rootDirectory = createTempDirectory("Project")

    rootDirectory.resolve("src").let { srcDirectory ->
        srcDirectory.createDirectory()
        srcDirectory.resolve("A.kt").createFile()
        srcDirectory.resolve("A.class").createFile()
    }

    rootDirectory.resolve("build").let { buildDirectory ->
        buildDirectory.createDirectory()
        buildDirectory.resolve("Project.jar").createFile()
    }


// Use walk function:
    val directoryStructure = rootDirectory.walk(PathWalkOption.INCLUDE_DIRECTORIES)
        .map { it.relativeTo(rootDirectory).toString() }
        .toList().sorted()
    assertPrints(directoryStructure, "[, build, build/Project.jar, src, src/A.class, src/A.kt]")

    rootDirectory.visitFileTree(cleanVisitor)

    val directoryStructureAfterClean = rootDirectory.walk(PathWalkOption.INCLUDE_DIRECTORIES)
        .map { it.relativeTo(rootDirectory).toString() }
        .toList().sorted()
    assertPrints(directoryStructureAfterClean, "[, src, src/A.kt]")
}
```

As is usual for an experimental API, the new extensions require an opt-in: `@OptIn(kotlin.io.path.ExperimentalPathApi::class)` or `@kotlin.io.path.ExperimentalPathApi`. Alternatively, you can use a compiler option: `-opt-in=kotlin.io.path.ExperimentalPathApi`.

We would appreciate your feedback on the `walk()` function and the visit extension functions in YouTrack.

Documentation updates

Since the previous release, the Kotlin documentation has received some notable changes:

Revamped and improved pages

- Basic types overview – learn about the basic types used in Kotlin: numbers, Booleans, characters, strings, arrays, and unsigned integer numbers.
- IDEs for Kotlin development – see the list of IDEs with official Kotlin support and tools that have community-supported plugins.

New articles in the Kotlin Multiplatform journal

- Native and cross-platform app development: how to choose? – check out our overview and advantages of cross-platform app development and the native approach.
- The six best cross-platform app development frameworks – read about the key aspects to help you choose the right framework for your cross-platform project.

New and updated tutorials

- Get started with Kotlin Multiplatform – learn about cross-platform mobile development with Kotlin and create an app that works on both Android and iOS.
- Build a web application with React and Kotlin/JS – create a browser app exploring Kotlin's DSLs and features of a typical React program.

Changes in release documentation

We no longer provide a list of recommended kotlinx libraries for each release. This list included only the versions recommended and tested with Kotlin itself. It didn't take into account that some libraries depend on each other and require a special kotlinx version, which may differ from the recommended Kotlin version.

We're working on finding a way to provide information on how libraries interrelate and depend on each other so that it will be clear which kotlinx library version you should use when you upgrade the Kotlin version in your project.

Install Kotlin 1.7.20

IntelliJ IDEA 2021.3, 2022.1, and 2022.2 automatically suggest updating the Kotlin plugin to 1.7.20.

> For Android Studio Dolphin (213), Electric Eel (221), and Flamingo (222), the Kotlin plugin 1.7.20 will be delivered with upcoming Android Studios updates.

The new command-line compiler is available for download on the GitHub release page.

Compatibility guide for Kotlin 1.7.20

Although Kotlin 1.7.20 is an incremental release, there are still incompatible changes we had to make to limit spread of the issues introduced in Kotlin 1.7.0.

Find the detailed list of such changes in the Compatibility guide for Kotlin 1.7.20.

What's new in Kotlin 1.7.0

Released: 9 June 2022

Kotlin 1.7.0 has been released. It unveils the Alpha version of the new Kotlin/JVM K2 compiler, stabilizes language features, and brings performance improvements for the JVM, JS, and Native platforms.

Here is a list of the major updates in this version:

- The new Kotlin K2 compiler is in Alpha now, and it offers serious performance improvements. It is available only for the JVM, and none of the compiler plugins, including kapt, work with it.
- A new approach to the incremental compilation in Gradle. Incremental compilation is now also supported for changes made inside dependent non-Kotlin modules and is compatible with Gradle.
- We've stabilized opt-in requirement annotations, definitely non-nullable types, and builder inference.
- There's now an underscore operator for type args. You can use it to automatically infer a type of argument when other types are specified.
- This release allows implementation by delegation to an inlined value of an inline class. You can now create lightweight wrappers that do not allocate memory in most cases.

You can also find a short overview of the changes in this video:

Watch video online.

For information about the Kotlin release cycle, see Kotlin release process.

New Kotlin K2 compiler for the JVM in Alpha

This Kotlin release introduces the **Alpha** version of the new Kotlin K2 compiler. The new compiler aims to speed up the development of new language features, unify all of the platforms Kotlin supports, bring performance improvements, and provide an API for compiler extensions.

We've already published some detailed explanations of our new compiler and its benefits:

- The Road to the New Kotlin Compiler
- K2 Compiler: a Top-Down View

It's important to point out that with the Alpha version of the new K2 compiler we were primarily focused on performance improvements, and it only works with JVM projects. It doesn't support Kotlin/JS, Kotlin/Native, or other multi-platform projects, and none of compiler plugins, including kapt, work with it.

Our benchmarks show some outstanding results on our internal projects:

Project	Current Kotlin compiler performance	New K2 Kotlin compiler performance	Performance boost
Kotlin	2.2 KLOC/s	4.8 KLOC/s	~ x2.2
YouTrack	1.8 KLOC/s	4.2 KLOC/s	~ x2.3
IntelliJ IDEA	1.8 KLOC/s	3.9 KLOC/s	~ x2.2
Space	1.2 KLOC/s	2.8 KLOC/s	~ x2.3

The KLOC/s performance numbers stand for the number of thousands of lines of code that the compiler processes per second.

You can check out the performance boost on your JVM projects and compare it with the results of the old compiler. To enable the Kotlin K2 compiler, use the following compiler option:

```
-Xuse-k2
```

Also, the K2 compiler includes a number of bugfixes. Please note that even issues with **State: Open** from this list are in fact fixed in K2.

The next Kotlin releases will improve the stability of the K2 compiler and provide more features, so stay tuned!

If you face any performance issues with the Kotlin K2 compiler, please report them to our issue tracker.

Language

Kotlin 1.7.0 introduces support for implementation by delegation and a new underscore operator for type arguments. It also stabilizes several language features introduced as previews in previous releases:

- Implementation by delegation to inlined value of inline class
- Underscore operator for type arguments
- Stable builder inference
- Stable opt-in requirements
- Stable definitely non-nullable types

Allow implementation by delegation to an inlined value of an inline class

If you want to create a lightweight wrapper for a value or class instance, it's necessary to implement all interface methods by hand. Implementation by delegation solves this issue, but it did not work with inline classes before 1.7.0. This restriction has been removed, so you can now create lightweight wrappers that do not allocate memory in most cases.

```
interface Bar {
    fun foo() = "foo"
}

@JvmInline
value class BarWrapper(val bar: Bar): Bar by bar

fun main() {
    val bw = BarWrapper(object: Bar {})
    println(bw.foo())
}
```

Underscore operator for type arguments

Kotlin 1.7.0 introduces an underscore operator, `_`, for type arguments. You can use it to automatically infer a type argument when other types are specified:

```
abstract class SomeClass<T> {
    abstract fun execute(): T
}

class SomeImplementation : SomeClass<String>() {
    override fun execute(): String = "Test"
}

class OtherImplementation : SomeClass<Int>() {
    override fun execute(): Int = 42
}

object Runner {
    inline fun <reified S: SomeClass<T>, T> run(): T {
        return S::class.java.getDeclaredConstructor().newInstance().execute()
    }
}

fun main() {
    // T is inferred as String because SomeImplementation derives from SomeClass<String>
    val s = Runner.run<SomeImplementation, _>()
    assert(s == "Test")

    // T is inferred as Int because OtherImplementation derives from SomeClass<Int>
    val n = Runner.run<OtherImplementation, _>()
    assert(n == 42)
}
```

You can use the underscore operator in any position in the variables list to infer a type argument.

Stable builder inference

Builder inference is a special kind of type inference that is useful when calling generic builder functions. It helps the compiler infer the type arguments of a call using the type information about other calls inside its lambda argument.

Starting with 1.7.0, builder inference is automatically activated if a regular type inference cannot get enough information about a type without specifying the `-Xenable-builder-inference` compiler option, which was introduced in 1.6.0.

Learn how to write custom generic builders.

Stable opt-in requirements

Opt-in requirements are now Stable and do not require additional compiler configuration.

Before 1.7.0, the opt-in feature itself required the argument `-opt-in=kotlin.RequiresOptIn` to avoid a warning. It no longer requires this; however, you can still use the compiler argument `-opt-in` to opt-in for other annotations, a module.

Stable definitely non-nullable types

In Kotlin 1.7.0, definitely non-nullable types have been promoted to Stable. They provide better interoperability when extending generic Java classes and interfaces.

You can mark a generic type parameter as definitely non-nullable at the use site with the new syntax `T & Any`. The syntactic form comes from the notation for intersection types and is now limited to a type parameter with nullable upper bounds on the left side of `&` and a non-nullable `Any` on the right side:

```
fun <T> elvisLike(x: T, y: T & Any): T & Any = x ?: y

fun main() {
    // OK
    elvisLike<String>("", "").length
    // Error: 'null' cannot be a value of a non-null type
    elvisLike<String>("", null).length

    // OK
    elvisLike<String?>(null, "").length
    // Error: 'null' cannot be a value of a non-null type
    elvisLike<String?>(null, null).length
}
```

Learn more about definitely non-nullable types in this KEEP.

Kotlin/JVM

This release brings performance improvements for the Kotlin/JVM compiler and a new compiler option. Additionally, callable references to functional interface constructors have become Stable. Note that since 1.7.0, the default target version for Kotlin/JVM compilations is `1.8`.

- Compiler performance optimizations
- New compiler option `-Xjdk-release`
- Stable callable references to functional interface constructors
- Removed the JVM target version 1.6

Compiler performance optimizations

Kotlin 1.7.0 introduces performance improvements for the Kotlin/JVM compiler. According to our benchmarks, compilation time has been reduced by 10% on average compared to Kotlin 1.6.0. Projects with lots of usages of inline functions, for example, projects using `kotlinx.html`, will compile faster thanks to the improvements to the bytecode postprocessing.

New compiler option: -Xjdk-release

Kotlin 1.7.0 presents a new compiler option, `-Xjdk-release`. This option is similar to the javac's command-line `--release` option. The `-Xjdk-release` option controls the target bytecode version and limits the API of the JDK in the classpath to the specified Java version. For example, `kotlinc -Xjdk-release=1.8` won't allow referencing `java.lang.Module` even if the JDK in the dependencies is version 9 or higher.

> This option is not guaranteed to be effective for each JDK distribution.

Please leave your feedback on this YouTrack ticket.

Stable callable references to functional interface constructors

Callable references to functional interface constructors are now Stable. Learn how to migrate from an interface with a constructor function to a functional interface using callable references.

Please report any issues you find in YouTrack.

Removed JVM target version 1.6

The default target version for Kotlin/JVM compilations is `1.8`. The `1.6` target has been removed.

Please migrate to JVM target 1.8 or above. Learn how to update the JVM target version for:

- Gradle
- Maven
- The command-line compiler

Kotlin/Native

Kotlin 1.7.0 includes changes to Objective-C and Swift interoperability and stabilizes features that were introduced in previous releases. It also brings performance improvements for the new memory manager along with other updates:

- Performance improvements for the new memory manager
- Unified compiler plugin ABI with JVM and JS IR backends
- Support for standalone Android executables
- Interop with Swift async/await: returning `Void` instead of `KotlinUnit`
- Prohibited undeclared exceptions through Objective-C bridges
- Improved CocoaPods integration
- Overriding of the Kotlin/Native compiler download URL

Performance improvements for the new memory manager

> The new Kotlin/Native memory manager is in Alpha. It may change incompatibly and require manual migration in the future. We would appreciate your feedback in YouTrack.

The new memory manager is still in Alpha, but it is on its way to becoming Stable. This release delivers significant performance improvements for the new memory manager, especially in garbage collection (GC). In particular, concurrent implementation of the sweep phase, introduced in 1.6.20, is now enabled by default. This helps reduce the time the application is paused for GC. The new GC scheduler is better at choosing the GC frequency, especially for larger heaps.

Also, we've specifically optimized debug binaries, ensuring that the proper optimization level and link-time optimizations are used in the implementation code of the memory manager. This helped us improve execution time by roughly 30% for debug binaries on our benchmarks.

Try using the new memory manager in your projects to see how it works, and share your feedback with us in YouTrack.

Unified compiler plugin ABI with JVM and JS IR backends

Starting with Kotlin 1.7.0, the Kotlin Multiplatform Gradle plugin uses the embeddable compiler jar for Kotlin/Native by default. This feature was announced in 1.6.0 as Experimental, and now it's Stable and ready to use.

This improvement is very handy for library authors, as it improves the compiler plugin development experience. Before this release, you had to provide separate artifacts for Kotlin/Native, but now you can use the same compiler plugin artifacts for Native and other supported platforms.

> This feature might require plugin developers to take migration steps for their existing plugins.
>
> Learn how to prepare your plugin for the update in this YouTrack issue.

Support for standalone Android executables

Kotlin 1.7.0 provides full support for generating standard executables for Android Native targets. It was introduced in 1.6.20, and now it's enabled by default.

If you want to roll back to the previous behavior when Kotlin/Native generated shared libraries, use the following setting:

```
binaryOptions["androidProgramType"] = "nativeActivity"
```

Interop with Swift async/await: returning Void instead of KotlinUnit

Kotlin `suspend` functions now return the `Void` type instead of `KotlinUnit` in Swift. This is the result of the improved interop with Swift's `async` / `await`. This feature was introduced in 1.6.20, and this release enables this behavior by default.

You don't need to use the `kotlin.native.binary.unitSuspendFunctionObjCExport=proper` property anymore to return the proper type for such functions.

Prohibited undeclared exceptions through Objective-C bridges

When you call Kotlin code from Swift/Objective-C code (or vice versa) and this code throws an exception, it should be handled by the code where the exception occurred, unless you specifically allowed the forwarding of exceptions between languages with proper conversion (for example, using the `@Throws` annotation).

Previously, Kotlin had another unintended behavior where undeclared exceptions could "leak" from one language to another in some cases. Kotlin 1.7.0 fixes that issue, and now such cases lead to program termination.

So, for example, if you have a `{ throw Exception() }` lambda in Kotlin and call it from Swift, in Kotlin 1.7.0 it will terminate as soon as the exception reaches the Swift code. In previous Kotlin versions, such an exception could leak to the Swift code.

The `@Throws` annotation continues to work as before.

Improved CocoaPods integration

Starting with Kotlin 1.7.0, you no longer need to install the `cocoapods-generate` plugin if you want to integrate CocoaPods in your projects.

Previously, you needed to install both the CocoaPods dependency manager and the `cocoapods-generate` plugin to use CocoaPods, for example, to handle iOS dependencies in Kotlin Multiplatform Mobile projects.

Now setting up the CocoaPods integration is easier, and we've resolved the issue when `cocoapods-generate` couldn't be installed on Ruby 3 and later. Now the newest Ruby versions that work better on Apple M1 are also supported.

See how to set up the initial CocoaPods integration.

Overriding the Kotlin/Native compiler download URL

Starting with Kotlin 1.7.0, you can customize the download URL for the Kotlin/Native compiler. This is useful when external links on the CI are forbidden.

To override the default base URL `https://download.jetbrains.com/kotlin/native/builds`, use the following Gradle property:

```
kotlin.native.distribution.baseDownloadUrl=https://example.com
```

The downloader will append the native version and target OS to this base URL to ensure it downloads the actual compiler distribution.

Kotlin/JS

Kotlin/JS is receiving further improvements to the JS IR compiler backend along with other updates that can make your development experience better:

- Performance improvements for the new IR backend
- Minification for member names when using IR
- Support for older browsers via polyfills in the IR backend
- Dynamically load JavaScript modules from js expressions
- Specify environment variables for JavaScript test runners

Performance improvements for the new IR backend

This release has some major updates that should improve your development experience:

- Incremental compilation performance of Kotlin/JS has been significantly improved. It takes less time to build your JS projects. Incremental rebuilds should now be roughly on par with the legacy backend in many cases now.
- The Kotlin/JS final bundle requires less space, as we have significantly reduced the size of the final artifacts. We've measured up to a 20% reduction in the production bundle size compared to the legacy backend for some large projects.
- Type checking for interfaces has been improved by orders of magnitude.
- Kotlin generates higher-quality JS code

Minification for member names when using IR

The Kotlin/JS IR compiler now uses its internal information about the relationships of your Kotlin classes and functions to apply more efficient minification, shortening the names of functions, properties, and classes. This shrinks the resulting bundled applications.

This type of minification is automatically applied when you build your Kotlin/JS application in production mode and is enabled by default. To disable member name minification, use the `-Xir-minimized-member-names` compiler flag:

```
kotlin {
    js(IR) {
        compilations.all {
            compileKotlinTask.kotlinOptions.freeCompilerArgs += listOf("-Xir-minimized-member-
names=false")
        }
    }
}
```

Support for older browsers via polyfills in the IR backend

The IR compiler backend for Kotlin/JS now includes the same polyfills as the legacy backend. This allows code compiled with the new compiler to run in older browsers that do not support all the methods from ES2015 used by the Kotlin standard library. Only those polyfills actually used by the project are included in the final bundle, which minimizes their potential impact on the bundle size.

This feature is enabled by default when using the IR compiler, and you don't need to configure it.

Dynamically load JavaScript modules from js expressions

When working with the JavaScript modules, most applications use static imports, whose use is covered with the JavaScript module integration. However, Kotlin/JS was missing a mechanism to load JavaScript modules dynamically at runtime in your applications.

Starting with Kotlin 1.7.0, the `import` statement from JavaScript is supported in `js` blocks, allowing you to dynamically bring packages into your application at runtime:

```
val myPackage = js("import('my-package')")
```

Specify environment variables for JavaScript test runners

To tune Node.js package resolution or pass external information to Node.js tests, you can now specify environment variables used by the JavaScript test runners. To define an environment variable, use the `environment()` function with a key-value pair inside the `testTask` block in your build script:

```
kotlin {
    js {
        nodejs {
            testTask {
                environment("key", "value")
            }
        }
    }
}
```

Standard library

In Kotlin 1.7.0, the standard library has received a range of changes and improvements. They introduce new features, stabilize experimental ones, and unify support for named capturing groups for Native, JS, and the JVM:

- min() and max() collection functions return as non-nullable
- Regular expression matching at specific indices
- Extended support of previous language and API versions
- Access to annotations via reflection
- Stable deep recursive functions
- Time marks based on inline classes for default time source
- New experimental extension functions for Java Optionals
- Support for named capturing groups in JS and Native

min() and max() collection functions return as non-nullable

In Kotlin 1.4.0, we renamed the `min()` and `max()` collection functions to `minOrNull()` and `maxOrNull()`. These new names better reflect their behavior – returning null if the receiver collection is empty. It also helped align the functions' behavior with naming conventions used throughout the Kotlin collections API.

The same was true of `minBy()`, `maxBy()`, `minWith()`, and `maxWith()`, which all got their *OrNull() synonyms in Kotlin 1.4.0. Older functions affected by this change were gradually deprecated.

Kotlin 1.7.0 reintroduces the original function names, but with a non-nullable return type. The new `min()`, `max()`, `minBy()`, `maxBy()`, `minWith()`, and `maxWith()` functions now strictly return the collection element or throw an exception.

```
fun main() {
    val numbers = listOf<Int>()
    println(numbers.maxOrNull()) // "null"
    println(numbers.max()) // "Exception in... Collection is empty."
}
```

Regular expression matching at specific indices

The `Regex.matchAt()` and `Regex.matchesAt()` functions, introduced in 1.5.30, are now Stable. They provide a way to check whether a regular expression has an exact match at a particular position in a `String` or `CharSequence`.

`matchesAt()` checks for a match and returns a boolean result:

```
fun main() {
    val releaseText = "Kotlin 1.7.0 is on its way!"
    // regular expression: one digit, dot, one digit, dot, one or more digits
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()

    println(versionRegex.matchesAt(releaseText, 0)) // "false"
    println(versionRegex.matchesAt(releaseText, 7)) // "true"
}
```

`matchAt()` returns the match if it's found, or `null` if it isn't:

```
fun main() {
    val releaseText = "Kotlin 1.7.0 is on its way!"
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()

    println(versionRegex.matchAt(releaseText, 0)) // "null"
    println(versionRegex.matchAt(releaseText, 7)?.value) // "1.7.0"
}
```

We'd be grateful for your feedback on this YouTrack issue.

Extended support for previous language and API versions

To support library authors developing libraries that are meant to be consumable in a wide range of previous Kotlin versions, and to address the increased frequency of major Kotlin releases, we have extended our support for previous language and API versions.

With Kotlin 1.7.0, we're supporting three previous language and API versions rather than two. This means Kotlin 1.7.0 supports the development of libraries targeting Kotlin versions down to 1.4.0. For more information on backward compatibility, see Compatibility options.

Access to annotations via reflection

The `KAnnotatedElement.findAnnotations()` extension function, which was first introduced in 1.6.0, is now Stable. This reflection function returns all annotations of a given type on an element, including individually applied and repeated annotations.

```
@Repeatable
annotation class Tag(val name: String)

@Tag("First Tag")
@Tag("Second Tag")
fun taggedFunction() {
    println("I'm a tagged function!")
}

fun main() {
    val x = ::taggedFunction
    val foo = x as KAnnotatedElement
    println(foo.findAnnotations<Tag>()) // [@Tag(name=First Tag), @Tag(name=Second Tag)]
}
```

Stable deep recursive functions

Deep recursive functions have been available as an experimental feature since Kotlin 1.4.0, and they are now Stable in Kotlin 1.7.0. Using `DeepRecursiveFunction`, you can define a function that keeps its stack on the heap instead of using the actual call stack. This allows you to run very deep recursive computations. To call a deep recursive function, `invoke` it.

In this example, a deep recursive function is used to calculate the depth of a binary tree recursively. Even though this sample function calls itself recursively 100,000 times, no `StackOverflowError` is thrown:

```
class Tree(val left: Tree?, val right: Tree?)

val calculateDepth = DeepRecursiveFunction<Tree?, Int> { t →
    if (t == null) 0 else maxOf(
        callRecursive(t.left),
        callRecursive(t.right)
    ) + 1
}

fun main() {
    // Generate a tree with a depth of 100_000
    val deepTree = generateSequence(Tree(null, null)) { prev →
        Tree(prev, null)
    }.take(100_000).last()

    println(calculateDepth(deepTree)) // 100000
}
```

Consider using deep recursive functions in your code where your recursion depth exceeds 1000 calls.

Time marks based on inline classes for default time source

Kotlin 1.7.0 improves the performance of time measurement functionality by changing the time marks returned by `TimeSource.Monotonic` into inline value classes. This means that calling functions like `markNow()`, `elapsedNow()`, `measureTime()`, and `measureTimedValue()` doesn't allocate wrapper classes for their `TimeMark` instances. Especially when measuring a piece of code that is part of a hot path, this can help minimize the performance impact of the measurement:

```
@OptIn(ExperimentalTime::class)
fun main() {
    val mark = TimeSource.Monotonic.markNow() // Returned `TimeMark` is inline class
    val elapsedDuration = mark.elapsedNow()
}
```

This optimization is only available if the time source from which the `TimeMark` is obtained is statically known to be `TimeSource.Monotonic`.

New experimental extension functions for Java Optionals

Kotlin 1.7.0 comes with new convenience functions that simplify working with `Optional` classes in Java. These new functions can be used to unwrap and convert optional objects on the JVM and help make working with Java APIs more concise.

The `getOrNull()`, `getOrDefault()`, and `getOrElse()` extension functions allow you to get the value of an `Optional` if it's present. Otherwise, you get `null`, a default value, or a value returned by a function, respectively:

```
val presentOptional = Optional.of("I'm here!")

println(presentOptional.getOrNull())
// "I'm here!"

val absentOptional = Optional.empty<String>()

println(absentOptional.getOrNull())
// null
println(absentOptional.getOrDefault("Nobody here!"))
// "Nobody here!"
println(absentOptional.getOrElse {
    println("Optional was absent!")
    "Default value!"
})
// "Optional was absent!"
// "Default value!"
```

The `toList()`, `toSet()`, and `asSequence()` extension functions convert the value of a present `Optional` to a list, set, or sequence, or return an empty collection otherwise. The `toCollection()` extension function appends the `Optional` value to an already existing destination collection:

```
val presentOptional = Optional.of("I'm here!")
val absentOptional = Optional.empty<String>()
println(presentOptional.toList() + "," + absentOptional.toList())
// ["I'm here!"], []
println(presentOptional.toSet() + "," + absentOptional.toSet())
// ["I'm here!"], []
val myCollection = mutableListOf<String>()
absentOptional.toCollection(myCollection)
println(myCollection)
// []
presentOptional.toCollection(myCollection)
println(myCollection)
// ["I'm here!"]
val list = listOf(presentOptional, absentOptional).flatMap { it.asSequence() }
println(list)
// ["I'm here!"]
```

These extension functions are being introduced as Experimental in Kotlin 1.7.0. You can learn more about `Optional` extensions in this KEEP. As always, we welcome your feedback in the Kotlin issue tracker.

Support for named capturing groups in JS and Native

Starting with Kotlin 1.7.0, named capturing groups are supported not only on the JVM, but on the JS and Native platforms as well.

To give a name to a capturing group, use the (`?<name>group`) syntax in your regular expression. To get the text matched by a group, call the newly introduced `MatchGroupCollection.get()` function and pass the group name.

Retrieve matched group value by name

Consider this example for matching city coordinates. To get a collection of groups matched by the regular expression, use `groups`. Compare retrieving a group's contents by its number (index) and by its name using `value`:

```
fun main() {
    val regex = "\\b(?<city>[A-Za-z\\s]+),\\s(?<state>[A-Z]{2}):\\s(?<areaCode>[0-9]{3})\\b".toRegex()
    val input = "Coordinates: Austin, TX: 123"
    val match = regex.find(input)!!
    println(match.groups["city"]?.value) // "Austin" - by name
    println(match.groups[2]?.value) // "TX" - by number
}
```

Named backreferencing

You can now also use group names when backreferencing groups. Backreferences match the same text that was previously matched by a capturing group. For this, use the `\k<name>` syntax in your regular expression:

```
fun backRef() {
    val regex = "(?<title>\\w+), yes \\k<title>".toRegex()
    val match = regex.find("Do you copy? Sir, yes Sir!")!!
    println(match.value) // "Sir, yes Sir"
    println(match.groups["title"]?.value) // "Sir"
}
```

Named groups in replacement expressions

Named group references can be used with replacement expressions. Consider the `replace()` function that substitutes all occurrences of the specified regular expression in the input with a replacement expression, and the `replaceFirst()` function that swaps the first match only.

Occurrences of `${name}` in the replacement string are substituted with the subsequences corresponding to the captured groups with the specified name. You can compare replacements in group references by name and index:

```
fun dateReplace() {
    val dateRegex = Regex("(?<dd>\\d{2})-(?<mm>\\d{2})-(?<yyyy>\\d{4})")
    val input = "Date of birth: 27-04-2022"
    println(dateRegex.replace(input, "\${yyyy}-\${mm}-\${dd}")) // "Date of birth: 2022-04-27" - by
name
    println(dateRegex.replace(input, "\$3-\$2-\$1")) // "Date of birth: 2022-04-27" - by number
}
```

Gradle

This release introduces new build reports, support for Gradle plugin variants, new statistics in kapt, and a lot more:

- A new approach to incremental compilation
- New build reports for tracking compiler performance
- Changes to the minimum supported versions of Gradle and the Android Gradle plugin
- Support for Gradle plugin variants
- Updates in the Kotlin Gradle plugin API
- Availability of the sam-with-receiver plugin via the plugins API
- Changes in compile tasks
- New statistics of generated files by each annotation processor in kapt

- Deprecation of the kotlin.compiler.execution.strategy system property
- Removal of deprecated options, methods, and plugins

A new approach to incremental compilation

> The new approach to incremental compilation is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below). We encourage you to use it only for evaluation purposes, and we would appreciate your feedback in YouTrack.

In Kotlin 1.7.0, we've reworked incremental compilation for cross-module changes. Now incremental compilation is also supported for changes made inside dependent non-Kotlin modules, and it is compatible with the Gradle build cache. Support for compilation avoidance has also been improved.

We expect you'll see the most significant benefit of the new approach if you use the build cache or frequently make changes in non-Kotlin Gradle modules. Our tests for the Kotlin project on the `kotlin-gradle-plugin` module show an improvement of greater than 80% for the changes after the cache hit.

To try this new approach, set the following option in your `gradle.properties`:

```
kotlin.incremental.useClasspathSnapshot=true
```

> The new approach to incremental compilation is currently available for the JVM backend in the Gradle build system only.

Learn how the new approach to incremental compilation is implemented under the hood in this blog post.

Our plan is to stabilize this technology and add support for other backends (JS, for instance) and build systems. We'd appreciate your reports in YouTrack about any issues or strange behavior you encounter in this compilation scheme. Thank you!

The Kotlin team is very grateful to Ivan Gavrilovic, Hung Nguyen, Cédric Champeau, and other external contributors for their help.

Build reports for Kotlin compiler tasks

> Kotlin build reports are Experimental. They may be dropped or changed at any time. Opt-in is required (see details below). Use them only for evaluation purposes. We appreciate your feedback on them in YouTrack.

Kotlin 1.7.0 introduces build reports that help track compiler performance. Reports contain the durations of different compilation phases and reasons why compilation couldn't be incremental.

Build reports come in handy when you want to investigate issues with compiler tasks, for example:

- When the Gradle build takes too much time and you want to understand the root cause of the poor performance.
- When the compilation time for the same project differs, sometimes taking seconds, sometimes taking minutes.

To enable build reports, declare where to save the build report output in `gradle.properties`:

```
kotlin.build.report.output=file
```

The following values (and their combinations) are available:

- `file` saves build reports in a local file.
- `build_scan` saves build reports in the `custom values` section of the build scan.

  > The Gradle Enterprise plugin limits the number of custom values and their length. In big projects, some values could be lost.

- `http` posts build reports using HTTP(S). The POST method sends metrics in the JSON format. Data may change from version to version. You can see the current version of the sent data in the Kotlin repository.

There are two common cases that analyzing build reports for long-running compilations can help you resolve:

- The build wasn't incremental. Analyze the reasons and fix underlying problems.
- The build was incremental, but took too much time. Try to reorganize source files — split big files, save separate classes in different files, refactor large classes, declare top-level functions in different files, and so on.

Learn more about new build reports in this blog post.

You are welcome to try using build reports in your infrastructure. If you have any feedback, encounter any issues, or want to suggest improvements, please don't hesitate to report them in our issue tracker. Thank you!

Bumping minimum supported versions

Starting with Kotlin 1.7.0, the minimum supported Gradle version is 6.7.1. We had to raise the version to support Gradle plugin variants and the new Gradle API. In the future, we should not have to raise the minimum supported version as often, thanks to the Gradle plugin variants feature.

Also, the minimal supported Android Gradle plugin version is now 3.6.4.

Support for Gradle plugin variants

Gradle 7.0 introduced a new feature for Gradle plugin authors — plugins with variants. This feature makes it easier to add support for new Gradle features while maintaining compatibility for Gradle versions below 7.1. Learn more about variant selection in Gradle.

With Gradle plugin variants, we can ship different Kotlin Gradle plugin variants for different Gradle versions. The goal is to support the base Kotlin compilation in the `main` variant, which corresponds to the oldest supported versions of Gradle. Each variant will have implementations for Gradle features from a corresponding release. The latest variant will support the widest Gradle feature set. With this approach, we can extend support for older Gradle versions with limited functionality.

Currently, there are only two variants of the Kotlin Gradle plugin:

- `main` for Gradle versions 6.7.1–6.9.3
- `gradle70` for Gradle versions 7.0 and higher

In future Kotlin releases, we may add more.

To check which variant your build uses, enable the `--info` log level and find a string in the output starting with `Using Kotlin Gradle plugin`, for example, `Using Kotlin Gradle plugin main variant`.

Here are workarounds for some known issues with variant selection in Gradle:

- ResolutionStrategy in pluginManagement is not working for plugins with multivariants
- Plugin variants are ignored when a plugin is added as the `buildSrc` common dependency

Leave your feedback on this YouTrack ticket.

Updates in the Kotlin Gradle plugin API

The Kotlin Gradle plugin API artifact has received several improvements:

- There are new interfaces for Kotlin/JVM and Kotlin/kapt tasks with user-configurable inputs.
- There is a new `KotlinBasePlugin` interface that all Kotlin plugins inherit from. Use this interface when you want to trigger some configuration action whenever any Kotlin Gradle plugin (JVM, JS, Multiplatform, Native, and other platforms) is applied:

  ```
  project.plugins.withType<org.jetbrains.kotlin.gradle.plugin.KotlinBasePlugin>() {
      // Configure your action here
  }
  ```

  You can leave your feedback about the `KotlinBasePlugin` in this YouTrack ticket.
- We've laid the groundwork for the Android Gradle plugin to configure Kotlin compilation within itself, meaning you won't need to add the Kotlin Android Gradle plugin to your build. Follow Android Gradle Plugin release announcements to learn about the added support and try it out!

The sam-with-receiver plugin is available via the plugins API

The sam-with-receiver compiler plugin is now available via the Gradle plugins DSL:

```
plugins {
    id("org.jetbrains.kotlin.plugin.sam.with.receiver") version "$kotlin_version"
}
```

Changes in compile tasks

Compile tasks have received lots of changes in this release:

- Kotlin compile tasks no longer inherit the Gradle `AbstractCompile` task. They inherit only the `DefaultTask`.
- The `AbstractCompile` task has the `sourceCompatibility` and `targetCompatibility` inputs. Since the `AbstractCompile` task is no longer inherited, these inputs are no longer available in Kotlin users' scripts.
- The `SourceTask.stableSources` input is no longer available, and you should use the `sources` input. `setSource(...)` methods are still available.
- All compile tasks now use the `libraries` input for a list of libraries required for compilation. The `KotlinCompile` task still has the deprecated Kotlin property `classpath`, which will be removed in future releases.
- Compile tasks still implement the `PatternFilterable` interface, which allows the filtering of Kotlin sources. The `sourceFilesExtensions` input was removed in favor of using `PatternFilterable` methods.

- The deprecated `Gradle destinationDir: File` output was replaced with the `destinationDirectory: DirectoryProperty` output.
- The Kotlin/Native `AbstractNativeCompile` task now inherits the `AbstractKotlinCompileTool` base class. This is an initial step toward integrating Kotlin/Native build tools into all the other tools.

Please leave your feedback in this YouTrack ticket.

Statistics of generated files by each annotation processor in kapt

The `kotlin-kapt` Gradle plugin already reports performance statistics for each processor. Starting with Kotlin 1.7.0, it can also report statistics on the number of generated files for each annotation processor.

This is useful to track if there are unused annotation processors as a part of the build. You can use the generated report to find modules that trigger unnecessary annotation processors and update the modules to prevent that.

Enable the statistics in two steps:

- Set the `showProcessorStats` flag to `true` in your `build.gradle.kts`:

```
kapt {
    showProcessorStats = true
}
```

- Set the `kapt.verbose` Gradle property to `true` in your `gradle.properties`:

```
kapt.verbose=true
```

> You can also enable verbose output with the command line option `verbose`.

The statistics will appear in the logs with the `info` level. You'll see the `Annotation processor stats:` line followed by statistics on the execution time of each annotation processor. After these lines, there will be the `Generated files report:` line followed by statistics on the number of generated files for each annotation processor. For example:

```
[INFO] Annotation processor stats:
[INFO] org.mapstruct.ap.MappingProcessor: total: 290 ms, init: 1 ms, 3 round(s): 289 ms, 0 ms, 0 ms
[INFO] Generated files report:
[INFO] org.mapstruct.ap.MappingProcessor: total sources: 2, sources per round: 2, 0, 0
```

Please leave your feedback in this YouTrack ticket.

Deprecation of the kotlin.compiler.execution.strategy system property

Kotlin 1.6.20 introduced new properties for defining a Kotlin compiler execution strategy. In Kotlin 1.7.0, a deprecation cycle has started for the old system property `kotlin.compiler.execution.strategy` in favor of the new properties.

When using the `kotlin.compiler.execution.strategy` system property, you'll receive a warning. This property will be deleted in future releases. To preserve the old behavior, replace the system property with the Gradle property of the same name. You can do this in `gradle.properties`, for example:

```
kotlin.compiler.execution.strategy=out-of-process
```

You can also use the compile task property `compilerExecutionStrategy`. Learn more about this on the Compiler execution strategy page.

Removal of deprecated options, methods, and plugins

Removal of the useExperimentalAnnotation method

In Kotlin 1.7.0, we completed the deprecation cycle for the `useExperimentalAnnotation` Gradle method. Use `optIn()` instead to opt in to using an API in a module.

For example, if your Gradle module is multiplatform:

```
sourceSets {
    all {
        languageSettings.optIn("org.mylibrary.OptInAnnotation")
    }
}
```

Learn more about opt-in requirements in Kotlin.

Removal of deprecated compiler options

We've completed the deprecation cycle for several compiler options:

- The `kotlinOptions.jdkHome` compiler option was deprecated in 1.5.30 and has been removed in the current release. Gradle builds now fail if they contain this option. We encourage you to use Java toolchains, which have been supported since Kotlin 1.5.30.
- The deprecated `noStdlib` compiler option has also been removed. The Gradle plugin uses the `kotlin.stdlib.default.dependency=true` property to control whether the Kotlin standard library is present.

The compiler arguments `-jdkHome` and `-no-stdlib` are still available.

Removal of deprecated plugins

In Kotlin 1.4.0, the `kotlin2js` and `kotlin-dce-plugin` plugins were deprecated, and they have been removed in this release. Instead of `kotlin2js`, use the new `org.jetbrains.kotlin.js` plugin. Dead code elimination (DCE) works when the Kotlin/JS Gradle plugin is properly configured.

In Kotlin 1.6.0, we changed the deprecation level of the `KotlinGradleSubplugin` class to `ERROR`. Developers used this class for writing compiler plugins. In this release, this class has been removed. Use the `KotlinCompilerPluginSupportPlugin` class instead.

The best practice is to use Kotlin plugins with versions 1.7.0 and higher throughout your project.

Removal of the deprecated coroutines DSL option and property

We removed the deprecated `kotlin.experimental.coroutines` Gradle DSL option and the `kotlin.coroutines` property used in `gradle.properties`. Now you can just use *suspending functions* or add the `kotlinx.coroutines` dependency to your build script.

Learn more about coroutines in the Coroutines guide.

Removal of the type cast in the toolchain extension method

Before Kotlin 1.7.0, you had to do the type cast into the `JavaToolchainSpec` class when configuring the Gradle toolchain with Kotlin DSL:

```
kotlin {
    jvmToolchain {
        (this as JavaToolchainSpec).languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)
    }
}
```

Now, you can omit the `(this as JavaToolchainSpec)` part:

```
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)
    }
}
```

Migrating to Kotlin 1.7.0

Install Kotlin 1.7.0

IntelliJ IDEA 2022.1 and Android Studio Chipmunk (212) automatically suggest updating the Kotlin plugin to 1.7.0.

> For IntelliJ IDEA 2022.2, and Android Studio Dolphin (213) or Android Studio Electric Eel (221), the Kotlin plugin 1.7.0 will be delivered with upcoming IntelliJ IDEA and Android Studios updates.

The new command-line compiler is available for download on the GitHub release page.

Migrate existing or start a new project with Kotlin 1.7.0

- To migrate existing projects to Kotlin 1.7.0, change the Kotlin version to `1.7.0` and reimport your Gradle or Maven project. Learn how to update to Kotlin 1.7.0.
- To start a new project with Kotlin 1.7.0, update the Kotlin plugin and run the Project Wizard from **File** | **New** | **Project**.

Compatibility guide for Kotlin 1.7.0

Kotlin 1.7.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of such changes in the Compatibility guide for Kotlin 1.7.0.

Compatibility guide for Kotlin 1.7.20

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

Usually incompatible changes happen only in feature releases, but this time we have to introduce two such changes in an incremental release to limit spread of the problems introduced by changes in Kotlin 1.7.

This document summarizes them, providing a reference for migration from Kotlin 1.7.0 and 1.7.10 to Kotlin 1.7.20.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Rollback attempt to fix proper constraints processing

Issue: KT-53813

Component: Core language

Incompatible change type: source

Short summary: Rollback an attempt of fixing issues in type inference constraints processing appeared in 1.7.0 after implementing the change described in KT-52668. The attempt was made in 1.7.10, but it in turn introduced new problems.

Deprecation cycle:

- 1.7.20: Rollback to 1.7.0 behavior

Forbid some builder inference cases to avoid problematic interaction with multiple lambdas and resolution

Issue: KT-53797

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.7 introduced a feature called unrestricted builder inference, so that even the lambdas passed to parameters not annotated with `@BuilderInference` could benefit from the builder inference. However, that could cause several problems if more than one such lambda occurred in a function invocation.

Kotlin 1.7.20 will report an error if more than one lambda function having the corresponding parameter not annotated with `@BuilderInference` requires using builder inference to complete inferring the types in the lambda.

Deprecation cycle:

- 1.7.20: report an error on such lambda functions, `-XXLanguage:+NoBuilderInferenceWithoutAnnotationRestriction` can be used to temporarily revert to the pre-1.7.20 behavior

Compatibility guide for Kotlin 1.7.0

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.6 to Kotlin 1.7.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Make safe call result always nullable

Issue: KT-46860

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.7 will consider the type of safe call result always nullable, even when the receiver of the safe call is non-nullable

Deprecation cycle:

- <1.3: report a warning on an unnecessary safe call on non-nullable receivers
- 1.6.20: warn additionally that the result of an unnecessary safe call will change its type in the next version
- 1.7.0: change the type of safe call result to nullable, `-XXLanguage:-SafeCallsAreAlwaysNullable` can be used to temporarily revert to the pre-1.7 behavior

Prohibit the delegation of super calls to an abstract superclass member

Issues: KT-45508, KT-49017, KT-38078

Component: Core language

Incompatible change type: source

Short summary: Kotlin will report a compile error when an explicit or implicit super call is delegated to an *abstract* member of the superclass, even if there's a default implementation in a super interface

Deprecation cycle:

- 1.5.20: introduce a warning when non-abstract classes that do not override all abstract members are used
- 1.7.0: report an error if a super call, in fact, accesses an abstract member from a superclass
- 1.7.0: report an error if the `-Xjvm-default=all` or `-Xjvm-default=all-compatibility` compatibility modes are enabled; report an error in the progressive mode
- >=1.8.0: report an error in all cases

Prohibit exposing non-public types through public properties declared in a non-public primary constructor

Issue: KT-28078

Component: Core language

Incompatible change type: source

Short summary: Kotlin will prevent declaring public properties having non-public types in a private primary constructor. Accessing such properties from another package could lead to an `IllegalAccessError`

Deprecation cycle:

- 1.3.20: report a warning on a public property that has a non-public type and is declared in a non-public constructor
- 1.6.20: raise this warning to an error in the progressive mode
- 1.7.0: raise this warning to an error

Prohibit access to uninitialized enum entries qualified with the enum name

Issue: KT-41124

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.7 will prohibit access to uninitialized enum entries from the enum static initializer block when these entries are qualified with the enum name

Deprecation cycle:

- 1.7.0: report an error when uninitialized enum entries are accessed from the enum static initializer block

Prohibit computing constant values of complex boolean expressions in when condition branches and conditions of loops

Issue: KT-39883

Component: Core language

Incompatible change type: source

Short summary: Kotlin will no longer make exhaustiveness and control flow assumptions based on constant boolean expressions other than literal `true` and `false`

Deprecation cycle:

- 1.5.30: report a warning when exhaustiveness of `when` or control flow reachability is determined based on a complex constant boolean expression in `when` branch or loop condition
- 1.7.0: raise this warning to an error

Make when statements with enum, sealed, and Boolean subjects exhaustive by default

Issue: KT-47709

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.7 will report an error about the `when` statement with an enum, sealed, or Boolean subject being non-exhaustive

Deprecation cycle:

- 1.6.0: introduce a warning when the `when` statement with an enum, sealed, or Boolean subject is non-exhaustive (error in the progressive mode)
- 1.7.0: raise this warning to an error

Deprecate confusing grammar in when-with-subject

Issue: KT-48385

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 deprecated several confusing grammar constructs in `when` condition expressions

Deprecation cycle:

- 1.6.20: introduce a deprecation warning on the affected expressions
- 1.8.0: raise this warning to an error
- >= 1.8: repurpose some deprecated constructs for new language features

Type nullability enhancement improvements

Issue: KT-48623

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.7 will change how it loads and interprets type nullability annotations in Java code

Deprecation cycle:

- 1.4.30: introduce warnings for cases where more precise type nullability could lead to an error
- 1.7.0: infer more precise nullability of Java types, `-XXLanguage:-TypeEnhancementImprovementsInStrictMode` can be used to temporarily revert to the pre-1.7 behavior

Prevent implicit coercions between different numeric types

Issue: KT-48645

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin will avoid converting numeric values automatically to a primitive numeric type where only a downcast to that type was needed semantically

Deprecation cycle:

- < 1.5.30: the old behavior in all affected cases
- 1.5.30: fix the downcast behavior in generated property delegate accessors, `-Xuse-old-backend` can be used to temporarily revert to the pre-1.5.30 fix behavior
- >= 1.7.20: fix the downcast behavior in other affected cases

Deprecate the enable and the compatibility modes of the compiler option -Xjvm-default

Issue: KT-46329

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6.20 warns about the usage of `enable` and `compatibility` modes of the `-Xjvm-default` compiler option

Deprecation cycle:

- 1.6.20: introduce a warning on the `enable` and `compatibility` modes of the `-Xjvm-default` compiler option
- >= 1.8.0: raise this warning to an error

Prohibit calls to functions named suspend with a trailing lambda

Issue: KT-22562

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 no longer allows calling user functions named `suspend` that have the single argument of a functional type passed as a trailing lambda

Deprecation cycle:

- 1.3.0: introduce a warning on such function calls
- 1.6.0: raise this warning to an error
- 1.7.0: introduce changes to the language grammar so that `suspend` before `{` is parsed as a keyword

Prohibit smart cast on a base class property if the base class is from another module

Issue: KT-52629

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.7 will no longer allow smart casts on properties of a superclass if that class is located in another module

Deprecation cycle:

- 1.6.0: report a warning on a smart cast on a property declared in the superclass located in another module
- 1.7.0: raise this warning to an error, `-XXLanguage:-ProhibitSmartcastsOnPropertyFromAlienBaseClass` can be used to temporarily revert to the pre-1.7 behavior

Do not neglect meaningful constraints during type inference

Issue: KT-52668

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.4-1.6 neglected some type constraints during type inference due to an incorrect optimization. It could allow writing unsound code, causing `ClassCastException` at runtime. Kotlin 1.7 takes these constraints into account, thus prohibiting the unsound code

Deprecation cycle:

- 1.5.20: report a warning on expressions where a type mismatch would happen if all the type inference constraints were taken into account
- 1.7.0: take all the constraints into account, thus raising this warning to an error, `-XXLanguage:-ProperTypeInferenceConstraintsProcessing` can be used to temporarily revert to the pre-1.7 behavior

Standard library

Gradually change the return type of collection min and max functions to non-nullable

Issue: KT-38854

Component: kotlin-stdlib

Incompatible change type: source

Short summary: the return type of collection `min` and `max` functions will be changed to non-nullable in Kotlin 1.7

Deprecation cycle:

- 1.4.0: introduce `...OrNull` functions as synonyms and deprecate the affected API (see details in the issue)
- 1.5.0: raise the deprecation level of the affected API to an error
- 1.6.0: hide the deprecated functions from the public API
- 1.7.0: reintroduce the affected API but with non-nullable return type

Deprecate floating-point array functions: contains, indexOf, lastIndexOf

Issue: KT-28753

Component: kotlin-stdlib

Incompatible change type: source

Short summary: Kotlin deprecates floating-point array functions `contains`, `indexOf`, `lastIndexOf` that compare values using the IEEE-754 order instead of the total order

Deprecation cycle:

- 1.4.0: deprecate the affected functions with a warning
- 1.6.0: raise the deprecation level to an error
- 1.7.0: hide the deprecated functions from the public API

Migrate declarations from kotlin.dom and kotlin.browser packages to kotlinx.*

Issue: KT-39330

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: declarations from the `kotlin.dom` and `kotlin.browser` packages are moved to the corresponding `kotlinx.*` packages to prepare for extracting them from stdlib

Deprecation cycle:

- 1.4.0: introduce the replacement API in `kotlinx.dom` and `kotlinx.browser` packages
- 1.4.0: deprecate the API in `kotlin.dom` and `kotlin.browser` packages and propose the new API above as a replacement
- 1.6.0: raise the deprecation level to an error
- >= 1.8: remove the deprecated functions from stdlib
- >= 1.8: move the API in kotlinx.* packages to a separate library

Deprecate some JS-only API

Issue: KT-48587

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: a number of JS-only functions in stdlib are deprecated for removal. They include: `String.concat(String)`, `String.match(regex: String)`, `String.matches(regex: String)`, and the `sort` functions on arrays taking a comparison function, for example, `Array<out T>.sort(comparison: (a: T, b: T) -> Int)`

Deprecation cycle:

- 1.6.0: deprecate the affected functions with a warning
- 1.8.0: raise the deprecation level to an error
- 1.9.0: remove the deprecated functions from the public API

Tools

Remove KotlinGradleSubplugin class

Issue: KT-48831

Component: Gradle

Incompatible change type: source

Short summary: remove the `KotlinGradleSubplugin` class. Use the `KotlinCompilerPluginSupportPlugin` class instead

Deprecation cycle:

- 1.6.0: raise the deprecation level to an error
- 1.7.0: remove the deprecated class

Remove useIR compiler option

Issue: KT-48847

Component: Gradle

Incompatible change type: source

Short summary: remove the deprecated and hidden `useIR` compiler option

Deprecation cycle:

- 1.5.0: raise the deprecation level to a warning
- 1.6.0: hide the option
- 1.7.0: remove the deprecated option

Deprecate kapt.use.worker.api Gradle property

Issue: KT-48826

Component: Gradle

Incompatible change type: source

Short summary: deprecate the `kapt.use.worker.api` property that allowed to run kapt via Gradle Workers API (default: true)

Deprecation cycle:

- 1.6.20: raise the deprecation level to a warning
- >= 1.8.0: remove this property

Remove kotlin.experimental.coroutines Gradle DSL option and kotlin.coroutines Gradle property

Issue: KT-50494

Component: Gradle

Incompatible change type: source

Short summary: remove the `kotlin.experimental.coroutines` Gradle DSL option and the `kotlin.coroutines` property

Deprecation cycle:

- 1.6.20: raise the deprecation level to a warning
- 1.7.0: remove the DSL option, its enclosing `experimental` block, and the property

Deprecate useExperimentalAnnotation compiler option

Issue: KT-47763

Component: Gradle

Incompatible change type: source

Short summary: remove the hidden `useExperimentalAnnotation()` Gradle function used to opt in to using an API in a module. `optIn()` function can be used instead

Deprecation cycle:

- 1.6.0: hide the deprecation option
- 1.7.0: remove the deprecated option

Deprecate kotlin.compiler.execution.strategy system property

Issue: KT-51830

Component: Gradle

Incompatible change type: source

Short summary: deprecate the `kotlin.compiler.execution.strategy` system property used to choose a compiler execution strategy. Use the Gradle property `kotlin.compiler.execution.strategy` or the compile task property `compilerExecutionStrategy` instead

Deprecation cycle:

- 1.7.0: raise the deprecation level to a warning
- > 1.7.0: remove the property

Remove kotlinOptions.jdkHome compiler option

Issue: KT-46541

Component: Gradle

Incompatible change type: source

Short summary: remove the `kotlinOptions.jdkHome` compiler option used to include a custom JDK from the specified location into the classpath instead of the default `JAVA_HOME`. Use Java toolchains instead

Deprecation cycle:

- 1.5.30: raise the deprecation level to a warning
- > 1.7.0: remove the option

Remove noStdlib compiler option

Issue: KT-49011

Component: Gradle

Incompatible change type: source

Short summary: remove the `noStdlib` compiler option. The Gradle plugin uses the `kotlin.stdlib.default.dependency=true` property to control whether the Kotlin standard library is present

Deprecation cycle:

- 1.5.0: raise the deprecation level to a warning
- 1.7.0: remove the option

Remove kotlin2js and kotlin-dce-plugin plugins

Issue: KT-48276

Component: Gradle

Incompatible change type: source

Short summary: remove the `kotlin2js` and `kotlin-dce-plugin` plugins. Instead of `kotlin2js`, use the new `org.jetbrains.kotlin.js` plugin. Dead code elimination (DCE) works when the Kotlin/JS Gradle plugin is properly configured

Deprecation cycle:

- 1.4.0: raise the deprecation level to a warning
- 1.7.0: remove the plugins

Changes in compile tasks

Issue: KT-32805

Component: Gradle

Incompatible change type: source

Short summary: Kotlin compile tasks no longer inherit the Gradle `AbstractCompile` task and that's why the `sourceCompatibility` and `targetCompatibility` inputs are no longer available in Kotlin users' scripts. The `SourceTask.stableSources` input is no longer available. The `sourceFilesExtensions` input was removed. The deprecated `Gradle destinationDir: File` output was replaced with the `destinationDirectory: DirectoryProperty` output. The `classpath` property of the `KotlinCompile` task is deprecated

Deprecation cycle:

- 1.7.0: inputs are not available, the output is replaced, the `classpath` property is deprecated

What's new in Kotlin 1.6.20

Released: 4 April 2022

Kotlin 1.6.20 reveals previews of the future language features, makes the hierarchical structure the default for multiplatform projects, and brings evolutionary improvements to other components.

You can also find a short overview of the changes in this video:

Watch video online.

For information about the Kotlin release cycle, see Kotlin release process.

Language

In Kotlin 1.6.20, you can try two new language features:

- Prototype of context receivers for Kotlin/JVM
- Definitely non-nullable types

Prototype of context receivers for Kotlin/JVM

The feature is a prototype available only for Kotlin/JVM. With `-Xcontext-receivers` enabled, the compiler will produce pre-release binaries that cannot be used in production code. Use context receivers only in your toy projects. We appreciate your feedback in YouTrack.

With Kotlin 1.6.20, you are no longer limited to having one receiver. If you need more, you can make functions, properties, and classes context-dependent (or *contextual*) by adding context receivers to their declaration. A contextual declaration does the following:

- It requires all declared context receivers to be present in a caller's scope as implicit receivers.

- It brings declared context receivers into its body scope as implicit receivers.

```
interface LoggingContext {
    val log: Logger // This context provides a reference to a logger
}

context(LoggingContext)
fun startBusinessOperation() {
    // You can access the log property since LoggingContext is an implicit receiver
    log.info("Operation has started")
}

fun test(loggingContext: LoggingContext) {
    with(loggingContext) {
        // You need to have LoggingContext in a scope as an implicit receiver
        // to call startBusinessOperation()
        startBusinessOperation()
    }
}
```

To enable context receivers in your project, use the `-Xcontext-receivers` compiler option. You can find a detailed description of the feature and its syntax in the KEEP.

Please note that the implementation is a prototype:

- With `-Xcontext-receivers` enabled, the compiler will produce pre-release binaries that cannot be used in production code
- The IDE support for context receivers is minimal for now

Try the feature in your toy projects and share your thoughts and experience with us in this YouTrack issue. If you run into any problems, please file a new issue.

Definitely non-nullable types

> Definitely non-nullable types are in Beta. They are almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you have to make.

To provide better interoperability when extending generic Java classes and interfaces, Kotlin 1.6.20 allows you to mark a generic type parameter as definitely non-nullable on the use site with the new syntax `T & Any`. The syntactic form comes from a notation of intersection types and is now limited to a type parameter with nullable upper bounds on the left side of `&` and non-nullable `Any` on the right side:

```
fun <T> elvisLike(x: T, y: T & Any): T & Any = x ?: y

fun main() {
    // OK
    elvisLike<String>("", "").length
    // Error: 'null' cannot be a value of a non-null type
    elvisLike<String>("", null).length

    // OK
    elvisLike<String?>(null, "").length
    // Error: 'null' cannot be a value of a non-null type
    elvisLike<String?>(null, null).length
}
```

Set the language version to `1.7` to enable the feature:

Kotlin

```
kotlin {
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.7"
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.7'
        }
    }
}
```

Learn more about definitely non-nullable types in the KEEP.

Kotlin/JVM

Kotlin 1.6.20 introduces:

- Compatibility improvements of default methods in JVM interfaces: new `@JvmDefaultWithCompatibility` annotation for interfaces and compatibility changes in the `-Xjvm-default` modes
- Support for parallel compilation of a single module in the JVM backend
- Support for callable references to functional interface constructors

New @JvmDefaultWithCompatibility annotation for interfaces

Kotlin 1.6.20 introduces the new annotation `@JvmDefaultWithCompatibility`: use it along with the `-Xjvm-default=all` compiler option to create the default method in JVM interface for any non-abstract member in any Kotlin interface.

If there are clients that use your Kotlin interfaces compiled without the `-Xjvm-default=all` option, they may be binary-incompatible with the code compiled with this option. Before Kotlin 1.6.20, to avoid this compatibility issue, the recommended approach was to use the `-Xjvm-default=all-compatibility` mode and also the `@JvmDefaultWithoutCompatibility` annotation for interfaces that didn't need this type of compatibility.

This approach had some disadvantages:

- You could easily forget to add the annotation when a new interface was added.
- Usually there are more interfaces in non-public parts than in the public API, so you end up having this annotation in many places in your code.

Now, you can use the `-Xjvm-default=all` mode and mark interfaces with the `@JvmDefaultWithCompatibility` annotation. This allows you to add this annotation to all interfaces in the public API once, and you won't need to use any annotations for new non-public code.

Leave your feedback about this new annotation in this YouTrack ticket.

Compatibility changes in the -Xjvm-default modes

Kotlin 1.6.20 adds the option to compile modules in the default mode (the `-Xjvm-default=disable` compiler option) against modules compiled with the `-Xjvm-default=all` or `-Xjvm-default=all-compatibility` modes. As before, compilations will also be successful if all modules have the `-Xjvm-default=all` or `-Xjvm-default=all-compatibility` modes. You can leave your feedback in this YouTrack issue.

Kotlin 1.6.20 deprecates the `compatibility` and `enable` modes of the compiler option `-Xjvm-default`. There are changes in other modes' descriptions regarding the compatibility, but the overall logic remains the same. You can check out the updated descriptions.

For more information about default methods in the Java interop, see the interoperability documentation and this blog post.

Support for parallel compilation of a single module in the JVM backend

> Support for parallel compilation of a single module in the JVM backend is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

We are continuing our work to improve the new JVM IR backend compilation time. In Kotlin 1.6.20, we added the experimental JVM IR backend mode to compile all the files in a module in parallel. Parallel compilation can reduce the total compilation time by up to 15%.

Enable the experimental parallel backend mode with the compiler option `-Xbackend-threads`. Use the following arguments for this option:

- `N` is the number of threads you want to use. It should not be greater than your number of CPU cores; otherwise, parallelization stops being effective because of switching context between threads
- `0` to use a separate thread for each CPU core

Gradle can run tasks in parallel, but this type of parallelization doesn't help a lot when a project (or a major part of a project) is just one big task from Gradle's perspective. If you have a very big monolithic module, use parallel compilation to compile more quickly. If your project consists of lots of small modules and has a build parallelized by Gradle, adding another layer of parallelization may hurt performance because of context switching.

> Parallel compilation has some constraints:
>
> - It doesn't work with kapt because kapt disables the IR backend
> - It requires more JVM heap by design. The amount of heap is proportional to the number of threads

Support for callable references to functional interface constructors

> Support for callable references to functional interface constructors is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Support for callable references to functional interface constructors adds a source-compatible way to migrate from an interface with a constructor function to a functional interface.

Consider the following code:

```
interface Printer {
    fun print()
}

fun Printer(block: () -> Unit): Printer = object : Printer { override fun print() = block() }
```

With callable references to functional interface constructors enabled, this code can be replaced with just a functional interface declaration:

```
fun interface Printer {
    fun print()
}
```

Its constructor will be created implicitly, and any code using the `::Printer` function reference will compile. For example:

```
documentsStorage.addPrinter(::Printer)
```

Preserve the binary compatibility by marking the legacy function `Printer` with the `@Deprecated` annotation with `DeprecationLevel.HIDDEN`:

```
@Deprecated(message = "Your message about the deprecation", level = DeprecationLevel.HIDDEN)
fun Printer(...) {...}
```

Use the compiler option `-XXLanguage:+KotlinFunInterfaceConstructorReference` to enable this feature.

Kotlin/Native

Kotlin/Native 1.6.20 marks continued development of its new components. We've taken another step toward consistent experience with Kotlin on other platforms:

- An update on the new memory manager
- Concurrent implementation for the sweep phase in new memory manager
- Instantiation of annotation classes
- Interop with Swift async/await: returning Swift's Void instead of KotlinUnit
- Better stack traces with libbacktrace
- Support for standalone Android executables
- Performance improvements

- Improved error handling during cinterop modules import
- Support for Xcode 13 libraries

An update on the new memory manager

> The new Kotlin/Native memory manager is in Alpha. It may change incompatibly and require manual migration in the future. We would appreciate your feedback on it in YouTrack.

With Kotlin 1.6.20, you can try the Alpha version of the new Kotlin/Native memory manager. It eliminates the differences between the JVM and Native platforms to provide a consistent developer experience in multiplatform projects. For example, you'll have a much easier time creating new cross-platform mobile applications that work on both Android and iOS.

The new Kotlin/Native memory manager lifts restrictions on object-sharing between threads. It also provides leak-free concurrent programming primitives that are safe and don't require any special management or annotations.

The new memory manager will become the default in future versions, so we encourage you to try it now. Check out our blog post to learn more about the new memory manager and explore demo projects, or jump right to the migration instructions to try it yourself.

Try using the new memory manager on your projects to see how it works and share feedback in our issue tracker, YouTrack.

Concurrent implementation for the sweep phase in new memory manager

If you have already switched to our new memory manager, which was announced in Kotlin 1.6, you might notice a huge execution time improvement: our benchmarks show 35% improvement on average. Starting with 1.6.20, there is also a concurrent implementation for the sweep phase available for the new memory manager. This should also improve the performance and decrease the duration of garbage collector pauses.

To enable the feature for the new Kotlin/Native memory manager, pass the following compiler option:

```
-Xgc=cms
```

Feel free to share your feedback on the new memory manager performance in this YouTrack issue.

Instantiation of annotation classes

In Kotlin 1.6.0, instantiation of annotation classes became Stable for Kotlin/JVM and Kotlin/JS. The 1.6.20 version delivers support for Kotlin/Native.

Learn more about instantiation of annotation classes.

Interop with Swift async/await: returning Void instead of KotlinUnit

> Concurrency interoperability with Swift async/await is Experimental. It may be dropped or changed at any time. You should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

We've continued working on the experimental interop with Swift's async/await (available since Swift 5.5). Kotlin 1.6.20 differs from previous versions in the way it works with `suspend` functions with the `Unit` return type.

Previously, such functions were presented in Swift as `async` functions returning `KotlinUnit`. However, the proper return type for them is `Void`, similar to non-suspending functions.

To avoid breaking the existing code, we're introducing a Gradle property that makes the compiler translate `Unit`-returning suspend functions to `async` Swift with the `Void` return type:

```
# gradle.properties
kotlin.native.binary.unitSuspendFunctionObjCExport=proper
```

We plan to make this behavior the default in future Kotlin releases.

Better stack traces with libbacktrace

> Using libbacktrace for resolving source locations is Experimental. It may be dropped or changed at any time. You should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin/Native is now able to produce detailed stack traces with file locations and line numbers for better debugging of `linux*` (except `linuxMips32` and `linuxMipsel32`) and `androidNative*` targets.

This feature uses the libbacktrace library under the hood. Take a look at the following code to see an example of the difference:

```
fun main() = bar()
fun bar() = baz()
inline fun baz() {
    error("")
}
```

- **Before 1.6.20:**

```
Uncaught Kotlin exception: kotlin.IllegalStateException:
   at 0   example.kexe        0x227190       kfun:kotlin.Throwable#<init>(kotlin.String?){} + 96
   at 1   example.kexe        0x221e4c       kfun:kotlin.Exception#<init>(kotlin.String?){} + 92
   at 2   example.kexe        0x221f4c       kfun:kotlin.RuntimeException#<init>(kotlin.String?){} +
92
   at 3   example.kexe        0x22234c       kfun:kotlin.IllegalStateException#<init>(kotlin.String?)
{} + 92
   at 4   example.kexe        0x25d708       kfun:#bar(){} + 104
   at 5   example.kexe        0x25d68c       kfun:#main(){} + 12
```

- **1.6.20 with libbacktrace:**

```
Uncaught Kotlin exception: kotlin.IllegalStateException:
   at 0   example.kexe        0x229550    kfun:kotlin.Throwable#<init>(kotlin.String?){} + 96
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Throwable.kt:24:37)
   at 1   example.kexe        0x22420c    kfun:kotlin.Exception#<init>(kotlin.String?){} + 92
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:23:44)
   at 2   example.kexe        0x22430c    kfun:kotlin.RuntimeException#<init>(kotlin.String?){} + 92
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:34:44)
   at 3   example.kexe        0x22470c    kfun:kotlin.IllegalStateException#<init>(kotlin.String?){} +
92 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:70:44)
   at 4   example.kexe        0x25fac8    kfun:#bar(){} + 104 [inlined]
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/libraries/stdlib/src/kotlin/util/Preconditions.kt:143:56)
   at 5   example.kexe        0x25fac8    kfun:#bar(){} + 104 [inlined]
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:4:5)
   at 6   example.kexe        0x25fac8    kfun:#bar(){} + 104
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:2:13)
   at 7   example.kexe        0x25fa4c    kfun:#main(){} + 12
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:1:14)
```

On Apple targets, which already had file locations and line numbers in stack traces, libbacktrace provides more details for inline function calls:

- **Before 1.6.20:**

```
Uncaught Kotlin exception: kotlin.IllegalStateException:
   at 0   example.kexe    0x10a85a8f8    kfun:kotlin.Throwable#<init>(kotlin.String?){} + 88
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Throwable.kt:24:37)
   at 1   example.kexe    0x10a855846    kfun:kotlin.Exception#<init>(kotlin.String?){} + 86
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:23:44)
   at 2   example.kexe    0x10a855936    kfun:kotlin.RuntimeException#<init>(kotlin.String?){} + 86
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:34:44)
   at 3   example.kexe    0x10a855c86    kfun:kotlin.IllegalStateException#<init>(kotlin.String?){} +
86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:70:44)
   at 4   example.kexe    0x10a8489a5    kfun:#bar(){} + 117
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:2:1)
   at 5   example.kexe    0x10a84891c    kfun:#main(){} + 12
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:1:14)
...
```

- **1.6.20 with libbacktrace:**

```
Uncaught Kotlin exception: kotlin.IllegalStateException:
   at 0   example.kexe    0x10669bc88    kfun:kotlin.Throwable#<init>(kotlin.String?){} + 88
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Throwable.kt:24:37)
   at 1   example.kexe    0x106696bd6    kfun:kotlin.Exception#<init>(kotlin.String?){} + 86
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:23:44)
   at 2   example.kexe    0x106696cc6    kfun:kotlin.RuntimeException#<init>(kotlin.String?){} + 86
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:34:44)
   at 3   example.kexe    0x106697016    kfun:kotlin.IllegalStateException#<init>(kotlin.String?){} +
86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:70:44)
   at 4   example.kexe    0x106689d35    kfun:#bar(){} + 117 [inlined]
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/libraries/stdlib/src/kotlin/util/Preconditions.kt:143:56)
>>  at 5   example.kexe    0x106689d35    kfun:#bar(){} + 117 [inlined]
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:4:5)
   at 6   example.kexe    0x106689d35    kfun:#bar(){} + 117
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:2:13)
   at 7   example.kexe    0x106689cac    kfun:#main(){} + 12
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:1:14)
...
```

To produce better stack traces with libbacktrace, add the following line to `gradle.properties`:

```
# gradle.properties
kotlin.native.binary.sourceInfoType=libbacktrace
```

Please tell us how debugging Kotlin/Native with libbacktrace works for you in this YouTrack issue.

Support for standalone Android executables

Previously, Android Native executables in Kotlin/Native were not actually executables but shared libraries that you could use as a NativeActivity. Now there's an option to generate standard executables for Android Native targets.

For that, in the `build.gradle(.kts)` part of your project, configure the executable block of your `androidNative` target. Add the following binary option:

```
kotlin {
    androidNativeX64("android") {
        binaries {
            executable {
                binaryOptions["androidProgramType"] = "standalone"
            }
        }
    }
}
```

Note that this feature will become the default in Kotlin 1.7.0. If you want to preserve the current behavior, use the following setting:

```
binaryOptions["androidProgramType"] = "nativeActivity"
```

Thanks to Mattia Iavarone for the implementation!

Performance improvements

We are working hard on Kotlin/Native to speed up the compilation process and improve your developing experience.

Kotlin 1.6.20 brings some performance updates and bug fixes that affect the LLVM IR that Kotlin generates. According to the benchmarks on our internal projects, we achieved the following performance boosts on average:

- 15% reduction in execution time
- 20% reduction in the code size of both release and debug binaries
- 26% reduction in the compilation time of release binaries

These changes also provide a 10% reduction in compilation time for a debug binary on a large internal project.

To achieve this, we've implemented static initialization for some of the compiler-generated synthetic objects, improved the way we structure LLVM IR for every function, and optimized the compiler caches.

Improved error handling during cinterop modules import

This release introduces improved error handling for cases where you import an Objective-C module using the `cinterop` tool (as is typical for CocoaPods pods). Previously, if you got an error while trying to work with an Objective-C module (for instance, when dealing with a compilation error in a header), you received an uninformative error message, such as `fatal error: could not build module $name`. We expanded upon this part of the `cinterop` tool, so you'll get an error message with an extended description.

Support for Xcode 13 libraries

Libraries delivered with Xcode 13 have full support as of this release. Feel free to access them from anywhere in your Kotlin code.

Kotlin Multiplatform

1.6.20 brings the following notable updates to Kotlin Multiplatform:

- Hierarchical structure support is now default for all new multiplatform projects
- Kotlin CocoaPods Gradle plugin received several useful features for CocoaPods integration

Hierarchical structure support for multiplatform projects

Kotlin 1.6.20 comes with hierarchical structure support enabled by default. Since introducing it in Kotlin 1.4.0, we've significantly improved the frontend and made IDE import stable.

Previously, there were two ways to add code in a multiplatform project. The first was to insert it in a platform-specific source set, which is limited to one target and can't be reused by other platforms. The second is to use a common source set shared across all the platforms that are currently supported by Kotlin.

Now you can share source code among several similar native targets that reuse a lot of the common logic and third-party APIs. The technology will provide the correct default dependencies and find the exact API available in the shared code. This eliminates a complex build setup and having to use workarounds to get IDE support for sharing source sets among native targets. It also helps prevent unsafe API usages meant for a different target.

The technology will come in handy for library authors, too, as a hierarchical project structure allows them to publish and consume libraries with common APIs for a subset of targets.

By default, libraries published with the hierarchical project structure are compatible only with hierarchical structure projects.

Better code-sharing in your project

Without hierarchical structure support, there is no straightforward way to share code across *some* but not *all* Kotlin targets. One popular example is sharing code across all iOS targets and having access to iOS-specific dependencies, like Foundation.

Thanks to the hierarchical project structure support, you can now achieve this out of the box. In the new structure, source sets form a hierarchy. You can use platform-specific language features and dependencies available for each target that a given source set compiles to.

For example, consider a typical multiplatform project with two targets — `iosArm64` and `iosX64` for iOS devices and simulators. The Kotlin tooling understands that both targets have the same function and allows you to access that function from the intermediate source set, `iosMain`.



iOS hierarchy example

The Kotlin toolchain provides the correct default dependencies, like Kotlin/Native stdlib or native libraries. Moreover, Kotlin tooling will try its best to find exactly the API surface area available in the shared code. This prevents such cases as, for example, the use of a macOS-specific function in code shared for Windows.

More opportunities for library authors

When a multiplatform library is published, the API of its intermediate source sets is now properly published alongside it, making it available for consumers. Again, the Kotlin toolchain will automatically figure out the API available in the consumer source set while carefully watching out for unsafe usages, like using an API meant for the JVM in JS code. Learn more about sharing code in libraries.

Configuration and setup

Starting with Kotlin 1.6.20, all your new multiplatform projects will have a hierarchical project structure. No additional setup is required.

- If you've already turned it on manually, you can remove the deprecated options from `gradle.properties`:

```
# gradle.properties
kotlin.mpp.enableGranularSourceSetsMetadata=true
kotlin.native.enableDependencyPropagation=false // or 'true', depending on your previous setup
```

- For Kotlin 1.6.20, we recommend using Android Studio 2021.1.1 (Bumblebee) or later to get the best experience.
- You can also opt out. To disable hierarchical structure support, set the following options in `gradle.properties`:

```
# gradle.properties
kotlin.mpp.hierarchicalStructureSupport=false
```

Leave your feedback

This is a significant change to the whole ecosystem. We would appreciate your feedback to help make it even better.

Try it now and report any difficulties you encounter to our issue tracker.

Kotlin CocoaPods Gradle plugin

To simplify CocoaPods integration, Kotlin 1.6.20 delivers the following features:

- The CocoaPods plugin now has tasks that build XCFrameworks with all registered targets and generate the Podspec file. This can be useful when you don't want to integrate with Xcode directly, but you want to build artifacts and deploy them to your local CocoaPods repository.

  Learn more about building XCFrameworks.

- If you use CocoaPods integration in your projects, you're used to specifying the required Pod version for the entire Gradle project. Now you have more options:
  - Specify the Pod version directly in the `cocoapods` block
  - Continue using a Gradle project version

  If none of these properties is configured, you'll get an error.

- You can now configure the CocoaPod name in the `cocoapods` block instead of changing the name of the whole Gradle project.
- The CocoaPods plugin introduces a new `extraSpecAttributes` property, which you can use to configure properties in a Podspec file that were previously hard-coded, like `libraries` or `vendored_frameworks`.

```
kotlin {
    cocoapods {
        version = "1.0"
        name = "MyCocoaPod"
        extraSpecAttributes["social_media_url"] = 'https://twitter.com/kotlin'
        extraSpecAttributes["vendored_frameworks"] = 'CustomFramework.xcframework'
        extraSpecAttributes["libraries"] = 'xml'
    }
}
```

See the full Kotlin CocoaPods Gradle plugin DSL reference.

Kotlin/JS

Kotlin/JS improvements in 1.6.20 mainly affect the IR compiler:

- Incremental compilation for development binaries (IR)
- Lazy initialization of top-level properties by default (IR)
- Separate JS files for project modules by default (IR)
- Char class optimization (IR)
- Export improvements (both IR and legacy backends)
- @AfterTest guarantees for asynchronous tests

Incremental compilation for development binaries with IR compiler

To make Kotlin/JS development with the IR compiler more efficient, we're introducing a new *incremental compilation* mode.

When building **development binaries** with the `compileDevelopmentExecutableKotlinJs` Gradle task in this mode, the compiler caches the results of previous compilations on the module level. It uses the cached compilation results for unchanged source files during subsequent compilations, making them complete more quickly, especially with small changes. Note that this improvement exclusively targets the development process (shortening the edit-build-debug cycle) and doesn't affect the building of production artifacts.

To enable incremental compilation for development binaries, add the following line to the project's `gradle.properties`:

```
# gradle.properties
kotlin.incremental.js.ir=true // false by default
```

In our test projects, the new mode made incremental compilation up to 30% faster. However, the clean build in this mode became slower because of the need to create and populate the caches.

Please tell us what you think of using incremental compilation with your Kotlin/JS projects in this YouTrack issue.

Lazy initialization of top-level properties by default with IR compiler

In Kotlin 1.4.30, we presented a prototype of lazy initialization of top-level properties in the JS IR compiler. By eliminating the need to initialize all properties when the application launches, lazy initialization reduces the startup time. Our measurements showed about a 10% speed-up on a real-life Kotlin/JS application.

Now, having polished and properly tested this mechanism, we're making lazy initialization the default for top-level properties in the IR compiler.

```
// lazy initialization
val a = run {
    val result = // intensive computations
        println(result)
    result
} // run is executed upon the first usage of the variable
```

If for some reason you need to initialize a property eagerly (upon the application start), mark it with the `@EagerInitialization` annotation.

Separate JS files for project modules by default with IR compiler

Previously, the JS IR compiler offered an ability to generate separate `.js` files for project modules. This was an alternative to the default option – a single `.js` file for the whole project. This file might be too large and inconvenient to use, because whenever you want to use a function from your project, you have to include the entire JS file as a dependency. Having multiple files adds flexibility and decreases the size of such dependencies. This feature was available with the `-Xir-per-module` compiler option.

Starting from 1.6.20, the JS IR compiler generates separate `.js` files for project modules by default.

Compiling the project into a single `.js` file is now available with the following Gradle property:

```
# gradle.properties
kotlin.js.ir.output.granularity=whole-program // `per-module` is the default
```

In previous releases, the experimental per-module mode (available via the `-Xir-per-module=true` flag) invoked `main()` functions in each module. This is inconsistent with the regular single `.js` mode. Starting with 1.6.20, the `main()` function will be invoked in the main module only in both cases. If you do need to run some code when a module is loaded, you can use top-level properties annotated with the `@EagerInitialization` annotation. See Lazy initialization of top-level properties by default (IR).

Char class optimization

The `Char` class is now handled by the Kotlin/JS compiler without introducing boxing (similar to inline classes). This speeds up operations on chars in Kotlin/JS code.

Aside from the performance improvement, this changes the way `Char` is exported to JavaScript: it's now translated to `Number`.

Improvements to export and TypeScript declaration generation

Kotlin 1.6.20 is bringing multiple fixes and improvements to the export mechanism (the `@JsExport` annotation), including the generation of TypeScript declarations (`.d.ts`) We've added the ability to export interfaces and enums, and we've fixed the export behavior in some corner cases that were reported to us previously. For more details, see the list of export improvements in YouTrack.

Learn more about using Kotlin code from JavaScript.

@AfterTest guarantees for asynchronous tests

Kotlin 1.6.20 makes `@AfterTest` functions work properly with asynchronous tests on Kotlin/JS. If a test function's return type is statically resolved to `Promise` , the compiler now schedules the execution of the `@AfterTest` function to the corresponding `then()` callback.

Security

Kotlin 1.6.20 introduces a couple of features to improve the security of your code:

- Using relative paths in klibs
- Persisting yarn.lock for Kotlin/JS Gradle projects
- Installation of npm dependencies with `--ignore-scripts` by default

Using relative paths in klibs

A library in `klib` format contains a serialized IR representation of source files, which also includes their paths for generating proper debug information. Before Kotlin 1.6.20, stored file paths were absolute. Since the library author may not want to share absolute paths, the 1.6.20 version comes with an alternative option.

If you are publishing a `klib` and want to use only relative paths of source files in the artifact, you can now pass the `-Xklib-relative-path-base` compiler option with one or multiple base paths of source files:

Kotlin

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinCompile::class).configureEach {
    // $base is a base path of source files
    kotlinOptions.freeCompilerArgs += "-Xklib-relative-path-base=$base"
}
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinCompile).configureEach {
    kotlinOptions {
        // $base is a base path of source files
        freeCompilerArgs += "-Xklib-relative-path-base=$base"
    }
}
```

Persisting yarn.lock for Kotlin/JS Gradle projects

The feature was backported to Kotlin 1.6.10.

The Kotlin/JS Gradle plugin now provides an ability to persist the `yarn.lock` file, making it possible to lock the versions of the npm dependencies for your project without additional Gradle configuration. The feature brings changes to the default project structure by adding the auto-generated `kotlin-js-store` directory to the project root. It holds the `yarn.lock` file inside.

We strongly recommend committing the `kotlin-js-store` directory and its contents to your version control system. Committing lockfiles to your version control system is a recommended practice because it ensures your application is being built with the exact same dependency tree on all machines, regardless of whether those are development environments on other machines or CI/CD services. Lockfiles also prevent your npm dependencies from being silently updated when a project is checked out on a new machine, which is a security concern.

Tools like Dependabot can also parse the `yarn.lock` files of your Kotlin/JS projects, and provide you with warnings if any npm package you depend on is compromised.

If needed, you can change both directory and lockfile names in the build script:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().lockFileDirectory
=
        project.rootDir.resolve("my-kotlin-js-store")
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().lockFileName =
"my-yarn.lock"
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).lockFile
=
        file("my-kotlin-js-store")

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).lockFile
= 'my-yarn.lock'
}
```

Changing the name of the lockfile may cause dependency inspection tools to no longer pick up the file.

Installation of npm dependencies with --ignore-scripts by default

The feature was backported to Kotlin 1.6.10.

The Kotlin/JS Gradle plugin now prevents the execution of lifecycle scripts during the installation of npm dependencies by default. The change is aimed at reducing the likelihood of executing malicious code from compromised npm packages.

To roll back to the old configuration, you can explicitly enable lifecycle scripts execution by adding the following lines to `build.gradle(.kts)`:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().ignoreScripts =
false
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).ignoreSc
= false
}
```

Learn more about npm dependencies of a Kotlin/JS Gradle project.

Gradle

Kotlin 1.6.20 brings the following changes for the Kotlin Gradle Plugin:

- New properties `kotlin.compiler.execution.strategy` and `compilerExecutionStrategy` for defining a Kotlin compiler execution strategy
- Deprecation of the options `kapt.use.worker.api`, `kotlin.experimental.coroutines`, and `kotlin.coroutines`
- Removal of the `kotlin.parallel.tasks.in.project` build option

Properties for defining Kotlin compiler execution strategy

Before Kotlin 1.6.20, you used the system property `-Dkotlin.compiler.execution.strategy` to define a Kotlin compiler execution strategy. This property might have been inconvenient in some cases. Kotlin 1.6.20 introduces a Gradle property with the same name, `kotlin.compiler.execution.strategy`, and the compile task property `compilerExecutionStrategy`.

The system property still works, but it will be removed in future releases.

The current priority of properties is the following:

- The task property `compilerExecutionStrategy` takes priority over the system property and the Gradle property `kotlin.compiler.execution.strategy`.
- The Gradle property takes priority over the system property.

There are three compiler execution strategies that you can assign to these properties:

Strategy	Where Kotlin compiler is executed	Incremental compilation	Other characteristics
Daemon	Inside its own daemon process	Yes	*The default strategy*. Can be shared between different Gradle daemons
In process	Inside the Gradle daemon process	No	May share the heap with the Gradle daemon
Out of process	In a separate process for each call	No	—

Accordingly, the available values for `kotlin.compiler.execution.strategy` properties (both system and Gradle's) are:

1. `daemon` (default)
2. `in-process`
3. `out-of-process`

Use the Gradle property `kotlin.compiler.execution.strategy` in `gradle.properties`:

```
# gradle.properties
kotlin.compiler.execution.strategy=out-of-process
```

The available values for the `compilerExecutionStrategy` task property are:

1. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.DAEMON` (default)
2. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.IN_PROCESS`
3. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.OUT_OF_PROCESS`

Use the task property `compilerExecutionStrategy` in the `build.gradle.kts` build script:

```
import org.jetbrains.kotlin.gradle.dsl.KotlinCompile
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy

// ...

tasks.withType<KotlinCompile>().configureEach {
    compilerExecutionStrategy.set(KotlinCompilerExecutionStrategy.IN_PROCESS)
}
```

Please leave your feedback in this YouTrack task.

Deprecation of build options for kapt and coroutines

In Kotlin 1.6.20, we changed deprecation levels of the properties:

- We deprecated the ability to run kapt via the Kotlin daemon with `kapt.use.worker.api` – now it produces a warning to Gradle's output. By default, kapt has been using Gradle workers since the 1.3.70 release, and we recommend sticking to this method.

  We are going to remove the option `kapt.use.worker.api` in future releases.

- We deprecated the `kotlin.experimental.coroutines` Gradle DSL option and the `kotlin.coroutines` property used in `gradle.properties`. Just use *suspending functions* or add the `kotlinx.coroutines` dependency to your `build.gradle(.kts)` file.

  Learn more about coroutines in the Coroutines guide.

Removal of the kotlin.parallel.tasks.in.project build option

In Kotlin 1.5.20, we announced the deprecation of the build option `kotlin.parallel.tasks.in.project`. This option has been removed in Kotlin 1.6.20.

Depending on the project, parallel compilation in the Kotlin daemon may require more memory. To reduce memory consumption, increase the heap size for the Kotlin daemon.

Learn more about the currently supported compiler options in the Kotlin Gradle plugin.

What's new in Kotlin 1.6.0

Released: 16 November 2021

Kotlin 1.6.0 introduces new language features, optimizations and improvements to existing features, and a lot of improvements to the Kotlin standard library.

You can also find an overview of the changes in the release blog post.

> For information about the Kotlin release cycle, see Kotlin release process.

Language

Kotlin 1.6.0 brings stabilization to several language features introduced for preview in the previous 1.5.30 release:

- Stable exhaustive when statements for enum, sealed and Boolean subjects
- Stable suspending functions as supertypes
- Stable suspend conversions
- Stable instantiation of annotation classes

It also includes various type inference improvements and support for annotations on class type parameters:

- Improved type inference for recursive generic types
- Changes to builder inference
- Support for annotations on class type parameters

Stable exhaustive when statements for enum, sealed, and Boolean subjects

An *exhaustive* `when` statement contains branches for all possible types or values of its subject, or for some types plus an `else` branch. It covers all possible cases, making your code safer.

We will soon prohibit non-exhaustive `when` statements to make the behavior consistent with `when` expressions. To ensure smooth migration, Kotlin 1.6.0 reports warnings about non-exhaustive `when` statements with an enum, sealed, or Boolean subject. These warnings will become errors in future releases.

```
sealed class Contact {
    data class PhoneCall(val number: String) : Contact()
    data class TextMessage(val number: String) : Contact()
}

fun Contact.messageCost(): Int =
    when(this) { // Error: 'when' expression must be exhaustive
        is Contact.PhoneCall -> 42
    }

fun sendMessage(contact: Contact, message: String) {
    // Starting with 1.6.0

    // Warning: Non exhaustive 'when' statements on Boolean will be
    // prohibited in 1.7, add 'false' branch or 'else' branch instead
    when(message.isEmpty()) {
        true -> return
    }
    // Warning: Non exhaustive 'when' statements on sealed class/interface will be
    // prohibited in 1.7, add 'is TextMessage' branch or 'else' branch instead
    when(contact) {
        is Contact.PhoneCall -> TODO()
    }
}
```

See this YouTrack ticket for a more detailed explanation of the change and its effects.

Stable suspending functions as supertypes

Implementation of suspending functional types has become Stable in Kotlin 1.6.0. A preview was available in 1.5.30.

The feature can be useful when designing APIs that use Kotlin coroutines and accept suspending functional types. You can now streamline your code by enclosing the desired behavior in a separate class that implements a suspending functional type.

```
class MyClickAction : suspend () -> Unit {
    override suspend fun invoke() { TODO() }
}

fun launchOnClick(action: suspend () -> Unit) {}
```

You can use an instance of this class where only lambdas and suspending function references were allowed previously: `launchOnClick(MyClickAction())`.

There are currently two limitations coming from implementation details:

- You can't mix ordinary functional types and suspending ones in the list of supertypes.
- You can't use multiple suspending functional supertypes.

Stable suspend conversions

Kotlin 1.6.0 introduces Stable conversions from regular to suspending functional types. Starting from 1.4.0, the feature supported functional literals and callable references. With 1.6.0, it works with any form of expression. As a call argument, you can now pass any expression of a suitable regular functional type where suspending is expected. The compiler will perform an implicit conversion automatically.

```
fun getSuspending(suspending: suspend () -> Unit) {}

fun suspending() {}

fun test(regular: () -> Unit) {
    getSuspending { }           // OK
    getSuspending(::suspending) // OK
    getSuspending(regular)      // OK
}
```

Stable instantiation of annotation classes

Kotlin 1.5.30 introduced experimental support for instantiation of annotation classes on the JVM platform. With 1.6.0, the feature is available by default both for Kotlin/JVM and Kotlin/JS.

Learn more about instantiation of annotation classes in this KEEP.

Improved type inference for recursive generic types

Kotlin 1.5.30 introduced an improvement to type inference for recursive generic types, which allowed their type arguments to be inferred based only on the upper bounds of the corresponding type parameters. The improvement was available with the compiler option. In version 1.6.0 and later, it is enabled by default.

```
// Before 1.5.30
val containerA = PostgreSQLContainer<Nothing>(DockerImageName.parse("postgres:13-alpine")).apply {
  withDatabaseName("db")
  withUsername("user")
  withPassword("password")
  withInitScript("sql/schema.sql")
}

// With compiler option in 1.5.30 or by default starting with 1.6.0
val containerB = PostgreSQLContainer(DockerImageName.parse("postgres:13-alpine"))
  .withDatabaseName("db")
  .withUsername("user")
  .withPassword("password")
  .withInitScript("sql/schema.sql")
```

Changes to builder inference

Builder inference is a type inference flavor which is useful when calling generic builder functions. It can infer the type arguments of a call with the help of type information from calls inside its lambda argument.

We're making multiple changes that are bringing us closer to fully stable builder inference. Starting with 1.6.0:

- You can make calls returning an instance of a not yet inferred type inside a builder lambda without specifying the `-Xunrestricted-builder-inference` compiler option introduced in 1.5.30.
- With `-Xenable-builder-inference`, you can write your own builders without applying the `@BuilderInference` annotation.

  > Note that clients of these builders will need to specify the same `-Xenable-builder-inference` compiler option.

- With the `-Xenable-builder-inference`, builder inference automatically activates if a regular type inference cannot get enough information about a type.

Learn how to write custom generic builders.

Support for annotations on class type parameters

Support for annotations on class type parameters looks like this:

```
@Target(AnnotationTarget.TYPE_PARAMETER)
annotation class BoxContent

class Box<@BoxContent T> {}
```

Annotations on all type parameters are emitted into JVM bytecode so annotation processors are able to use them.

For the motivating use case, read this YouTrack ticket.

Learn more about annotations.

Supporting previous API versions for a longer period

Starting with Kotlin 1.6.0, we will support development for three previous API versions instead of two, along with the current stable one. Currently, we support versions 1.3, 1.4, 1.5, and 1.6.

Kotlin/JVM

For Kotlin/JVM, starting with 1.6.0, the compiler can generate classes with a bytecode version corresponding to JVM 17. The new language version also includes optimized delegated properties and repeatable annotations, which we had on the roadmap:

- Repeatable annotations with runtime retention for 1.8 JVM target
- Optimize delegated properties which call get/set on the given KProperty instance

Repeatable annotations with runtime retention for 1.8 JVM target

Java 8 introduced repeatable annotations, which can be applied multiple times to a single code element. The feature requires two declarations to be present in the Java code: the repeatable annotation itself marked with `@java.lang.annotation.Repeatable` and the containing annotation to hold its values.

Kotlin also has repeatable annotations, but requires only `@kotlin.annotation.Repeatable` to be present on an annotation declaration to make it repeatable. Before 1.6.0, the feature supported only `SOURCE` retention and was incompatible with Java's repeatable annotations. Kotlin 1.6.0 removes these limitations. `@kotlin.annotation.Repeatable` now accepts any retention and makes the annotation repeatable both in Kotlin and Java. Java's repeatable annotations are now also supported from the Kotlin side.

While you can declare a containing annotation, it's not necessary. For example:

- If an annotation `@Tag` is marked with `@kotlin.annotation.Repeatable`, the Kotlin compiler automatically generates a containing annotation class under the name of `@Tag.Container`:

```
@Repeatable
annotation class Tag(val name: String)

// The compiler generates @Tag.Container containing annotation
```

- To set a custom name for a containing annotation, apply the `@kotlin.jvm.JvmRepeatable` meta-annotation and pass the explicitly declared containing annotation class as an argument:

```
@JvmRepeatable(Tags::class)
annotation class Tag(val name: String)

annotation class Tags(val value: Array<Tag>)
```

Kotlin reflection now supports both Kotlin's and Java's repeatable annotations via a new function, `KAnnotatedElement.findAnnotations()`.

Learn more about Kotlin repeatable annotations in this KEEP.

Optimize delegated properties which call get/set on the given KProperty instance

We optimized the generated JVM bytecode by omitting the `$delegate` field and generating immediate access to the referenced property.

For example, in the following code

```
class Box<T> {
    private var impl: T = ...

    var content: T by ::impl
}
```

Kotlin no longer generates the field `content$delegate`. Property accessors of the `content` variable invoke the `impl` variable directly, skipping the delegated property's `getValue` / `setValue` operators and thus avoiding the need for the property reference object of the `KProperty` type.

Thanks to our Google colleagues for the implementation!

Learn more about delegated properties.

Kotlin/Native

Kotlin/Native is receiving multiple improvements and component updates, some of them in the preview state:

- Preview of the new memory manager
- Support for Xcode 13
- Compilation of Windows targets on any host
- LLVM and linker updates
- Performance improvements
- Unified compiler plugin ABI with JVM and JS IR backends
- Detailed error messages for klib linkage failures
- Reworked unhandled exception handling API

Preview of the new memory manager

> The new Kotlin/Native memory manager is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

With Kotlin 1.6.0, you can try the development preview of the new Kotlin/Native memory manager. It moves us closer to eliminating the differences between the JVM and Native platforms to provide a consistent developer experience in multiplatform projects.

One of the notable changes is the lazy initialization of top-level properties, like in Kotlin/JVM. A top-level property gets initialized when a top-level property or function from the same file is accessed for the first time. This mode also includes global interprocedural optimization (enabled only for release binaries), which removes redundant initialization checks.

We've recently published a blog post about the new memory manager. Read it to learn about the current state of the new memory manager and find some demo projects, or jump right to the migration instructions to try it yourself. Please check how the new memory manager works on your projects and share feedback in our issue tracker, YouTrack.

Support for Xcode 13

Kotlin/Native 1.6.0 supports Xcode 13 – the latest version of Xcode. Feel free to update your Xcode and continue working on your Kotlin projects for Apple operating systems.

> New libraries added in Xcode 13 aren't available for use in Kotlin 1.6.0, but we're going to add support for them in upcoming versions.

Compilation of Windows targets on any host

Starting from 1.6.0, you don't need a Windows host to compile the Windows targets `mingwX64` and `mingwX86`. They can be compiled on any host that supports Kotlin/Native.

LLVM and linker updates

We've reworked the LLVM dependency that Kotlin/Native uses under the hood. This brings various benefits, including:

- Updated LLVM version to 11.1.0.
- Decreased dependency size. For example, on macOS it's now about 300 MB instead of 1200 MB in the previous version.
- Excluded dependency on the `ncurses5` library that isn't available in modern Linux distributions.

In addition to the LLVM update, Kotlin/Native now uses the LLD linker (a linker from the LLVM project) for MingGW targets. It provides various benefits over the previously used ld.bfd linker, and will allow us to improve runtime performance of produced binaries and support compiler caches for MinGW targets. Note that LLD requires import libraries for DLL linkage. Learn more in this Stack Overflow thread.

Performance improvements

Kotlin/Native 1.6.0 delivers the following performance improvements:

- Compilation time: compiler caches are enabled by default for `linuxX64` and `iosArm64` targets. This speeds up most compilations in debug mode (except the first one). Measurements showed about a 200% speed increase on our test projects. The compiler caches have been available for these targets since Kotlin 1.5.0 with additional Gradle properties; you can remove them now.
- Runtime: iterating over arrays with `for` loops is now up to 12% faster thanks to optimizations in the produced LLVM code.

Unified compiler plugin ABI with JVM and JS IR backends

> The option to use the common IR compiler plugin ABI for Kotlin/Native is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

In previous versions, authors of compiler plugins had to provide separate artifacts for Kotlin/Native because of the differences in the ABI.

Starting from 1.6.0, the Kotlin Multiplatform Gradle plugin is able to use the embeddable compiler jar – the one used for the JVM and JS IR backends – for Kotlin/Native. This is a step toward unification of the compiler plugin

development experience, as you can now use the same compiler plugin artifacts for Native and other supported platforms.

This is a preview version of such support, and it requires an opt-in. To start using generic compiler plugin artifacts for Kotlin/Native, add the following line to `gradle.properties`: `kotlin.native.useEmbeddableCompilerJar=true`.

We're planning to use the embeddable compiler jar for Kotlin/Native by default in the future, so it's vital for us to hear how the preview works for you.

If you are an author of a compiler plugin, please try this mode and check if it works for your plugin. Note that depending on your plugin's structure, migration steps may be required. See this YouTrack issue for migration instructions and leave your feedback in the comments.

Detailed error messages for klib linkage failures

The Kotlin/Native compiler now provides detailed error messages for klib linkage errors. The messages now have clear error descriptions, and they also include information about possible causes and ways to fix them.

For example:

- 1.5.30:

```
e: java.lang.IllegalStateException: IrTypeAliasSymbol expected: Unbound public symbol for public
kotlinx.coroutines/CancellationException|null[0]
<stack trace>
```

- 1.6.0:

```
e: The symbol of unexpected type encountered during IR deserialization: IrClassPublicSymbolImpl,
kotlinx.coroutines/CancellationException|null[0].
IrTypeAliasSymbol is expected.

This could happen if there are two libraries, where one library was compiled against the different
version of the other library than the one currently used in the project.
Please check that the project configuration is correct and has consistent versions of dependencies.

The list of libraries that depend on "org.jetbrains.kotlinx:kotlinx-coroutines-core
(org.jetbrains.kotlinx:kotlinx-coroutines-core-macosx64)" and may lead to conflicts:
<list of libraries and potential version mismatches>

Project dependencies:
<dependencies tree>
```

Reworked unhandled exception handling API

We've unified the processing of unhandled exceptions throughout the Kotlin/Native runtime and exposed the default processing as the function `processUnhandledException(throwable: Throwable)` for use by custom execution environments, like `kotlinx.coroutines`. This processing is also applied to exceptions that escape operation in `Worker.executeAfter()`, but only for the new memory manager.

API improvements also affected the hooks that have been set by `setUnhandledExceptionHook()`. Previously such hooks were reset after the Kotlin/Native runtime called the hook with an unhandled exception, and the program would always terminate right after. Now these hooks may be used more than once, and if you want the program to always terminate on an unhandled exception, either do not set an unhandled exception hook (`setUnhandledExceptionHook()`), or make sure to call `terminateWithUnhandledException()` at the end of your hook. This will help you send exceptions to a third-party crash reporting service (like Firebase Crashlytics) and

then terminate the program. Exceptions that escape `main()` and exceptions that cross the interop boundary will always terminate the program, even if the hook did not call `terminateWithUnhandledException()`.

Kotlin/JS

We're continuing to work on stabilizing the IR backend for the Kotlin/JS compiler. Kotlin/JS now has an option to disable downloading of Node.js and Yarn.

Option to use pre-installed Node.js and Yarn

You can now disable downloading Node.js and Yarn when building Kotlin/JS projects and use the instances already installed on the host. This is useful for building on servers without internet connectivity, such as CI servers.

To disable downloading external components, add the following lines to your `build.gradle(.kts)`:

- Yarn:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().download =
false // or true for default behavior
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).downl
= false
}
```

- Node.js:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExtension>().download =
false // or true for default behavior
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExtension).d
= false
}
```

Kotlin Gradle plugin

In Kotlin 1.6.0, we changed the deprecation level of the `KotlinGradleSubplugin` class to 'ERROR'. This class was used for writing compiler plugins. In the following releases, we'll remove this class. Use the class `KotlinCompilerPluginSupportPlugin` instead.

We removed the `kotlin.useFallbackCompilerSearch` build option and the `noReflect` and `includeRuntime` compiler options. The `useIR` compiler option has been hidden and will be removed in upcoming releases.

Learn more about the currently supported compiler options in the Kotlin Gradle plugin.

Standard library

The new 1.6.0 version of the standard library stabilizes experimental features, introduces new ones, and unifies its behavior across the platforms:

- New readline functions
- Stable typeOf()
- Stable collection builders
- Stable Duration API
- Splitting Regex into a sequence
- Bit rotation operations on integers
- Changes for replace() and replaceFirst() in JS
- Improvements to the existing API
- Deprecations

New readline functions

Kotlin 1.6.0 offers new functions for handling standard input: `readln()` and `readlnOrNull()`.

For now, new functions are available for the JVM and Native target platforms only.

Earlier versions	1.6.0 alternative	Usage
`readLine()!!`	`readln()`	Reads a line from stdin and returns it, or throws a `RuntimeException` if EOF has been reached.
`readLine()`	`readlnOrNull()`	Reads a line from stdin and returns it, or returns `null` if EOF has been reached.

We believe that eliminating the need to use `!!` when reading a line will improve the experience for newcomers and simplify teaching Kotlin. To make the read-line operation name consistent with its `println()` counterpart, we've decided to shorten the names of new functions to 'ln'.

```
println("What is your nickname?")
val nickname = readln()
println("Hello, $nickname!")
```

```
fun main() {
    var sum = 0
    while (true) {
        val nextLine = readlnOrNull().takeUnless {
            it.isNullOrEmpty()
        } ?: break
        sum += nextLine.toInt()
    }
    println(sum)
}
```

The existing `readLine()` function will get a lower priority than `readln()` and `readlnOrNull()` in your IDE code completion. IDE inspections will also recommend using new functions instead of the legacy `readLine()`.

We're planning to gradually deprecate the `readLine()` function in future releases.

Stable typeOf()

Version 1.6.0 brings a Stable `typeOf()` function, closing one of the major roadmap items.

Since 1.3.40, `typeOf()` was available on the JVM platform as an experimental API. Now you can use it in any Kotlin platform and get `KType` representation of any Kotlin type that the compiler can infer:

```
inline fun <reified T> renderType(): String {
    val type = typeOf<T>()
    return type.toString()
}

fun main() {
    val fromExplicitType = typeOf<Int>()
    val fromReifiedType = renderType<List<Int>>()
}
```

Stable collection builders

In Kotlin 1.6.0, collection builder functions have been promoted to Stable. Collections returned by collection builders are now serializable in their read-only state.

You can now use `buildMap()`, `buildList()`, and `buildSet()` without the opt-in annotation:

```
fun main() {
    val x = listOf('b', 'c')
    val y = buildList {
        add('a')
        addAll(x)
        add('d')
    }
    println(y)  // [a, b, c, d]
}
```

Stable Duration API

The Duration class for representing duration amounts in different time units has been promoted to Stable. In 1.6.0, the Duration API has received the following changes:

- The first component of the `toComponents()` function that decomposes the duration into days, hours, minutes, seconds, and nanoseconds now has the `Long` type instead of `Int`. Before, if the value didn't fit into the `Int` range, it was coerced into that range. With the `Long` type, you can decompose any value in the duration range without cutting off the values that don't fit into `Int`.
- The `DurationUnit` enum is now standalone and not a type alias of `java.util.concurrent.TimeUnit` on the JVM. We haven't found any convincing cases in which having `typealias DurationUnit = TimeUnit` could be useful. Also, exposing the `TimeUnit` API through a type alias might confuse `DurationUnit` users.
- In response to community feedback, we're bringing back extension properties like `Int.seconds`. But we'd like to limit their applicability, so we put them into the companion of the `Duration` class. While the IDE can still propose extensions in completion and automatically insert an import from the companion, in the future we plan to limit this behavior to cases when the `Duration` type is expected.

  ```
  import kotlin.time.Duration.Companion.seconds

  fun main() {
      val duration = 10000
      println("There are ${duration.seconds.inWholeMinutes} minutes in $duration seconds")
      // There are 166 minutes in 10000 seconds
  }
  ```

  We suggest replacing previously introduced companion functions, such as `Duration.seconds(Int)`, and deprecated top-level extensions like `Int.seconds` with new extensions in `Duration.Companion`.

  > Such a replacement may cause ambiguity between old top-level extensions and new companion extensions. Be sure to use the wildcard import of the kotlin.time package – `import kotlin.time.*` – before doing automated migration.

Splitting Regex into a sequence

The `Regex.splitToSequence(CharSequence)` and `CharSequence.splitToSequence(Regex)` functions are promoted to Stable. They split the string around matches of the given regex, but return the result as a Sequence so that all operations on this result are executed lazily:

```
fun main() {
    val colorsText = "green, red, brown&blue, orange, pink&green"
    val regex = "[,\\s]+".toRegex()
    val mixedColor = regex.splitToSequence(colorsText)
    // or
    // val mixedColor = colorsText.splitToSequence(regex)
        .onEach { println(it) }
        .firstOrNull { it.contains('&') }
    println(mixedColor) // "brown&blue"
}
```

Bit rotation operations on integers

In Kotlin 1.6.0, the `rotateLeft()` and `rotateRight()` functions for bit manipulations became Stable. The functions rotate the binary representation of the number left or right by a specified number of bits:

```
fun main() {
    val number: Short = 0b10001
    println(number
        .rotateRight(2)
        .toString(radix = 2)) // 100000000000100
    println(number
        .rotateLeft(2)
        .toString(radix = 2))  // 1000100
}
```

Changes for replace() and replaceFirst() in JS

Before Kotlin 1.6.0, the `replace()` and `replaceFirst()` Regex functions behaved differently in Java and JS when the replacement string contained a group reference. To make the behavior consistent across all target platforms, we've changed their implementation in JS.

Occurrences of `${name}` or `$index` in the replacement string are substituted with the subsequences corresponding to the captured groups with the specified index or a name:

- `$index` – the first digit after '$' is always treated as a part of the group reference. Subsequent digits are incorporated into the `index` only if they form a valid group reference.Only digits '0'–'9' are considered potential components of the group reference. Note that indexes of captured groups start from '1'. The group with index '0' stands for the whole match.
- `${name}` – the `name` can consist of Latin letters 'a'–'z', 'A'–'Z', or digits '0'–'9'. The first character must be a letter.

  > Named groups in replacement patterns are currently supported only on the JVM.

- To include the succeeding character as a literal in the replacement string, use the backslash character `\` :

  ```
  fun main() {
      println(Regex("(.+)").replace("Kotlin", """\$ $1""")) // $ Kotlin
      println(Regex("(.+)").replaceFirst("1.6.0", """\\ $1""")) // \ 1.6.0
  }
  ```

  You can use `Regex.escapeReplacement()` if the replacement string has to be treated as a literal string.

Improvements to the existing API

- Version 1.6.0 added the infix extension function for `Comparable.compareTo()`. You can now use the infix form for comparing two objects for order:

  ```
  class WrappedText(val text: String) : Comparable<WrappedText> {
       override fun compareTo(other: WrappedText): Int =
           this.text compareTo other.text
  }
  ```

- `Regex.replace()` in JS is now also not inline to unify its implementation across all platforms.
- The `compareTo()` and `equals()` String functions, as well as the `isBlank()` CharSequence function now behave in JS exactly the same way they do on the JVM. Previously there were deviations when it came to non-ASCII characters.

Deprecations

In Kotlin 1.6.0, we're starting the deprecation cycle with a warning for some JS-only stdlib API.

concat(), match(), and matches() string functions

- To concatenate the string with the string representation of a given other object, use `plus()` instead of `concat()`.
- To find all occurrences of a regular expression within the input, use `findAll()` of the Regex class instead of `String.match(regex: String)`.
- To check if the regular expression matches the entire input, use `matches()` of the Regex class instead of `String.matches(regex: String)`.

sort() on arrays taking comparison functions

We've deprecated the `Array<out T>.sort()` function and the inline functions `ByteArray.sort()`, `ShortArray.sort()`, `IntArray.sort()`, `LongArray.sort()`, `FloatArray.sort()`, `DoubleArray.sort()`, and `CharArray.sort()`, which sorted arrays following the order passed by the comparison function. Use other standard library functions for array sorting.

See the collection ordering section for reference.

Tools

Kover – a code coverage tool for Kotlin

The Kover Gradle plugin is Experimental. We would appreciate your feedback on it in GitHub.

With Kotlin 1.6.0, we're introducing Kover – a Gradle plugin for the IntelliJ and JaCoCo Kotlin code coverage agents. It works with all language constructs, including inline functions.

Learn more about Kover on its GitHub repository or in this video:

Watch video online.

Coroutines 1.6.0-RC

`kotlinx.coroutines` 1.6.0-RC is out with multiple features and improvements:

- Support for the new Kotlin/Native memory manager
- Introduction of dispatcher *views* API, which allows limiting parallelism without creating additional threads
- Migrating from Java 6 to Java 8 target
- `kotlinx-coroutines-test` with the new reworked API and multiplatform support
- Introduction of `CopyableThreadContextElement`, which gives coroutines a thread-safe write access to `ThreadLocal` variables

Learn more in the changelog.

Migrating to Kotlin 1.6.0

IntelliJ IDEA and Android Studio will suggest updating the Kotlin plugin to 1.6.0 once it is available.

To migrate existing projects to Kotlin 1.6.0, change the Kotlin version to `1.6.0` and reimport your Gradle or Maven project. Learn how to update to Kotlin 1.6.0.

To start a new project with Kotlin 1.6.0, update the Kotlin plugin and run the Project Wizard from **File** | **New** | **Project**.

The new command-line compiler is available for download on the GitHub release page.

Kotlin 1.6.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of such changes in the Compatibility Guide for Kotlin 1.6.

Compatibility guide for Kotlin 1.6.x

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.5 to Kotlin 1.6.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Make when statements with enum, sealed, and Boolean subjects exhaustive by default

Issue: KT-47709

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will warn about the `when` statement with an enum, sealed, or Boolean subject being non-exhaustive

Deprecation cycle:

- 1.6.0: introduce a warning when the `when` statement with an enum, sealed, or Boolean subject is non-exhaustive (error in the progressive mode)
- 1.7.0: raise this warning to an error

Deprecate confusing grammar in when-with-subject

Issue: KT-48385

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will deprecate several confusing grammar constructs in `when` condition expressions

Deprecation cycle:

- 1.6.20: introduce a deprecation warning on the affected expressions
- 1.8.0: raise this warning to an error
- >= 1.8: repurpose some deprecated constructs for new language features

Prohibit access to class members in the super constructor call of its companion and nested objects

Issue: KT-25289

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will report an error for arguments of super constructor call of companion and regular objects if the receiver of such arguments refers to the containing declaration

Deprecation cycle:

- 1.5.20: introduce a warning on the problematic arguments
- 1.6.0: raise this warning to an error, `-XXLanguage:-ProhibitSelfCallsInNestedObjects` can be used to temporarily revert to the pre-1.6 behavior

Type nullability enhancement improvements

Issue: KT-48623

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.7 will change how it loads and interprets type nullability annotations in Java code

Deprecation cycle:

- 1.4.30: introduce warnings for cases where more precise type nullability could lead to an error
- 1.7.0: infer more precise nullability of Java types, `-XXLanguage:-TypeEnhancementImprovementsInStrictMode` can be used to temporarily revert to the pre-1.7 behavior

Prevent implicit coercions between different numeric types

Issue: KT-48645

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin will avoid converting numeric values automatically to a primitive numeric type where only a downcast to that type was needed semantically

Deprecation cycle:

- < 1.5.30: the old behavior in all affected cases
- 1.5.30: fix the downcast behavior in generated property delegate accessors, `-Xuse-old-backend` can be used to temporarily revert to the pre-1.5.30 fix behavior
- >= 1.6.20: fix the downcast behavior in other affected cases

Prohibit declarations of repeatable annotation classes whose container annotation violates JLS

Issue: KT-47928

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6 will check that the container annotation of a repeatable annotation satisfies the same requirements as in JLS 9.6.3: array-typed value method, retention, and target

Deprecation cycle:

- 1.5.30: introduce a warning on repeatable container annotation declarations violating JLS requirements (error in the progressive mode)
- 1.6.0: raise this warning to an error, `-XXLanguage:-RepeatableAnnotationContainerConstraints` can be used to temporarily disable the error reporting

Prohibit declaring a nested class named Container in a repeatable annotation class

Issue: KT-47971

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6 will check that a repeatable annotation declared in Kotlin doesn't have a nested class with the predefined name `Container`

Deprecation cycle:

- 1.5.30: introduce a warning on nested classes with the name `Container` in a Kotlin-repeatable annotation class (error in the progressive mode)
- 1.6.0: raise this warning to an error, `-XXLanguage:-RepeatableAnnotationContainerConstraints` can be used to temporarily disable the error reporting

Prohibit @JvmField on a property in the primary constructor that overrides an interface property

Issue: KT-32753

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6 will outlaw annotating a property declared in the primary constructor that overrides an interface property with the `@JvmField` annotation

Deprecation cycle:

- 1.5.20: introduce a warning on the `@JvmField` annotation on such properties in the primary constructor
- 1.6.0: raise this warning to an error, `-XXLanguage:-ProhibitJvmFieldOnOverrideFromInterfaceInPrimaryConstructor` can be used to temporarily disable the error reporting

Deprecate the enable and the compatibility modes of the compiler option -Xjvm-default

Issue: KT-46329

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6.20 will warn about the usage of `enable` and `compatibility` modes of the `-Xjvm-default` compiler option

Deprecation cycle:

- 1.6.20: introduce a warning on the `enable` and `compatibility` modes of the `-Xjvm-default` compiler option
- >= 1.8.0: raise this warning to an error

Prohibit super calls from public-abi inline functions

Issue: KT-45379

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will outlaw calling functions with a `super` qualifier from public or protected inline functions and properties

Deprecation cycle:

- 1.5.0: introduce a warning on super calls from public or protected inline functions or property accessors
- 1.6.0: raise this warning to an error, `-XXLanguage:-ProhibitSuperCallsFromPublicInline` can be used to temporarily disable the error reporting

Prohibit protected constructor calls from public inline functions

Issue: KT-48860

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will outlaw calling protected constructors from public or protected inline functions and properties

Deprecation cycle:

- 1.4.30: introduce a warning on protected constructor calls from public or protected inline functions or property accessors
- 1.6.0: raise this warning to an error, `-XXLanguage:-ProhibitProtectedConstructorCallFromPublicInline` can be used to temporarily disable the error reporting

Prohibit exposing private nested types from private-in-file types

Issue: KT-20094

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will outlaw exposing private nested types and inner classes from private-in-file types

Deprecation cycle:

- 1.5.0: introduce a warning on private types exposed from private-in-file types
- 1.6.0: raise this warning to an error, `-XXLanguage:-PrivateInFileEffectiveVisibility` can be used to temporarily disable the error reporting

Annotation target is not analyzed in several cases for annotations on a type

Issue: KT-28449

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will no longer allow annotations on types that should not be applicable to types

Deprecation cycle:

- 1.5.20: introduce an error in the progressive mode
- 1.6.0: introduce an error, `-XXLanguage:-ProperCheckAnnotationsTargetInTypeUsePositions` can be used to temporarily disable the error reporting

Prohibit calls to functions named suspend with a trailing lambda

Issue: KT-22562

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.6 will no longer allow calling functions named `suspend` that have the single argument of a functional type passed as a trailing lambda

Deprecation cycle:

- 1.3.0: introduce a warning on such function calls
- 1.6.0: raise this warning to an error
- >= 1.7.0: introduce changes to the language grammar, so that `suspend` before `{` is parsed as a keyword

Standard library

Remove brittle contains optimization in minus/removeAll/retainAll

Issue: KT-45438

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: Kotlin 1.6 will no longer perform conversion to set for the argument of functions and operators that remove several elements from collection/iterable/array/sequence.

Deprecation cycle:

- < 1.6: the old behavior: the argument is converted to set in some cases
- 1.6.0: if the function argument is a collection, it's no longer converted to `Set`. If it's not a collection, it can be converted to `List` instead.
  The old behavior can be temporarily turned back on JVM by setting the system property `kotlin.collections.convert_arg_to_set_in_removeAll=true`
- >= 1.7: the system property above will no longer have an effect

Change value generation algorithm in Random.nextLong

Issue: KT-47304

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: Kotlin 1.6 changes the value generation algorithm in the `Random.nextLong` function to avoid producing values out of the specified range.

Deprecation cycle:

- 1.6.0: the behavior is fixed immediately

Gradually change the return type of collection min and max functions to non-nullable

Issue: KT-38854

Component: kotlin-stdlib

Incompatible change type: source

Short summary: the return type of collection `min` and `max` functions will be changed to non-nullable in Kotlin 1.7

Deprecation cycle:

- 1.4.0: introduce `...OrNull` functions as synonyms and deprecate the affected API (see details in the issue)
- 1.5.0: raise the deprecation level of the affected API to an error
- 1.6.0: hide the deprecated functions from the public API
- >= 1.7: reintroduce the affected API but with non-nullable return type

Deprecate floating-point array functions: contains, indexOf, lastIndexOf

Issue: KT-28753

Component: kotlin-stdlib

Incompatible change type: source

Short summary: Kotlin deprecates floating-point array functions `contains`, `indexOf`, `lastIndexOf` that compare values using the IEEE-754 order instead of the total order

Deprecation cycle:

- 1.4.0: deprecate the affected functions with a warning
- 1.6.0: raise the deprecation level to an error
- >= 1.7: hide the deprecated functions from the public API

Migrate declarations from kotlin.dom and kotlin.browser packages to kotlinx.*

Issue: KT-39330

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: declarations from the `kotlin.dom` and `kotlin.browser` packages are moved to the corresponding `kotlinx.*` packages to prepare for extracting them from stdlib

Deprecation cycle:

- 1.4.0: introduce the replacement API in `kotlinx.dom` and `kotlinx.browser` packages
- 1.4.0: deprecate the API in `kotlin.dom` and `kotlin.browser` packages and propose the new API above as a replacement
- 1.6.0: raise the deprecation level to an error
- >= 1.7: remove the deprecated functions from stdlib
- >= 1.7: move the API in kotlinx.* packages to a separate library

Make Regex.replace function not inline in Kotlin/JS

Issue: KT-27738

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: the `Regex.replace` function with the functional `transform` parameter will no longer be inline in Kotlin/JS

Deprecation cycle:

- 1.6.0: remove the `inline` modifier from the affected function

Different behavior of the Regex.replace function in JVM and JS when replacement string contains group reference

Issue: KT-28378

Component: kotlin-stdlib (JS)

Incompatible change type: behavioral

Short summary: the function `Regex.replace` in Kotlin/JS with the replacement pattern string will follow the same syntax of that pattern as in Kotlin/JVM

Deprecation cycle:

- 1.6.0: change the replacement pattern handling in `Regex.replace` of the Kotlin/JS stdlib

Use the Unicode case folding in JS Regex

Issue: KT-45928

Component: kotlin-stdlib (JS)

Incompatible change type: behavioral

Short summary: the `Regex` class in Kotlin/JS will use `unicode` flag when calling the underlying JS Regular expressions engine to search and compare characters according to the Unicode rules. This brings certain version requirements of the JS environment and causes more strict validation of unnecessary escaping in the regex pattern string.

Deprecation cycle:

- 1.5.0: enable the Unicode case folding in most functions of the JS `Regex` class
- 1.6.0: enable the Unicode case folding in the `Regex.replaceFirst` function

Deprecate some JS-only API

Issue: KT-48587

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: a number of JS-only functions in stdlib are deprecated for removal. They include: `String.concat(String)`, `String.match(regex: String)`, `String.matches(regex: String)`, and the `sort` functions on arrays taking a comparison function, for example, `Array<out T>.sort(comparison: (a: T, b: T) -> Int)`

Deprecation cycle:

- 1.6.0: deprecate the affected functions with a warning
- 1.7.0: raise the deprecation level to an error
- 1.8.0: remove the deprecated functions from the public API

Hide implementation- and interop-specific functions from the public API of classes in Kotlin/JS

Issue: KT-48587

Component: kotlin-stdlib (JS)

Incompatible change type: source, binary

Short summary: the functions `HashMap.createEntrySet` and `AbstactMutableCollection.toJSON` change their visibility to internal

Deprecation cycle:

- 1.6.0: make the functions internal, thus removing them from the public API

Tools

Deprecate KotlinGradleSubplugin class

Issue: KT-48830

Component: Gradle

Incompatible change type: source

Short summary: the class `KotlinGradleSubplugin` will be deprecated in favor of `KotlinCompilerPluginSupportPlugin`

Deprecation cycle:

- 1.6.0: raise the deprecation level to an error
- >= 1.7.0: remove the deprecated class

Remove kotlin.useFallbackCompilerSearch build option

Issue: KT-46719

Component: Gradle

Incompatible change type: source

Short summary: remove the deprecated 'kotlin.useFallbackCompilerSearch' build option

Deprecation cycle:

- 1.5.0: raise the deprecation level to a warning
- 1.6.0: remove the deprecated option

Remove several compiler options

Issue: KT-48847

Component: Gradle

Incompatible change type: source

Short summary: remove the deprecated `noReflect` and `includeRuntime` compiler options

Deprecation cycle:

- 1.5.0: raise the deprecation level to an error
- 1.6.0: remove the deprecated options

Deprecate useIR compiler option

Issue: KT-48847

Component: Gradle

Incompatible change type: source

Short summary: hide the deprecated `useIR` compiler option

Deprecation cycle:

- 1.5.0: raise the deprecation level to a warning
- 1.6.0: hide the option
- >= 1.7.0: remove the deprecated option

Deprecate kapt.use.worker.api Gradle property

Issue: KT-48826

Component: Gradle

Incompatible change type: source

Short summary: deprecate the `kapt.use.worker.api` property that allowed to run kapt via Gradle Workers API (default: true)

Deprecation cycle:

- 1.6.20: raise the deprecation level to a warning
- >= 1.8.0: remove this property

Remove kotlin.parallel.tasks.in.project Gradle property

Issue: KT-46406

Component: Gradle

Incompatible change type: source

Short summary: remove the `kotlin.parallel.tasks.in.project` property

Deprecation cycle:

- 1.5.20: raise the deprecation level to a warning
- 1.6.20: remove this property

Deprecate kotlin.experimental.coroutines Gradle DSL option and kotlin.coroutines Gradle property

Issue: KT-50369

Component: Gradle

Incompatible change type: source

Short summary: deprecate the `kotlin.experimental.coroutines` Gradle DSL option and the `kotlin.coroutines` property

Deprecation cycle:

- 1.6.20: raise the deprecation level to a warning
- >= 1.7.0: remove the DSL option and the property

What's new in Kotlin 1.5.30

Released: 24 August 2021

Kotlin 1.5.30 offers language updates including previews of future changes, various improvements in platform support and tooling, and new standard library functions.

Here are some major improvements:

- Language features, including experimental sealed `when` statements, changes in using opt-in requirement, and others
- Native support for Apple silicon
- Kotlin/JS IR backend reaches Beta
- Improved Gradle plugin experience

You can also find a short overview of the changes in the release blog post and this video:

Watch video online.

For information about the Kotlin release cycle, see Kotlin release process.

Language features

Kotlin 1.5.30 is presenting previews of future language changes and bringing improvements to the opt-in requirement mechanism and type inference:

- Exhaustive when statements for sealed and Boolean subjects
- Suspending functions as supertypes
- Requiring opt-in on implicit usages of experimental APIs
- Changes to using opt-in requirement annotations with different targets
- Improvements to type inference for recursive generic types

- Eliminating builder inference restrictions

Exhaustive when statements for sealed and Boolean subjects

> Support for sealed (exhaustive) when statements is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

An *exhaustive* `when` statement contains branches for either all possible types or values of its subject, or for certain types and includes an `else` branch to cover any remaining cases.

We're planning to prohibit non-exhaustive `when` statements soon to make the behavior consistent with `when` expressions. To ensure smooth migration, you can configure the compiler to report warnings about non-exhaustive `when` statements with a sealed class or a Boolean. Such warnings will appear by default in Kotlin 1.6 and will become errors later.

> Enums already get a warning.

```
sealed class Mode {
    object ON : Mode()
    object OFF : Mode()
}

fun main() {
    val x: Mode = Mode.ON
    when (x) {
        Mode.ON -> println("ON")
    }
// WARNING: Non exhaustive 'when' statements on sealed classes/interfaces
// will be prohibited in 1.7, add an 'OFF' or 'else' branch instead

    val y: Boolean = true
    when (y) {
        true -> println("true")
    }
// WARNING: Non exhaustive 'when' statements on Booleans will be prohibited
// in 1.7, add a 'false' or 'else' branch instead
}
```

To enable this feature in Kotlin 1.5.30, use language version `1.6`. You can also change the warnings to errors by enabling progressive mode.

Kotlin

```
kotlin {
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.6"
            //progressiveMode = true // false by default
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.6'
            //progressiveMode = true // false by default
        }
    }
}
```

Suspending functions as supertypes

Support for suspending functions as supertypes is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 1.5.30 provides a preview of the ability to use a `suspend` functional type as a supertype with some limitations.

```
class MyClass: suspend () -> Unit {
    override suspend fun invoke() { TODO() }
}
```

Use the `-language-version 1.6` compiler option to enable the feature:

Kotlin

```
kotlin {
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.6"
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.6'
        }
    }
}
```

The feature has the following restrictions:

- You can't mix an ordinary functional type and a `suspend` functional type as supertype. This is because of the implementation details of `suspend` functional types in the JVM backend. They are represented in it as ordinary

functional types with a marker interface. Because of the marker interface, there is no way to tell which of the superinterfaces are suspended and which are ordinary.

- You can't use multiple `suspend` functional supertypes. If there are type checks, you also can't use multiple ordinary functional supertypes.

Requiring opt-in on implicit usages of experimental APIs

> The opt-in requirement mechanism is Experimental. It may change at any time. See how to opt-in. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

The author of a library can mark an experimental API as requiring opt-in to inform users about its experimental state. The compiler raises a warning or error when the API is used and requires explicit consent to suppress it.

In Kotlin 1.5.30, the compiler treats any declaration that has an experimental type in the signature as experimental. Namely, it requires opt-in even for implicit usages of an experimental API. For example, if the function's return type is marked as an experimental API element, a usage of the function requires you to opt-in even if the declaration is not marked as requiring an opt-in explicitly.

```
// Library code

@RequiresOptIn(message = "This API is experimental.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS)
annotation class MyDateTime // Opt-in requirement annotation

@MyDateTime
class DateProvider // A class requiring opt-in

// Client code

// Warning: experimental API usage
fun createDateSource(): DateProvider { /* ... */ }

fun getDate(): Date {
    val dateSource = createDateSource() // Also warning: experimental API usage
    // ...
}
```

Learn more about opt-in requirements.

Changes to using opt-in requirement annotations with different targets

> The opt-in requirement mechanism is Experimental. It may change at any time. See how to opt-in. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 1.5.30 presents new rules for using and declaring opt-in requirement annotations on different targets. The compiler now reports an error for use cases that are impractical to handle at compile time. In Kotlin 1.5.30:

- Marking local variables and value parameters with opt-in requirement annotations is forbidden at the use site.
- Marking override is allowed only if its basic declaration is also marked.
- Marking backing fields and getters is forbidden. You can mark the basic property instead.

- Setting `TYPE` and `TYPE_PARAMETER` annotation targets is forbidden at the opt-in requirement annotation declaration site.

Learn more about opt-in requirements.

Improvements to type inference for recursive generic types

In Kotlin and Java, you can define a recursive generic type, which references itself in its type parameters. In Kotlin 1.5.30, the Kotlin compiler can infer a type argument based only on upper bounds of the corresponding type parameter if it is a recursive generic. This makes it possible to create various patterns with recursive generic types that are often used in Java to make builder APIs.

```
// Kotlin 1.5.20
val containerA = PostgreSQLContainer<Nothing>(DockerImageName.parse("postgres:13-alpine")).apply {
    withDatabaseName("db")
    withUsername("user")
    withPassword("password")
    withInitScript("sql/schema.sql")
}

// Kotlin 1.5.30
val containerB = PostgreSQLContainer(DockerImageName.parse("postgres:13-alpine"))
    .withDatabaseName("db")
    .withUsername("user")
    .withPassword("password")
    .withInitScript("sql/schema.sql")
```

You can enable the improvements by passing the `-Xself-upper-bound-inference` or the `-language-version 1.6` compiler options. See other examples of newly supported use cases in this YouTrack ticket.

Eliminating builder inference restrictions

Builder inference is a special kind of type inference that allows you to infer the type arguments of a call based on type information from other calls inside its lambda argument. This can be useful when calling generic builder functions such as `buildList()` or `sequence()`: `buildList { add("string") }`.

Inside such a lambda argument, there was previously a limitation on using the type information that the builder inference tries to infer. This means you can only specify it and cannot get it. For example, you cannot call `get()` inside a lambda argument of `buildList()` without explicitly specified type arguments.

Kotlin 1.5.30 removes these limitations with the `-Xunrestricted-builder-inference` compiler option. Add this option to enable previously prohibited calls inside a lambda argument of generic builder functions:

```
@kotlin.ExperimentalStdlibApi
val list = buildList {
    add("a")
    add("b")
    set(1, null)
    val x = get(1)
    if (x != null) {
        removeAt(1)
    }
}

@kotlin.ExperimentalStdlibApi
val map = buildMap {
    put("a", 1)
    put("b", 1.1)
    put("c", 2f)
}
```

Also, you can enable this feature with the `-language-version 1.6` compiler option.

Kotlin/JVM

With Kotlin 1.5.30, Kotlin/JVM receives the following features:

- Instantiation of annotation classes
- Improved nullability annotation support configuration

See the Gradle section for Kotlin Gradle plugin updates on the JVM platform.

Instantiation of annotation classes

> Instantiation of annotation classes is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

With Kotlin 1.5.30 you can now call constructors of annotation classes in arbitrary code to obtain a resulting instance. This feature covers the same use cases as the Java convention that allows the implementation of an annotation interface.

```
annotation class InfoMarker(val info: String)

fun processInfo(marker: InfoMarker) = ...

fun main(args: Array<String>) {
    if (args.size != 0)
        processInfo(getAnnotationReflective(args))
    else
        processInfo(InfoMarker("default"))
}
```

Use the `-language-version 1.6` compiler option to enable this feature. Note that all current annotation class limitations, such as restrictions to define non-`val` parameters or members different from secondary constructors, remain intact.

Learn more about instantiation of annotation classes in this KEEP

Improved nullability annotation support configuration

The Kotlin compiler can read various types of nullability annotations to get nullability information from Java. This information allows it to report nullability mismatches in Kotlin when calling Java code.

In Kotlin 1.5.30, you can specify whether the compiler reports a nullability mismatch based on the information from specific types of nullability annotations. Just use the compiler option `-Xnullability-annotations=@<package-name>:<report-level>`. In the argument, specify the fully qualified nullability annotations package and one of these report levels:

- `ignore` to ignore nullability mismatches
- `warn` to report warnings
- `strict` to report errors.

See the full list of supported nullability annotations along with their fully qualified package names.

Here is an example showing how to enable error reporting for the newly supported RxJava 3 nullability annotations: `-Xnullability-annotations=@io.reactivex.rxjava3.annotations:strict`. Note that all such nullability mismatches are warnings by default.

Kotlin/Native

Kotlin/Native has received various changes and improvements:

- Apple silicon support
- Improved Kotlin DSL for the CocoaPods Gradle plugin
- Experimental interoperability with Swift 5.5 async/await
- Improved Swift/Objective-C mapping for objects and companion objects
- Deprecation of linkage against DLLs without import libraries for MinGW targets

Apple silicon support

Kotlin 1.5.30 introduces native support for Apple silicon.

Previously, the Kotlin/Native compiler and tooling required the Rosetta translation environment for working on Apple silicon hosts. In Kotlin 1.5.30, the translation environment is no longer needed – the compiler and tooling can run on Apple silicon hardware without requiring any additional actions.

We've also introduced new targets that make Kotlin code run natively on Apple silicon:

- `macosArm64`
- `iosSimulatorArm64`
- `watchosSimulatorArm64`
- `tvosSimulatorArm64`

They are available on both Intel-based and Apple silicon hosts. All existing targets are available on Apple silicon hosts as well.

Note that in 1.5.30 we provide only basic support for Apple silicon targets in the `kotlin-multiplatform` Gradle plugin. Particularly, the new simulator targets aren't included in the `ios`, `tvos`, and `watchos` target shortcuts. We will keep working to improve the user experience with the new targets.

Improved Kotlin DSL for the CocoaPods Gradle plugin

New parameters for Kotlin/Native frameworks

Kotlin 1.5.30 introduces the improved CocoaPods Gradle plugin DSL for Kotlin/Native frameworks. In addition to the name of the framework, you can specify other parameters in the Pod configuration:

- Specify the dynamic or static version of the framework
- Enable export dependencies explicitly
- Enable Bitcode embedding

To use the new DSL, update your project to Kotlin 1.5.30, and specify the parameters in the `cocoapods` section of your `build.gradle(.kts)` file:

```
cocoapods {
    frameworkName = "MyFramework" // This property is deprecated
    // and will be removed in future versions
    // New DSL for framework configuration:
    framework {
        // All Framework properties are supported
        // Framework name configuration. Use this property instead of
        // deprecated 'frameworkName'
        baseName = "MyFramework"
        // Dynamic framework support
        isStatic = false
        // Dependency export
        export(project(":anotherKMMModule"))
        transitiveExport = false // This is default.
        // Bitcode embedding
        embedBitcode(BITCODE)
    }
}
```

Support custom names for Xcode configuration

The Kotlin CocoaPods Gradle plugin supports custom names in the Xcode build configuration. It will also help you if you're using special names for the build configuration in Xcode, for example `Staging`.

To specify a custom name, use the `xcodeConfigurationToNativeBuildType` parameter in the `cocoapods` section of your `build.gradle(.kts)` file:

```
cocoapods {
    // Maps custom Xcode configuration to NativeBuildType
    xcodeConfigurationToNativeBuildType["CUSTOM_DEBUG"] = NativeBuildType.DEBUG
    xcodeConfigurationToNativeBuildType["CUSTOM_RELEASE"] = NativeBuildType.RELEASE
}
```

This parameter will not appear in the Podspec file. When Xcode runs the Gradle build process, the Kotlin CocoaPods Gradle plugin will select the necessary native build type.

> There's no need to declare the `Debug` and `Release` configurations because they are supported by default.

Experimental interoperability with Swift 5.5 async/await

> Concurrency interoperability with Swift async/await is Experimental. It may be dropped or changed at any time. You should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

We added support for calling Kotlin's suspending functions from Objective-C and Swift in 1.4.0, and now we're improving it to keep up with a new Swift 5.5 feature – concurrency with `async` and `await` modifiers.

The Kotlin/Native compiler now emits the `_Nullable_result` attribute in the generated Objective-C headers for suspending functions with nullable return types. This makes it possible to call them from Swift as `async` functions with the proper nullability.

Note that this feature is experimental and can be affected in the future by changes in both Kotlin and Swift. For now, we're offering a preview of this feature that has certain limitations, and we are eager to hear what you think. Learn more about its current state and leave your feedback in this YouTrack issue.

Improved Swift/Objective-C mapping for objects and companion objects

Getting objects and companion objects can now be done in a way that is more intuitive for native iOS developers. For example, if you have the following objects in Kotlin:

```
object MyObject {
    val x = "Some value"
}

class MyClass {
    companion object {
        val x = "Some value"
    }
}
```

To access them in Swift, you can use the `shared` and `companion` properties:

```
MyObject.shared
MyObject.shared.x
MyClass.companion
MyClass.Companion.shared
```

Learn more about Swift/Objective-C interoperability.

Deprecation of linkage against DLLs without import libraries for MinGW targets

LLD is a linker from the LLVM project, which we plan to start using in Kotlin/Native for MinGW targets because of its benefits over the default ld.bfd – primarily its better performance.

However, the latest stable version of LLD doesn't support direct linkage against DLL for MinGW (Windows) targets. Such linkage requires using import libraries. Although they aren't needed with Kotlin/Native 1.5.30, we're adding a warning to inform you that such usage is incompatible with LLD that will become the default linker for MinGW in the future.

Please share your thoughts and concerns about the transition to the LLD linker in this YouTrack issue.

Kotlin Multiplatform

1.5.30 brings the following notable updates to Kotlin Multiplatform:

- Ability to use custom `cinterop` libraries in shared native code
- Support for XCFrameworks
- New default publishing setup for Android artifacts

Ability to use custom cinterop libraries in shared native code

Kotlin Multiplatform gives you an option to use platform-dependent interop libraries in shared source sets. Before 1.5.30, this worked only with platform libraries shipped with Kotlin/Native distribution. Starting from 1.5.30, you can use it with your custom `cinterop` libraries. To enable this feature, add the `kotlin.mpp.enableCInteropCommonization=true` property in your `gradle.properties`:

```
kotlin.mpp.enableGranularSourceSetsMetadata=true
kotlin.native.enableDependencyPropagation=false
kotlin.mpp.enableCInteropCommonization=true
```

Support for XCFrameworks

All Kotlin Multiplatform projects can now have XCFrameworks as an output format. Apple introduced XCFrameworks as a replacement for universal (fat) frameworks. With the help of XCFrameworks you:

- Can gather logic for all the target platforms and architectures in a single bundle.
- Don't need to remove all unnecessary architectures before publishing the application to the App Store.

XCFrameworks is useful if you want to use your Kotlin framework for devices and simulators on Apple M1.

To use XCFrameworks, update your `build.gradle(.kts)` script:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.plugin.mpp.apple.XCFramework

plugins {
    kotlin("multiplatform")
}

kotlin {
    val xcf = XCFramework()

    ios {
        binaries.framework {
            baseName = "shared"
            xcf.add(this)
        }
    }
    watchos {
        binaries.framework {
            baseName = "shared"
            xcf.add(this)
        }
    }
    tvos {
        binaries.framework {
            baseName = "shared"
            xcf.add(this)
        }
    }
}
```

Groovy

```
import org.jetbrains.kotlin.gradle.plugin.mpp.apple.XCFrameworkConfig

plugins {
    id 'org.jetbrains.kotlin.multiplatform'
}

kotlin {
    def xcf = new XCFrameworkConfig(project)

    ios {
        binaries.framework {
            baseName = "shared"
            xcf.add(it)
        }
    }
    watchos {
        binaries.framework {
            baseName = "shared"
            xcf.add(it)
        }
    }
    tvos {
        binaries.framework {
            baseName = "shared"
            xcf.add(it)
        }
    }
}
```

When you declare XCFrameworks, these new Gradle tasks will be registered:

- `assembleXCFramework`
- `assembleDebugXCFramework` (additionally debug artifact that contains dSYMs)
- `assembleReleaseXCFramework`

Learn more about XCFrameworks in this WWDC video.

New default publishing setup for Android artifacts

Using the `maven-publish` Gradle plugin, you can publish your multiplatform library for the Android target by specifying Android variant names in the build script. The Kotlin Gradle plugin will generate publications automatically.

Before 1.5.30, the generated publication metadata included the build type attributes for every published Android variant, making it compatible only with the same build type used by the library consumer. Kotlin 1.5.30 introduces a new default publishing setup:

- If all Android variants that the project publishes have the same build type attribute, then the published variants won't have the build type attribute and will be compatible with any build type.
- If the published variants have different build type attributes, then only those with the `release` value will be published without the build type attribute. This makes the release variants compatible with any build type on the consumer side, while non-release variants will only be compatible with the matching consumer build types.

To opt-out and keep the build type attributes for all variants, you can set this Gradle property: `kotlin.android.buildTypeAttribute.keep=true`.

Kotlin/JS

Two major improvements are coming to Kotlin/JS with 1.5.30:

- JS IR compiler backend reaches Beta
- Better debugging experience for applications with the Kotlin/JS IR backend

JS IR compiler backend reaches Beta

The IR-based compiler backend for Kotlin/JS, which was introduced in 1.4.0 in Alpha, has reached Beta.

Previously, we published the migration guide for the JS IR backend to help you migrate your projects to the new backend. Now we would like to present the Kotlin/JS Inspection Pack IDE plugin, which displays the required changes directly in IntelliJ IDEA.

Better debugging experience for applications with the Kotlin/JS IR backend

Kotlin 1.5.30 brings JavaScript source map generation for the Kotlin/JS IR backend. This will improve the Kotlin/JS debugging experience when the IR backend is enabled, with full debugging support that includes breakpoints, stepping, and readable stack traces with proper source references.

Learn how to debug Kotlin/JS in the browser or IntelliJ IDEA.

Gradle

As a part of our mission to improve the Kotlin Gradle plugin user experience, we've implemented the following features:

- Support for Java toolchains, which includes an ability to specify a JDK home with the `UsesKotlinJavaToolchain` interface for older Gradle versions
- An easier way to explicitly specify the Kotlin daemon's JVM arguments

Support for Java toolchains

Gradle 6.7 introduced the "Java toolchains support" feature. Using this feature, you can:

- Run compilations, tests, and executables using JDKs and JREs that are different from the Gradle ones.
- Compile and test code with an unreleased language version.

With toolchains support, Gradle can autodetect local JDKs and install missing JDKs that Gradle requires for the build. Now Gradle itself can run on any JDK and still reuse the build cache feature.

The Kotlin Gradle plugin supports Java toolchains for Kotlin/JVM compilation tasks. A Java toolchain:

- Sets the `jdkHome` option available for JVM targets.

> The ability to set the `jdkHome` option directly has been deprecated.

- Sets the `kotlinOptions.jvmTarget` to the toolchain's JDK version if the user didn't set the `jvmTarget` option explicitly. If the toolchain is not configured, the `jvmTarget` field uses the default value. Learn more about JVM target compatibility.
- Affects which JDK `kapt` workers are running on.

Use the following code to set a toolchain. Replace the placeholder `<MAJOR_JDK_VERSION>` with the JDK version you would like to use:

Kotlin

```
kotlin {
    jvmToolchain {
        (this as JavaToolchainSpec).languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
// "8"
    }
}
```

Groovy

```
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)) // "8"
    }
}
```

Note that setting a toolchain via the `kotlin` extension will update the toolchain for Java compile tasks as well.

You can set a toolchain via the `java` extension, and Kotlin compilation tasks will use it:

```
java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)) // "8"
    }
}
```

For information about setting any JDK version for `KotlinCompile` tasks, look through the docs about setting the JDK version with the Task DSL.

For Gradle versions from 6.1 to 6.6, use the `UsesKotlinJavaToolchain` interface to set the JDK home.

Ability to specify JDK home with UsesKotlinJavaToolchain interface

All Kotlin tasks that support setting the JDK via `kotlinOptions` now implement the `UsesKotlinJavaToolchain` interface. To set the JDK home, put a path to your JDK and replace the `<JDK_VERSION>` placeholder:

Kotlin

```
project.tasks
    .withType<UsesKotlinJavaToolchain>()
    .configureEach {
        it.kotlinJavaToolchain.jdk.use(
            "/path/to/local/jdk",
            JavaVersion.<LOCAL_JDK_VERSION>
        )
    }
```

Groovy

```
project.tasks
    .withType(UsesKotlinJavaToolchain.class)
    .configureEach {
        it.kotlinJavaToolchain.jdk.use(
            '/path/to/local/jdk',
            JavaVersion.<LOCAL_JDK_VERSION>
        )
    }
```

Use the `UsesKotlinJavaToolchain` interface for Gradle versions from 6.1 to 6.6. Starting from Gradle 6.7, use the Java toolchains instead.

When using this feature, note that kapt task workers will only use process isolation mode, and the `kapt.workers.isolation` property will be ignored.

Easier way to explicitly specify Kotlin daemon JVM arguments

In Kotlin 1.5.30, there's a new logic for the Kotlin daemon's JVM arguments. Each of the options in the following list overrides the ones that came before it:

- If nothing is specified, the Kotlin daemon inherits arguments from the Gradle daemon (as before). For example, in the `gradle.properties` file:

  ```
  org.gradle.jvmargs=-Xmx1500m -Xms=500m
  ```

- If the Gradle daemon's JVM arguments have the `kotlin.daemon.jvm.options` system property, use it as before:

  ```
  org.gradle.jvmargs=-Dkotlin.daemon.jvm.options=-Xmx1500m -Xms=500m
  ```

- You can add the `kotlin.daemon.jvmargs` property in the `gradle.properties` file:

  ```
  kotlin.daemon.jvmargs=-Xmx1500m -Xms=500m
  ```

- You can specify arguments in the `kotlin` extension:

Kotlin

```
kotlin {
    kotlinDaemonJvmArgs = listOf("-Xmx486m", "-Xms256m", "-XX:+UseParallelGC")
}
```

Groovy

```
kotlin {
    kotlinDaemonJvmArgs = ["-Xmx486m", "-Xms256m", "-XX:+UseParallelGC"]
}
```

- You can specify arguments for a specific task:

Kotlin

```
tasks
    .matching { it.name == "compileKotlin" && it is CompileUsingKotlinDaemon }
    .configureEach {
        (this as CompileUsingKotlinDaemon).kotlinDaemonJvmArguments.set(listOf("-Xmx486m", "-
Xms256m", "-XX:+UseParallelGC"))
    }
```

Groovy

```
tasks
    .matching {
        it.name == "compileKotlin" && it instanceof CompileUsingKotlinDaemon
    }
    .configureEach {
        kotlinDaemonJvmArguments.set(["-Xmx1g", "-Xms512m"])
    }
```

In this case a new Kotlin daemon instance can start on task execution. Learn more about the Kotlin daemon's interactions with JVM arguments.

For more information about the Kotlin daemon, see the Kotlin daemon and using it with Gradle.

Standard library

Kotlin 1.5.30 is bringing improvements to the standard library's `Duration` and `Regex` APIs:

- Changing `Duration.toString()` output
- Parsing Duration from String
- Matching with Regex at a particular position
- Splitting Regex to a sequence

Changing Duration.toString() output

> The Duration API is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate hearing your feedback on it in YouTrack.

Before Kotlin 1.5.30, the `Duration.toString()` function would return a string representation of its argument expressed in the unit that yielded the most compact and readable number value. From now on, it will return a string value expressed as a combination of numeric components, each in its own unit. Each component is a number followed by the unit's abbreviated name: `d`, `h`, `m`, `s`. For example:

Example of function call	Previous output	Current output
Duration.days(45).toString()	`45.0d`	`45d`
Duration.days(1.5).toString()	`36.0h`	`1d 12h`
Duration.minutes(1230).toString()	`20.5h`	`20h 30m`
Duration.minutes(2415).toString()	`40.3h`	`1d 16h 15m`
Duration.minutes(920).toString()	`920m`	`15h 20m`
Duration.seconds(1.546).toString()	`1.55s`	`1.546s`
Duration.milliseconds(25.12).toString()	`25.1ms`	`25.12ms`

The way negative durations are represented has also been changed. A negative duration is prefixed with a minus sign (`-`), and if it consists of multiple components, it is surrounded with parentheses: `-12m` and `-(1h 30m)`.

Note that small durations of less than one second are represented as a single number with one of the subsecond units. For example, `ms` (milliseconds), `us` (microseconds), or `ns` (nanoseconds): `140.884ms`, `500us`, `24ns`. Scientific notation is no longer used to represent them.

If you want to express duration in a single unit, use the overloaded `Duration.toString(unit, decimals)` function.

> We recommend using `Duration.toIsoString()` in certain cases, including serialization and interchange. `Duration.toIsoString()` uses the stricter ISO-8601 format instead of `Duration.toString()`.

Parsing Duration from String

> The Duration API is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate hearing your feedback on it in this issue.

In Kotlin 1.5.30, there are new functions in the Duration API:

- `parse()`, which supports parsing the outputs of:
  - `toString()`.
  - `toString(unit, decimals)`.
  - `toIsoString()`.
- `parseIsoString()`, which only parses from the format produced by `toIsoString()`.
- `parseOrNull()` and `parseIsoStringOrNull()`, which behave like the functions above but return `null` instead of throwing `IllegalArgumentException` on invalid duration formats.

Here are some examples of `parse()` and `parseOrNull()` usages:

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime

@ExperimentalTime
fun main() {
    val isoFormatString = "PT1H30M"
    val defaultFormatString = "1h 30m"
    val singleUnitFormatString = "1.5h"
    val invalidFormatString = "1 hour 30 minutes"
    println(Duration.parse(isoFormatString)) // "1h 30m"
    println(Duration.parse(defaultFormatString)) // "1h 30m"
    println(Duration.parse(singleUnitFormatString)) // "1h 30m"
    //println(Duration.parse(invalidFormatString)) // throws exception
    println(Duration.parseOrNull(invalidFormatString)) // "null"
}
```

And here are some examples of `parseIsoString()` and `parseIsoStringOrNull()` usages:

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime

@ExperimentalTime
fun main() {
    val isoFormatString = "PT1H30M"
    val defaultFormatString = "1h 30m"
    println(Duration.parseIsoString(isoFormatString)) // "1h 30m"
    //println(Duration.parseIsoString(defaultFormatString)) // throws exception
    println(Duration.parseIsoStringOrNull(defaultFormatString)) // "null"
}
```

Matching with Regex at a particular position

> `Regex.matchAt()` and `Regex.matchesAt()` functions are Experimental. They may be dropped or changed at any time. Use them only for evaluation purposes. We would appreciate hearing your feedback on them in YouTrack.

The new `Regex.matchAt()` and `Regex.matchesAt()` functions provide a way to check whether a regex has an exact match at a particular position in a `String` or `CharSequence`.

`matchesAt()` returns a boolean result:

```
fun main(){
    val releaseText = "Kotlin 1.5.30 is released!"
    // regular expression: one digit, dot, one digit, dot, one or more digits
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()
    println(versionRegex.matchesAt(releaseText, 0)) // "false"
    println(versionRegex.matchesAt(releaseText, 7)) // "true"
}
```

`matchAt()` returns the match if one is found or `null` if one isn't:

```
fun main(){
    val releaseText = "Kotlin 1.5.30 is released!"
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()
    println(versionRegex.matchAt(releaseText, 0)) // "null"
    println(versionRegex.matchAt(releaseText, 7)?.value) // "1.5.30"
}
```

Splitting Regex to a sequence

> `Regex.splitToSequence()` and `CharSequence.splitToSequence(Regex)` functions are Experimental. They may be dropped or changed at any time. Use them only for evaluation purposes. We would appreciate hearing your feedback on them in YouTrack.

The new `Regex.splitToSequence()` function is a lazy counterpart of `split()`. It splits the string around matches of the given regex, but it returns the result as a Sequence so that all operations on this result are executed lazily.

```
fun main(){
    val colorsText = "green, red , brown&blue, orange, pink&green"
    val regex = "[,\\s]+".toRegex()
    val mixedColor = regex.splitToSequence(colorsText)
        .onEach { println(it) }
        .firstOrNull { it.contains('&') }
    println(mixedColor) // "brown&blue"
}
```

A similar function was also added to `CharSequence`:

```
val mixedColor = colorsText.splitToSequence(regex)
```

Serialization 1.3.0-RC

`kotlinx.serialization` 1.3.0-RC is here with new JSON serialization capabilities:

- Java IO streams serialization
- Property-level control over default values
- An option to exclude null values from serialization
- Custom class discriminators in polymorphic serialization

Learn more in the changelog.

What's new in Kotlin 1.5.20

Released: 24 June 2021

Kotlin 1.5.20 has fixes for issues discovered in the new features of 1.5.0, and it also includes various tooling improvements.

You can find an overview of the changes in the release blog post and this video:

Watch video online.

For information about the Kotlin release cycle, see Kotlin release process.

Kotlin/JVM

Kotlin 1.5.20 is receiving the following updates on the JVM platform:

- String concatenation via invokedynamic
- Support for JSpecify nullness annotations
- Support for calling Java's Lombok-generated methods within modules that have Kotlin and Java code

String concatenation via invokedynamic

Kotlin 1.5.20 compiles string concatenations into dynamic invocations (`invokedynamic`) on JVM 9+ targets, thereby keeping up with modern Java versions. More precisely, it uses `StringConcatFactory.makeConcatWithConstants()` for string concatenation.

To switch back to concatenation via `StringBuilder.append()` used in previous versions, add the compiler option `-Xstring-concat=inline` .

Learn how to add compiler options in Gradle, Maven, and the command-line compiler.

Support for JSpecify nullness annotations

The Kotlin compiler can read various types of nullability annotations to pass nullability information from Java to Kotlin. Version 1.5.20 introduces support for the JSpecify project, which includes the standard unified set of Java nullness annotations.

With JSpecify, you can provide more detailed nullability information to help Kotlin keep null-safety interoperating with Java. You can set default nullability for the declaration, package, or module scope, specify parametric nullability, and more. You can find more details about this in the JSpecify user guide.

Here is the example of how Kotlin can handle JSpecify annotations:

```
// JavaClass.java
import org.jspecify.nullness.*;

@NullMarked
public class JavaClass {
  public String notNullableString() { return ""; }
  public @Nullable String nullableString() { return ""; }
}
```

```
// Test.kt
fun kotlinFun() = with(JavaClass()) {
  notNullableString().length // OK
  nullableString().length    // Warning: receiver nullability mismatch
}
```

In 1.5.20, all nullability mismatches according to the JSpecify-provided nullability information are reported as warnings. Use the `-Xjspecify-annotations=strict` and `-Xtype-enhancement-improvements-strict-mode` compiler options to enable strict mode (with error reporting) when working with JSpecify. Please note that the JSpecify project is under active development. Its API and implementation can change significantly at any time.

Learn more about null-safety and platform types.

Support for calling Java's Lombok-generated methods within modules that have Kotlin and Java code

> The Lombok compiler plugin is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

Kotlin 1.5.20 introduces an experimental Lombok compiler plugin. This plugin makes it possible to generate and use Java's Lombok declarations within modules that have Kotlin and Java code. Lombok annotations work only in Java sources and are ignored if you use them in Kotlin code.

The plugin supports the following annotations:

- `@Getter`, `@Setter`
- `@NoArgsConstructor`, `@RequiredArgsConstructor`, and `@AllArgsConstructor`
- `@Data`
- `@With`
- `@Value`

We're continuing to work on this plugin. To find out the detailed current state, visit the Lombok compiler plugin's README.

Currently, we don't have plans to support the `@Builder` annotation. However, we can consider this if you vote for `@Builder` in YouTrack.

Learn how to configure the Lombok compiler plugin.

Kotlin/Native

Kotlin/Native 1.5.20 offers a preview of the new feature and the tooling improvements:

- Opt-in export of KDoc comments to generated Objective-C headers
- Compiler bug fixes
- Improved performance of Array.copyInto() inside one array

Opt-in export of KDoc comments to generated Objective-C headers

> The ability to export KDoc comments to generated Objective-C headers is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

You can now set the Kotlin/Native compiler to export the documentation comments (KDoc) from Kotlin code to the Objective-C frameworks generated from it, making them visible to the frameworks' consumers.

For example, the following Kotlin code with KDoc:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit integer.
 */
fun printSum(a: Int, b: Int) = println(a.toLong() + b)
```

produces the following Objective-C headers:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit integer.
 */
+ (void)printSumA:(int32_t)a b:(int32_t)b __attribute__((swift_name("printSum(a:b:)")));
```

This also works well with Swift.

To try out this ability to export KDoc comments to Objective-C headers, use the `-Xexport-kdoc` compiler option. Add the following lines to `build.gradle(.kts)` of the Gradle projects you want to export comments from:

Kotlin

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        compilations.get("main").kotlinOptions.freeCompilerArgs += "-Xexport-kdoc"
    }
}
```

Groovy

```
kotlin {
    targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget) {
        compilations.get("main").kotlinOptions.freeCompilerArgs += "-Xexport-kdoc"
    }
}
```

We would be very grateful if you would share your feedback with us using this YouTrack ticket.

Compiler bug fixes

The Kotlin/Native compiler has received multiple bug fixes in 1.5.20. You can find the complete list in the changelog.

There is an important bug fix that affects compatibility: in previous versions, string constants that contained incorrect UTF surrogate pairs were losing their values during compilation. Now such values are preserved. Application developers can safely update to 1.5.20 – nothing will break. However, libraries compiled with 1.5.20 are incompatible with earlier compiler versions. See this YouTrack issue for details.

Improved performance of Array.copyInto() inside one array

We've improved the way `Array.copyInto()` works when its source and destination are the same array. Now such operations finish up to 20 times faster (depending on the number of objects being copied) due to memory management optimizations for this use case.

Kotlin/JS

With 1.5.20, we're publishing a guide that will help you migrate your projects to the new IR-based backend for Kotlin/JS.

Migration guide for the JS IR backend

The new migration guide for the JS IR backend identifies issues you may encounter during migration and provides solutions for them. If you find any issues that aren't covered in the guide, please report them to our issue tracker.

Gradle

Kotlin 1.5.20 introduces the following features that can improve the Gradle experience:

- Caching for annotation processors classloaders in kapt
- Deprecation of the `kotlin.parallel.tasks.in.project` build property

Caching for annotation processors' classloaders in kapt

> Caching for annotation processors' classloaders in kapt is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

There is now a new experimental feature that makes it possible to cache the classloaders of annotation processors in kapt. This feature can increase the speed of kapt for consecutive Gradle runs.

To enable this feature, use the following properties in your `gradle.properties` file:

```
# positive value will enable caching
# use the same value as the number of modules that use kapt
kapt.classloaders.cache.size=5

# disable for caching to work
kapt.include.compile.classpath=false
```

Learn more about kapt.

Deprecation of the kotlin.parallel.tasks.in.project build property

With this release, Kotlin parallel compilation is controlled by the Gradle parallel execution flag `--parallel`. Using this flag, Gradle executes tasks concurrently, increasing the speed of compiling tasks and utilizing the resources more efficiently.

You no longer need to use the `kotlin.parallel.tasks.in.project` property. This property has been deprecated and will be removed in the next major release.

Standard library

Kotlin 1.5.20 changes the platform-specific implementations of several functions for working with characters and as a result brings unification across platforms:

- Support for all Unicode digits in Char.digitToInt() for Kotlin/Native and Kotlin/JS.
- Unification of Char.isLowerCase()/isUpperCase() implementations across platforms.

Support for all Unicode digits in Char.digitToInt() in Kotlin/Native and Kotlin/JS

`Char.digitToInt()` returns the numeric value of the decimal digit that the character represents. Before 1.5.20, the function supported all Unicode digit characters only for Kotlin/JVM: implementations on the Native and JS platforms supported only ASCII digits.

From now, both with Kotlin/Native and Kotlin/JS, you can call `Char.digitToInt()` on any Unicode digit character and get its numeric representation.

```
fun main() {
    val ten = '\u0661'.digitToInt() + '\u0039'.digitToInt() // ARABIC-INDIC DIGIT ONE + DIGIT NINE
    println(ten)
}
```

Unification of Char.isLowerCase()/isUpperCase() implementations across platforms

The functions `Char.isUpperCase()` and `Char.isLowerCase()` return a boolean value depending on the case of the character. For Kotlin/JVM, the implementation checks both the `General_Category` and the `Other_Uppercase` / `Other_Lowercase` Unicode properties.

Prior to 1.5.20, implementations for other platforms worked differently and considered only the general category. In 1.5.20, implementations are unified across platforms and use both properties to determine the character case:

```
fun main() {
    val latinCapitalA = 'A' // has "Lu" general category
    val circledLatinCapitalA = 'Ⓐ' // has "Other_Uppercase" property
    println(latinCapitalA.isUpperCase() && circledLatinCapitalA.isUpperCase())
}
```

What's new in Kotlin 1.5.0

Released: 5 May 2021

Kotlin 1.5.0 introduces new language features, stable IR-based JVM compiler backend, performance improvements, and evolutionary changes such as stabilizing experimental features and deprecating outdated ones.

You can also find an overview of the changes in the release blog post.

> For information about the Kotlin release cycle, see Kotlin release process.

Language features

Kotlin 1.5.0 brings stable versions of the new language features presented for preview in 1.4.30:

- JVM records support
- Sealed interfaces and sealed class improvements
- Inline classes

Detailed descriptions of these features are available in this blog post and the corresponding pages of Kotlin documentation.

JVM records support

Java is evolving fast, and to make sure Kotlin remains interoperable with it, we've introduced support for one of its latest features – record classes.

Kotlin's support for JVM records includes bidirectional interoperability:

- In Kotlin code, you can use Java record classes like you would use typical classes with properties.
- To use a Kotlin class as a record in Java code, make it a `data` class and mark it with the `@JvmRecord` annotation.

```
@JvmRecord
data class User(val name: String, val age: Int)
```

Learn more about using JVM records in Kotlin.

Watch video online.

Sealed interfaces

Kotlin interfaces can now have the `sealed` modifier, which works on interfaces in the same way it works on classes: all implementations of a sealed interface are known at compile time.

```
sealed interface Polygon
```

You can rely on that fact, for example, to write exhaustive `when` expressions.

```
fun draw(polygon: Polygon) = when (polygon) {
    is Rectangle -> // ...
    is Triangle -> // ...
    // else is not needed - all possible implementations are covered
}
```

Additionally, sealed interfaces enable more flexible restricted class hierarchies because a class can directly inherit more than one sealed interface.

```
class FilledRectangle: Polygon, Fillable
```

Learn more about sealed interfaces.

Watch video online.

Package-wide sealed class hierarchies

Sealed classes can now have subclasses in all files of the same compilation unit and the same package. Previously, all subclasses had to appear in the same file.

Direct subclasses may be top-level or nested inside any number of other named classes, named interfaces, or named objects.

The subclasses of a sealed class must have a name that is properly qualified – they cannot be local or anonymous objects.

Learn more about sealed class hierarchies.

Inline classes

Inline classes are a subset of value-based classes that only hold values. You can use them as wrappers for a value of a certain type without the additional overhead that comes from using memory allocations.

Inline classes can be declared with the `value` modifier before the name of the class:

```
value class Password(val s: String)
```

The JVM backend also requires a special `@JvmInline` annotation:

```
@JvmInline
value class Password(val s: String)
```

The `inline` modifier is now deprecated with a warning.

Learn more about inline classes.

Watch video online.

Kotlin/JVM

Kotlin/JVM has received a number of improvements, both internal and user-facing. Here are the most notable among them:

- Stable JVM IR backend
- New default JVM target: 1.8
- SAM adapters via invokedynamic
- Lambdas via invokedynamic
- Deprecation of @JvmDefault and old Xjvm-default modes
- Improvements to handling nullability annotations

Stable JVM IR backend

The IR-based backend for the Kotlin/JVM compiler is now Stable and enabled by default.

Starting from Kotlin 1.4.0, early versions of the IR-based backend were available for preview, and it has now become the default for language version `1.5`. The old backend is still used by default for earlier language versions.

You can find more details about the benefits of the IR backend and its future development in this blog post.

If you need to use the old backend in Kotlin 1.5.0, you can add the following lines to the project's configuration file:

- In Gradle:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile> {
  kotlinOptions.useOldBackend = true
}
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {
 kotlinOptions.useOldBackend = true
}
```

- In Maven:

```
<configuration>
    <args>
        <arg>-Xuse-old-backend</arg>
    </args>
</configuration>
```

New default JVM target: 1.8

The default target version for Kotlin/JVM compilations is now `1.8`. The `1.6` target is deprecated.

If you need a build for JVM 1.6, you can still switch to this target. Learn how:

- in Gradle
- in Maven
- in the command-line compiler

SAM adapters via invokedynamic

Kotlin 1.5.0 now uses dynamic invocations (`invokedynamic`) for compiling SAM (Single Abstract Method) conversions:

- Over any expression if the SAM type is a Java interface
- Over lambda if the SAM type is a Kotlin functional interface

The new implementation uses `LambdaMetafactory.metafactory()` and auxiliary wrapper classes are no longer generated during compilation. This decreases the size of the application's JAR, which improves the JVM startup performance.

To roll back to the old implementation scheme based on anonymous class generation, add the compiler option `-Xsam-conversions=class`.

Learn how to add compiler options in Gradle, Maven, and the command-line compiler.

Lambdas via invokedynamic

> Compiling plain Kotlin lambdas into invokedynamic is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate hearing your feedback on it in YouTrack.

Kotlin 1.5.0 is introducing experimental support for compiling plain Kotlin lambdas (which are not converted to an instance of a functional interface) into dynamic invocations (`invokedynamic`). The implementation produces lighter binaries by using `LambdaMetafactory.metafactory()`, which effectively generates the necessary classes at runtime. Currently, it has three limitations compared to ordinary lambda compilation:

- A lambda compiled into invokedynamic is not serializable.
- Calling `toString()` on such a lambda produces a less readable string representation.
- Experimental `reflect` API does not support lambdas created with `LambdaMetafactory`.

To try this feature, add the `-Xlambdas=indy` compiler option. We would be grateful if you could share your feedback on it using this YouTrack ticket.

Learn how to add compiler options in Gradle, Maven, and command-line compiler.

Deprecation of @JvmDefault and old Xjvm-default modes

Prior to Kotlin 1.4.0, there was the `@JvmDefault` annotation along with `-Xjvm-default=enable` and `-Xjvm-default=compatibility` modes. They served to create the JVM default method for any particular non-abstract member in the Kotlin interface.

In Kotlin 1.4.0, we introduced the new `Xjvm-default` modes, which switch on default method generation for the whole project.

In Kotlin 1.5.0, we are deprecating `@JvmDefault` and the old Xjvm-default modes: `-Xjvm-default=enable` and `-Xjvm-default=compatibility`.

Learn more about default methods in the Java interop.

Improvements to handling nullability annotations

Kotlin supports handling type nullability information from Java with nullability annotations. Kotlin 1.5.0 introduces a number of improvements for the feature:

- It reads nullability annotations on type arguments in compiled Java libraries that are used as dependencies.
- It supports nullability annotations with the `TYPE_USE` target for:
  - Arrays

- Varargs
- Fields
- Type parameters and their bounds
- Type arguments of base classes and interfaces

- If a nullability annotation has multiple targets applicable to a type, and one of these targets is `TYPE_USE`, then `TYPE_USE` is preferred. For example, the method signature `@Nullable String[] f()` becomes `fun f(): Array<String?>!` if `@Nullable` supports both `TYPE_USE` and `METHOD` as targets.

For these newly supported cases, using the wrong type nullability when calling Java from Kotlin produces warnings. Use the `-Xtype-enhancement-improvements-strict-mode` compiler option to enable strict mode for these cases (with error reporting).

Learn more about null-safety and platform types.

Kotlin/Native

Kotlin/Native is now more performant and stable. The notable changes are:

- Performance improvements
- Deactivation of the memory leak checker

Performance improvements

In 1.5.0, Kotlin/Native is receiving a set of performance improvements that speed up both compilation and execution.

Compiler caches are now supported in debug mode for `linuxX64` (only on Linux hosts) and `iosArm64` targets. With compiler caches enabled, most debug compilations complete much faster, except for the first one. Measurements showed about a 200% speed increase on our test projects.

To use compiler caches for new targets, opt in by adding the following lines to the project's `gradle.properties`:

- For `linuxX64`: `kotlin.native.cacheKind.linuxX64=static`
- For `iosArm64`: `kotlin.native.cacheKind.iosArm64=static`

If you encounter any issues after enabling the compiler caches, please report them to our issue tracker YouTrack.

Other improvements speed up the execution of Kotlin/Native code:

- Trivial property accessors are inlined.
- `trimIndent()` on string literals is evaluated during the compilation.

Deactivation of the memory leak checker

The built-in Kotlin/Native memory leak checker has been disabled by default.

It was initially designed for internal use, and it is able to find leaks only in a limited number of cases, not all of them. Moreover, it later turned out to have issues that can cause application crashes. So we've decided to turn off the memory leak checker.

The memory leak checker can still be useful for certain cases, for example, unit testing. For these cases, you can enable it by adding the following line of code:

```
Platform.isMemoryLeakCheckerActive = true
```

Note that enabling the checker for the application runtime is not recommended.

Kotlin/JS

Kotlin/JS is receiving evolutionary changes in 1.5.0. We're continuing our work on moving the JS IR compiler backend towards stable and shipping other updates:

- Upgrade of webpack to version 5
- Frameworks and libraries for the IR compiler

Upgrade to webpack 5

The Kotlin/JS Gradle plugin now uses webpack 5 for browser targets instead of webpack 4. This is a major webpack upgrade that brings incompatible changes. If you're using a custom webpack configuration, be sure to check the webpack 5 release notes.

Learn more about bundling Kotlin/JS projects with webpack.

Frameworks and libraries for the IR compiler

The Kotlin/JS IR compiler is in Alpha. It may change incompatibly and require manual migration in the future. We would appreciate your feedback on it in YouTrack.

Along with working on the IR-based backend for Kotlin/JS compiler, we encourage and help library authors to build their projects in `both` mode. This means they are able to produce artifacts for both Kotlin/JS compilers, therefore growing the ecosystem for the new compiler.

Many well-known frameworks and libraries are already available for the IR backend: KVision, fritz2, doodle, and others. If you're using them in your project, you can already build it with the IR backend and see the benefits it brings.

If you're writing your own library, compile it in the 'both' mode so that your clients can also use it with the new compiler.

Kotlin Multiplatform

In Kotlin 1.5.0, choosing a testing dependency for each platform has been simplified and it is now done automatically by the Gradle plugin.

A new API for getting a char category is now available in multiplatform projects.

Standard library

The standard library has received a range of changes and improvements, from stabilizing experimental parts to adding new features:

- Stable unsigned integer types
- Stable locale-agnostic API for uppercase/lowercase text

- Stable Char-to-integer conversion API
- Stable Path API
- Floored division and the mod operator
- Duration API changes
- New API for getting a char category now available in multiplatform code
- New collections function firstNotNullOf()
- Strict version of String?.toBoolean()

You can learn more about the standard library changes in this blog post.

Watch video online.

Stable unsigned integer types

The `UInt`, `ULong`, `UByte`, `UShort` unsigned integer types are now Stable. The same goes for operations on these types, ranges, and progressions of them. Unsigned arrays and operations on them remain in Beta.

Learn more about unsigned integer types.

Stable locale-agnostic API for upper/lowercasing text

This release brings a new locale-agnostic API for uppercase/lowercase text conversion. It provides an alternative to the `toLowerCase()`, `toUpperCase()`, `capitalize()`, and `decapitalize()` API functions, which are locale-sensitive. The new API helps you avoid errors due to different locale settings.

Kotlin 1.5.0 provides the following fully Stable alternatives:

- For `String` functions:

Earlier versions	1.5.0 alternative
`String.toUpperCase()`	`String.uppercase()`
`String.toLowerCase()`	`String.lowercase()`
`String.capitalize()`	`String.replaceFirstChar { it.uppercase() }`
`String.decapitalize()`	`String.replaceFirstChar { it.lowercase() }`

- For `Char` functions:

Earlier versions	1.5.0 alternative
`Char.toUpperCase()`	`Char.uppercaseChar(): Char` `Char.uppercase(): String`

Earlier versions	1.5.0 alternative
`Char.toLowerCase()`	`Char.lowercaseChar(): Char` `Char.lowercase(): String`
`Char.toTitleCase()`	`Char.titlecaseChar(): Char` `Char.titlecase(): String`

For Kotlin/JVM, there are also overloaded `uppercase()`, `lowercase()`, and `titlecase()` functions with an explicit `Locale` parameter.

The old API functions are marked as deprecated and will be removed in a future release.

See the full list of changes to the text processing functions in KEEP.

Stable char-to-integer conversion API

Starting from Kotlin 1.5.0, new char-to-code and char-to-digit conversion functions are Stable. These functions replace the current API functions, which were often confused with the similar string-to-Int conversion.

The new API removes this naming confusion, making the code behavior more transparent and unambiguous.

This release introduces `Char` conversions that are divided into the following sets of clearly named functions:

- Functions to get the integer code of `Char` and to construct `Char` from the given code:

```
fun Char(code: Int): Char
fun Char(code: UShort): Char
val Char.code: Int
```

- Functions to convert `Char` to the numeric value of the digit it represents:

```
fun Char.digitToInt(radix: Int): Int
fun Char.digitToIntOrNull(radix: Int): Int?
```

- An extension function for `Int` to convert the non-negative single digit it represents to the corresponding `Char` representation:

```
fun Int.digitToChar(radix: Int): Char
```

The old conversion APIs, including `Number.toChar()` with its implementations (all except `Int.toChar()`) and `Char` extensions for conversion to a numeric type, like `Char.toInt()`, are now deprecated.

Learn more about the char-to-integer conversion API in KEEP.

Stable Path API

The experimental Path API with extensions for `java.nio.file.Path` is now Stable.

```
// list files in a directory
val kotlinFiles: List<Path> = Path("/home/user").listDirectoryEntries("*.kt")
```

Learn more about the Path API.

Floored division and the mod operator

New operations for modular arithmetics have been added to the standard library:

- `floorDiv()` returns the result of floored division. It is available for integer types.
- `mod()` returns the remainder of floored division (*modulus*). It is available for all numeric types.

These operations look quite similar to the existing division of integers and rem() function (or the `%` operator), but they work differently on negative numbers:

- `a.floorDiv(b)` differs from a regular `/` in that `floorDiv` rounds the result down (towards the lesser integer), whereas `/` truncates the result to the integer closer to 0.
- `a.mod(b)` is the difference between `a` and `a.floorDiv(b) * b`. It's either zero or has the same sign as `b`, while `a % b` can have a different one.

```
fun main() {
    println("Floored division -5/3: ${(-5).floorDiv(3)}")
    println( "Modulus: ${(-5).mod(3)}")

    println("Truncated division -5/3: ${-5 / 3}")
    println( "Remainder: ${-5 % 3}")
}
```

Duration API changes

> The Duration API is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate hearing your feedback on it in YouTrack.

There is an experimental Duration class for representing duration amounts in different time units. In 1.5.0, the Duration API has received the following changes:

- Internal value representation now uses `Long` instead of `Double` to provide better precision.
- There is a new API for conversion to a particular time unit in `Long`. It comes to replace the old API, which operates with `Double` values and is now deprecated. For example, `Duration.inWholeMinutes` returns the value of the duration expressed as `Long` and replaces `Duration.inMinutes`.
- There are new companion functions for constructing a `Duration` from a number. For example, `Duration.seconds(Int)` creates a `Duration` object representing an integer number of seconds. Old extension properties like `Int.seconds` are now deprecated.

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime

@ExperimentalTime
fun main() {
    val duration = Duration.milliseconds(120000)
    println("There are ${duration.inWholeSeconds} seconds in ${duration.inWholeMinutes} minutes")
}
```

New API for getting a char category now available in multiplatform code

Kotlin 1.5.0 introduces the new API for getting a character's category according to Unicode in multiplatform projects. Several functions are now available in all the platforms and in the common code.

Functions for checking whether a char is a letter or a digit:

- `Char.isDigit()`
- `Char.isLetter()`
- `Char.isLetterOrDigit()`

```
fun main() {
    val chars = listOf('a', '1', '+')
    val (letterOrDigitList, notLetterOrDigitList) = chars.partition { it.isLetterOrDigit() }
    println(letterOrDigitList) // [a, 1]
    println(notLetterOrDigitList) // [+]
}
```

Functions for checking the case of a char:

- `Char.isLowerCase()`
- `Char.isUpperCase()`
- `Char.isTitleCase()`

```
fun main() {
    val chars = listOf('Dž', 'Lj', 'Nj', 'Dz', '1', 'A', 'a', '+')
    val (titleCases, notTitleCases) = chars.partition { it.isTitleCase() }
    println(titleCases) // [Dž, Lj, Nj, Dz]
    println(notTitleCases) // [1, A, a, +]
}
```

Some other functions:

- `Char.isDefined()`
- `Char.isISOControl()`

The property `Char.category` and its return type enum class `CharCategory`, which indicates a char's general category according to Unicode, are now also available in multiplatform projects.

Learn more about characters.

New collections function firstNotNullOf()

The new `firstNotNullOf()` and `firstNotNullOfOrNull()` functions combine `mapNotNull()` with `first()` or `firstOrNull()`. They map the original collection with the custom selector function and return the first non-null value. If there is no such value, `firstNotNullOf()` throws an exception, and `firstNotNullOfOrNull()` returns null.

```
fun main() {
    val data = listOf("Kotlin", "1.5")
    println(data.firstNotNullOf(String::toDoubleOrNull))
    println(data.firstNotNullOfOrNull(String::toIntOrNull))
}
```

Strict version of String?.toBoolean()

Two new functions introduce case-sensitive strict versions of the existing String?.toBoolean():

- `String.toBooleanStrict()` throws an exception for all inputs except the literals `true` and `false`.
- `String.toBooleanStrictOrNull()` returns null for all inputs except the literals `true` and `false`.

```
fun main() {
    println("true".toBooleanStrict())
    println("1".toBooleanStrictOrNull())
    // println("1".toBooleanStrict()) // Exception
}
```

kotlin-test library

The kotlin-test library introduces some new features:

- Simplified test dependencies usage in multiplatform projects
- Automatic selection of a testing framework for Kotlin/JVM source sets
- Assertion function updates

Simplified test dependencies usage in multiplatform projects

Now you can use the `kotlin-test` dependency to add dependencies for testing in the `commonTest` source set, and the Gradle plugin will infer the corresponding platform dependencies for each test source set:

- `kotlin-test-junit` for JVM source sets, see automatic choice of a testing framework for Kotlin/JVM source sets
- `kotlin-test-js` for Kotlin/JS source sets
- `kotlin-test-common` and `kotlin-test-annotations-common` for common source sets
- No extra artifact for Kotlin/Native source sets

Additionally, you can use the `kotlin-test` dependency in any shared or platform-specific source set.

An existing kotlin-test setup with explicit dependencies will continue to work both in Gradle and in Maven.

Learn more about setting dependencies on test libraries.

Automatic selection of a testing framework for Kotlin/JVM source sets

The Gradle plugin now chooses and adds a dependency on a testing framework automatically. All you need to do is add the dependency `kotlin-test` in the common source set.

Gradle uses JUnit 4 by default. Therefore, the `kotlin("test")` dependency resolves to the variant for JUnit 4, namely `kotlin-test-junit`:

Kotlin

```
kotlin {
    sourceSets {
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test")) // This brings the dependency
                                               // on JUnit 4 transitively
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // This brings the dependency
                                              // on JUnit 4 transitively
            }
        }
    }
}
```

You can choose JUnit 5 or TestNG by calling `useJUnitPlatform()` or `useTestNG()` in the test task:

```
tasks {
    test {
        // enable TestNG support
        useTestNG()
        // or
        // enable JUnit Platform (a.k.a. JUnit 5) support
        useJUnitPlatform()
    }
}
```

You can disable automatic testing framework selection by adding the line `kotlin.test.infer.jvm.variant=false` to the project's `gradle.properties`.

Learn more about setting dependencies on test libraries.

Assertion function updates

This release brings new assertion functions and improves the existing ones.

The `kotlin-test` library now has the following features:

- **Checking the type of a value**

  You can use the new `assertIs<T>` and `assertIsNot<T>` to check the type of a value:

```
@Test
fun testFunction() {
    val s: Any = "test"
    assertIs<String>(s)  // throws AssertionError mentioning the actual type of s if the assertion
fails
    // can now print s.length because of contract in assertIs
    println("${s.length}")
}
```

Because of type erasure, this assert function only checks whether the `value` is of the `List` type in the following example and doesn't check whether it's a list of the particular `String` element type: `assertIs<List<String>>(value)`.

- **Comparing the container content for arrays, sequences, and arbitrary iterables**

  There is a new set of overloaded `assertContentEquals()` functions for comparing content for different collections that don't implement structural equality:

```
@Test
fun test() {
    val expectedArray = arrayOf(1, 2, 3)
    val actualArray = Array(3) { it + 1 }
    assertContentEquals(expectedArray, actualArray)
}
```

- **New overloads to `assertEquals()` and `assertNotEquals()` for `Double` and `Float` numbers**

  There are new overloads for the `assertEquals()` function that make it possible to compare two `Double` or `Float` numbers with absolute precision. The precision value is specified as the third parameter of the function:

```
@Test
fun test() {
    val x = sin(PI)

    // precision parameter
    val tolerance = 0.000001

    assertEquals(0.0, x, tolerance)
}
```

- **New functions for checking the content of collections and elements**

  You can now check whether the collection or element contains something with the `assertContains()` function. You can use it with Kotlin collections and elements that have the `contains()` operator, such as `IntRange`, `String`, and others:

```
@Test
fun test() {
    val sampleList = listOf<String>("sample", "sample2")
    val sampleString = "sample"
    assertContains(sampleList, sampleString)  // element in collection
    assertContains(sampleString, "amp")       // substring in string
}
```

- **`assertTrue()`, `assertFalse()`, `expect()` functions are now inline**

  From now on, you can use these as inline functions, so it's possible to call suspend functions inside a lambda expression:

```
@Test
fun test() = runBlocking<Unit> {
    val deferred = async { "Kotlin is nice" }
    assertTrue("Kotlin substring should be present") {
        deferred.await() .contains("Kotlin")
    }
}
```

kotlinx libraries

Along with Kotlin 1.5.0, we are releasing new versions of the kotlinx libraries:

- `kotlinx.coroutines` 1.5.0-RC
- `kotlinx.serialization` 1.2.1
- `kotlinx-datetime` 0.2.0

Coroutines 1.5.0-RC

`kotlinx.coroutines` 1.5.0-RC is here with:

- New channels API
- Stable reactive integrations
- And more

Starting with Kotlin 1.5.0, experimental coroutines are disabled and the `-Xcoroutines=experimental` flag is no longer supported.

Learn more in the changelog and the `kotlinx.coroutines` 1.5.0 release blog post.

Watch video online.

Serialization 1.2.1

`kotlinx.serialization` 1.2.1 is here with:

- Improvements to JSON serialization performance
- Support for multiple names in JSON serialization
- Experimental .proto schema generation from `@Serializable` classes
- And more

Learn more in the changelog and the `kotlinx.serialization` 1.2.1 release blog post.

Watch video online.

dateTime 0.2.0

`kotlinx-datetime` 0.2.0 is here with:

- `@Serializable` Datetime objects
- Normalized API of `DateTimePeriod` and `DatePeriod`

- And more

Learn more in the changelog and the `kotlinx-datetime` 0.2.0 release blog post.

Migrating to Kotlin 1.5.0

IntelliJ IDEA and Android Studio will suggest updating the Kotlin plugin to 1.5.0 once it is available.

To migrate existing projects to Kotlin 1.5.0, just change the Kotlin version to `1.5.0` and re-import your Gradle or Maven project. Learn how to update to Kotlin 1.5.0.

To start a new project with Kotlin 1.5.0, update the Kotlin plugin and run the Project Wizard from **File** | **New** | **Project**.

The new command-line compiler is available for downloading on the GitHub release page.

Kotlin 1.5.0 is a feature release and therefore can bring incompatible changes to the language. Find the detailed list of such changes in the Compatibility Guide for Kotlin 1.5.

Compatibility guide for Kotlin 1.5.x

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.4 to Kotlin 1.5.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language and stdlib

Forbid spread operator in signature-polymorphic calls

Issue: KT-35226

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw the use of spread operator (*) on signature-polymorphic calls

Deprecation cycle:

- < 1.5: introduce warning for the problematic operator at call-site
- >= 1.5: raise this warning to an error, `-XXLanguage:-ProhibitSpreadOnSignaturePolymorphicCall` can be used to temporarily revert to pre-1.5 behavior

Forbid non-abstract classes containing abstract members invisible from that classes (internal/package-private)

Issue: KT-27825

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw non-abstract classes containing abstract members invisible from that classes (internal/package-private)

Deprecation cycle:

- < 1.5: introduce warning for the problematic classes
- >= 1.5: raise this warning to an error, `-XXLanguage:-ProhibitInvisibleAbstractMethodsInSuperclasses` can be used to temporarily revert to pre-1.5 behavior

Forbid using array based on non-reified type parameters as reified type arguments on JVM

Issue: KT-31227

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw using array based on non-reified type parameters as reified type arguments on JVM

Deprecation cycle:

- < 1.5: introduce warning for the problematic calls
- >= 1.5: raise this warning to an error, `-XXLanguage:-ProhibitNonReifiedArraysAsReifiedTypeArguments` can be used to temporarily revert to pre-1.5 behavior

Forbid secondary enum class constructors which do not delegate to the primary constructor

Issue: KT-35870

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw secondary enum class constructors which do not delegate to the primary constructor

Deprecation cycle:

- < 1.5: introduce warning for the problematic constructors
- >= 1.5: raise this warning to an error, `-XXLanguage:-RequiredPrimaryConstructorDelegationCallInEnums` can be used to temporarily revert to pre-1.5 behavior

Forbid exposing anonymous types from private inline functions

Issue: KT-33917

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw exposing anonymous types from private inline functions

Deprecation cycle:

- < 1.5: introduce warning for the problematic constructors
- >= 1.5: raise this warning to an error, `-XXLanguage:-ApproximateAnonymousReturnTypesInPrivateInlineFunctions` can be used to temporarily revert to pre-1.5 behavior

Forbid passing non-spread arrays after arguments with SAM-conversion

Issue: KT-35224

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw passing non-spread arrays after arguments with SAM-conversion

Deprecation cycle:

- 1.3.70: introduce warning for the problematic calls
- >= 1.5: raise this warning to an error, `-XXLanguage:-ProhibitVarargAsArrayAfterSamArgument` can be used to temporarily revert to pre-1.5 behavior

Support special semantics for underscore-named catch block parameters

Issue: KT-31567

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw references to the underscore symbol (`_`) that is used to omit parameter name of an exception in the catch block

Deprecation cycle:

- 1.4.20: introduce warning for the problematic references
- >= 1.5: raise this warning to an error, `-XXLanguage:-ForbidReferencingToUnderscoreNamedParameterOfCatchBlock` can be used to temporarily revert to pre-1.5 behavior

Change implementation strategy of SAM conversion from anonymous class-based to invokedynamic

Issue: KT-44912

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since Kotlin 1.5, implementation strategy of SAM (single abstract method) conversion will be changed from generating an anonymous class to using the `invokedynamic` JVM instruction

Deprecation cycle:

- 1.5: change implementation strategy of SAM conversion, `-Xsam-conversions=class` can be used to revert implementation scheme to the one that used before

Performance issues with the JVM IR-based backend

Issue: KT-48233

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin 1.5 uses the IR-based backend for the Kotlin/JVM compiler by default. The old backend is still used by default for earlier language versions.

You might encounter some performance degradation issues using the new compiler in Kotlin 1.5. We are working on fixing such cases.

Deprecation cycle:

- < 1.5: by default, the old JVM backend is used
- >= 1.5: by default, the IR-based backend is used. If you need to use the old backend in Kotlin 1.5, add the following lines to the project's configuration file to temporarily revert to pre-1.5 behavior:

In Gradle:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile> {
  kotlinOptions.useOldBackend = true
}
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {
  kotlinOptions.useOldBackend = true
}
```

In Maven:

```
<configuration>
    <args>
        <arg>-Xuse-old-backend</arg>
    </args>
</configuration>
```

Support for this flag will be removed in one of the future releases.

New field sorting in the JVM IR-based backend

Issue: KT-46378

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since version 1.5, Kotlin uses the IR-based backend that sorts JVM bytecode differently: it generates fields declared in the constructor before fields declared in the body, while it's vice versa for the old backend. The new sorting may change the behavior of programs that use serialization frameworks that depend on the field order, such as Java serialization.

Deprecation cycle:

- < 1.5: by default, the old JVM backend is used. It has fields declared in the body before fields declared in the constructor.
- >= 1.5: by default, the new IR-based backend is used. Fields declared in the constructor are generated before fields declared in the body. As a workaround, you can temporarily switch to the old backend in Kotlin 1.5. To do that, add the following lines to the project's configuration file:

In Gradle:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile> {
  kotlinOptions.useOldBackend = true
}
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {
  kotlinOptions.useOldBackend = true
}
```

In Maven:

```
<configuration>
    <args>
        <arg>-Xuse-old-backend</arg>
    </args>
</configuration>
```

Support for this flag will be removed in one of the future releases.

Generate nullability assertion for delegated properties with a generic call in the delegate expression

Issue: KT-44304

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since Kotlin 1.5, the Kotlin compiler will emit nullability assertions for delegated properties with a generic call in the delegate expression

Deprecation cycle:

- 1.5: emit nullability assertion for delegated properties (see details in the issue), `-Xuse-old-backend` or `-language-version 1.4` can be used to temporarily revert to pre-1.5 behavior

Turn warnings into errors for calls with type parameters annotated by @OnlyInputTypes

Issue: KT-45861

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw calls like `contains`, `indexOf`, and `assertEquals` with senseless arguments to improve type safety

Deprecation cycle:

- 1.4.0: introduce warning for the problematic constructors
- >= 1.5: raise this warning to an error, `-XXLanguage:-StrictOnlyInputTypesChecks` can be used to temporarily revert to pre-1.5 behavior

Use the correct order of arguments execution in calls with named vararg

Issue: KT-17691

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin 1.5 will change the order of arguments execution in calls with named vararg

Deprecation cycle:

- < 1.5: introduce warning for the problematic constructors
- >= 1.5: raise this warning to an error, `-XXLanguage:-UseCorrectExecutionOrderForVarargArguments` can be used to temporarily revert to pre-1.5 behavior

Use default value of the parameter in operator functional calls

Issue: KT-42064

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin 1.5 will use default value of the parameter in operator calls

Deprecation cycle:

- < 1.5: old behavior (see details in the issue)
- >= 1.5: behavior changed, `-XXLanguage:-JvmIrEnabledByDefault` can be used to temporarily revert to pre-1.5 behavior

Produce empty reversed progressions in for loops if regular progression is also empty

Issue: KT-42533

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin 1.5 will produce empty reversed progressions in for loops if regular progression is also empty

Deprecation cycle:

- < 1.5: old behavior (see details in the issue)
- >= 1.5: behavior changed, `-XXLanguage:-JvmIrEnabledByDefault` can be used to temporarily revert to pre-1.5 behavior

Straighten Char-to-code and Char-to-digit conversions out

Issue: KT-23451

Component: kotlin-stdlib

Incompatible change type: source

Short summary: Since Kotlin 1.5, conversions of Char to number types will be deprecated

Deprecation cycle:

- 1.5: deprecate `Char.toInt()/toShort()/toLong()/toByte()/toDouble()/toFloat()` and the reverse functions like `Long.toChar()`, and propose replacement

Inconsistent case-insensitive comparison of characters in kotlin.text functions

Issue: KT-45496

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: Since Kotlin 1.5, `Char.equals` will be improved in case-insensitive case by first comparing whether the uppercase variants of characters are equal, then whether the lowercase variants of those uppercase variants (as opposed to the characters themselves) are equal

Deprecation cycle:

- < 1.5: old behavior (see details in the issue)
- 1.5: change behavior for `Char.equals` function

Remove default locale-sensitive case conversion API

Issue: KT-43023

Component: kotlin-stdlib

Incompatible change type: source

Short summary: Since Kotlin 1.5, default locale-sensitive case conversion functions like `String.toUpperCase()` will be deprecated

Deprecation cycle:

- 1.5: deprecate case conversions functions with the default locale (see details in the issue), and propose replacement

Gradually change the return type of collection min and max functions to non-nullable

Issue: KT-38854

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: return type of collection `min` and `max` functions will be changed to non-nullable in 1.6

Deprecation cycle:

- 1.4: introduce `...OrNull` functions as synonyms and deprecate the affected API (see details in the issue)
- 1.5.0: raise the deprecation level of the affected API to error
- >=1.6: reintroduce the affected API but with non-nullable return type

Raise the deprecation level of conversions of floating-point types to Short and Byte

Issue: KT-30360

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: conversions of floating-point types to `Short` and `Byte` deprecated in Kotlin 1.4 with `WARNING` level will cause errors since Kotlin 1.5.0.

Deprecation cycle:

- 1.4: deprecate `Double.toShort()/toByte()` and `Float.toShort()/toByte()` and propose replacement
- 1.5.0: raise the deprecation level to error

Tools

Do not mix several JVM variants of kotlin-test in a single project

Issue: KT-40225

Component: Gradle

Incompatible change type: behavioral

Short summary: several mutually exclusive `kotlin-test` variants for different testing frameworks could have been in a project if one of them is brought by a transitive dependency. From 1.5.0, Gradle won't allow having mutually exclusive `kotlin-test` variants for different testing frameworks.

Deprecation cycle:

- < 1.5: having several mutually exclusive `kotlin-test` variants for different testing frameworks is allowed
- >= 1.5: behavior changed,
  Gradle throws an exception like "Cannot select module with conflict on capability...". Possible solutions:
  - use the same `kotlin-test` variant and the corresponding testing framework as the transitive dependency brings.
  - find another variant of the dependency that doesn't bring the `kotlin-test` variant transitively, so you can use the testing framework you would like to use.
  - find another variant of the dependency that brings another `kotlin-test` variant transitively, which uses the same testing framework you would like to use.
  - exclude the testing framework that is brought transitively. The following example is for excluding JUnit 4:

    ```
    configurations {
        testImplementation.get().exclude("org.jetbrains.kotlin", "kotlin-test-junit")
    }
    ```

    After excluding the testing framework, test your application. If it stopped working, rollback excluding changes, use the same testing framework as the library does, and exclude your testing framework.

What's new in Kotlin 1.4.30

Released: 3 February 2021

Kotlin 1.4.30 offers preview versions of new language features, promotes the new IR backend of the Kotlin/JVM compiler to Beta, and ships various performance and functional improvements.

You can also learn about new features in this blog post.

For information about the Kotlin release cycle, see Kotlin release process.

Language features

Kotlin 1.5.0 is going to deliver new language features – JVM records support, sealed interfaces, and Stable inline classes. In Kotlin 1.4.30, you can try these features and improvements in preview mode. We would be very grateful if you share your feedback with us in the corresponding YouTrack tickets, as that will allow us to address it before the release of 1.5.0.

- JVM records support
- Sealed interfaces and sealed class improvements
- Improved inline classes

To enable these language features and improvements in preview mode, you need to opt in by adding specific compiler options. See the sections below for details.

Learn more about the new features preview in this blog post.

JVM records support

> The JVM records feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

The JDK 16 release includes plans to stabilize a new Java class type called record. To provide all the benefits of Kotlin and maintain its interoperability with Java, Kotlin is introducing experimental record class support.

You can use record classes that are declared in Java just like classes with properties in Kotlin. No additional steps are required.

Starting with 1.4.30, you can declare the record class in Kotlin using the `@JvmRecord` annotation for a data class:

```
@JvmRecord
data class User(val name: String, val age: Int)
```

To try the preview version of JVM records, add the compiler options `-Xjvm-enable-preview` and `-language-version 1.5`.

We're continuing to work on JVM records support, and we would be very grateful if you would share your feedback with us using this YouTrack ticket.

Learn more about implementation, restrictions, and the syntax in KEEP.

Sealed interfaces

> Sealed interfaces are Experimental. They may be dropped or changed at any time. Opt-in is required (see the details below), and you should use them only for evaluation purposes. We would appreciate your feedback on them in YouTrack.

In Kotlin 1.4.30, we're shipping the prototype of *sealed interfaces*. They complement sealed classes and make it possible to build more flexible restricted class hierarchies.

They can serve as "internal" interfaces that cannot be implemented outside the same module. You can rely on that fact, for example, to write exhaustive `when` expressions.

```
sealed interface Polygon

class Rectangle(): Polygon
class Triangle(): Polygon

// when() is exhaustive: no other polygon implementations can appear
// after the module is compiled
fun draw(polygon: Polygon) = when (polygon) {
    is Rectangle -> // ...
    is Triangle -> // ...
}
```

Another use-case: with sealed interfaces, you can inherit a class from two or more sealed superclasses.

```
sealed interface Fillable {
   fun fill()
}
sealed interface Polygon {
   val vertices: List<Point>
}

class Rectangle(override val vertices: List<Point>): Fillable, Polygon {
   override fun fill() { /*...*/ }
}
```

To try the preview version of sealed interfaces, add the compiler option `-language-version 1.5`. Once you switch to this version, you'll be able to use the `sealed` modifier on interfaces. We would be very grateful if you would share your feedback with us using this YouTrack ticket.

Learn more about sealed interfaces.

Package-wide sealed class hierarchies

Package-wide hierarchies of sealed classes are Experimental. They may be dropped or changed at any time. Opt-in is required (see the details below), and you should use them only for evaluation purposes. We would appreciate your feedback on them in YouTrack.

Sealed classes can now form more flexible hierarchies. They can have subclasses in all files of the same compilation unit and the same package. Previously, all subclasses had to appear in the same file.

Direct subclasses may be top-level or nested inside any number of other named classes, named interfaces, or named objects. The subclasses of a sealed class must have a name that is properly qualified – they cannot be local nor anonymous objects.

To try package-wide hierarchies of sealed classes, add the compiler option `-language-version 1.5`. We would be very grateful if you would share your feedback with us using this YouTrack ticket.

Learn more about package-wide hierarchies of sealed classes.

Improved inline classes

Inline value classes are in Beta. They are almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you have to make. We would appreciate your feedback on the inline classes feature in YouTrack.

Kotlin 1.4.30 promotes inline classes to Beta and brings the following features and improvements to them:

- Since inline classes are value-based, you can define them using the `value` modifier. The `inline` and `value` modifiers are now equivalent to each other. In future Kotlin versions, we're planning to deprecate the `inline` modifier.

  From now on, Kotlin requires the `@JvmInline` annotation before a class declaration for the JVM backend:

  ```
  inline class Name(private val s: String)

  value class Name(private val s: String)

  // For JVM backends
  @JvmInline
  value class Name(private val s: String)
  ```

- Inline classes can have `init` blocks. You can add code to be executed right after the class is instantiated:

  ```
  @JvmInline
  value class Negative(val x: Int) {
    init {
        require(x < 0) { }
    }
  }
  ```

- Calling functions with inline classes from Java code: before Kotlin 1.4.30, you couldn't call functions that accept inline classes from Java because of mangling. From now on, you can disable mangling manually. To call such functions from Java code, you should add the `@JvmName` annotation before the function declaration:

  ```
  inline class UInt(val x: Int)

  fun compute(x: Int) { }

  @JvmName("computeUInt")
  fun compute(x: UInt) { }
  ```

- In this release, we've changed the mangling scheme for functions to fix the incorrect behavior. These changes led to ABI changes.

  Starting with 1.4.30, the Kotlin compiler uses a new mangling scheme by default. Use the `-Xuse-14-inline-classes-mangling-scheme` compiler flag to force the compiler to use the old 1.4.0 mangling scheme and preserve binary compatibility.

Kotlin 1.4.30 promotes inline classes to Beta and we are planning to make them Stable in future releases. We'd be very grateful if you would share your feedback with us using this YouTrack ticket.

To try the preview version of inline classes, add the compiler option `-Xinline-classes` or `-language-version 1.5`.

Learn more about the mangling algorithm in KEEP.

Learn more about inline classes.

Kotlin/JVM

JVM IR compiler backend reaches Beta

The IR-based compiler backend for Kotlin/JVM, which was presented in 1.4.0 in Alpha, has reached Beta. This is the last pre-stable level before the IR backend becomes the default for the Kotlin/JVM compiler.

We're now dropping the restriction on consuming binaries produced by the IR compiler. Previously, you could use code compiled by the new JVM IR backend only if you had enabled the new backend. Starting from 1.4.30, there is no such limitation, so you can use the new backend to build components for third-party use, such as libraries. Try the Beta version of the new backend and share your feedback in our issue tracker.

To enable the new JVM IR backend, add the following lines to the project's configuration file:

- In Gradle:

Kotlin

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile::class) {
  kotlinOptions.useIR = true
}
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {
  kotlinOptions.useIR = true
}
```

- In Maven:

```
<configuration>
    <args>
        <arg>-Xuse-ir</arg>
    </args>
</configuration>
```

Learn more about the changes that the JVM IR backend brings in this blog post.

Kotlin/Native

Performance improvements

Kotlin/Native has received a variety of performance improvements in 1.4.30, which has resulted in faster compilation times. For example, the time required to rebuild the framework in the Networking and data storage with Kotlin Multiplatform Mobile sample has decreased from 9.5 seconds (in 1.4.10) to 4.5 seconds (in 1.4.30).

Apple watchOS 64-bit simulator target

The x86 simulator target has been deprecated for watchOS since version 7.0. To keep up with the latest watchOS versions, Kotlin/Native has the new target `watchosX64` for running the simulator on 64-bit architecture.

Support for Xcode 12.2 libraries

We have added support for the new libraries delivered with Xcode 12.2. You can now use them from Kotlin code.

Kotlin/JS

Lazy initialization of top-level properties

> Lazy initialization of top-level properties is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

The IR backend for Kotlin/JS is receiving a prototype implementation of lazy initialization for top-level properties. This reduces the need to initialize all top-level properties when the application starts, and it should significantly improve application start-up times.

We'll keep working on the lazy initialization, and we ask you to try the current prototype and share your thoughts and results in this YouTrack ticket or the `#javascript` channel in the official Kotlin Slack (get an invite here).

To use the lazy initialization, add the `-Xir-property-lazy-initialization` compiler option when compiling the code with the JS IR compiler.

Gradle project improvements

Support the Gradle configuration cache

Starting with 1.4.30, the Kotlin Gradle plugin supports the configuration cache feature. It speeds up the build process: once you run the command, Gradle executes the configuration phase and calculates the task graph. Gradle caches the result and reuses it for subsequent builds.

To start using this feature, you can use the Gradle command or set up the IntelliJ based IDE.

Standard library

Locale-agnostic API for upper/lowercasing text

> The locale-agnostic API feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

This release introduces the experimental locale-agnostic API for changing the case of strings and characters. The current `toLowerCase()`, `toUpperCase()`, `capitalize()`, `decapitalize()` API functions are locale-sensitive. This means that different platform locale settings can affect code behavior. For example, in the Turkish locale, when the string "kotlin" is converted using `toUpperCase`, the result is "KOTLİN", not "KOTLIN".

```
// current API
println("Needs to be capitalized".toUpperCase()) // NEEDS TO BE CAPITALIZED

// new API
println("Needs to be capitalized".uppercase()) // NEEDS TO BE CAPITALIZED
```

Kotlin 1.4.30 provides the following alternatives:

- For `String` functions:

Earlier versions	1.4.30 alternative
`String.toUpperCase()`	`String.uppercase()`
`String.toLowerCase()`	`String.lowercase()`
`String.capitalize()`	`String.replaceFirstChar { it.uppercase() }`
`String.decapitalize()`	`String.replaceFirstChar { it.lowercase() }`

- For `Char` functions:

Earlier versions	1.4.30 alternative
`Char.toUpperCase()`	`Char.uppercaseChar(): Char` `Char.uppercase(): String`
`Char.toLowerCase()`	`Char.lowercaseChar(): Char` `Char.lowercase(): String`
`Char.toTitleCase()`	`Char.titlecaseChar(): Char` `Char.titlecase(): String`

For Kotlin/JVM, there are also overloaded `uppercase()`, `lowercase()`, and `titlecase()` functions with an explicit `Locale` parameter.

See the full list of changes to the text processing functions in KEEP.

Clear Char-to-code and Char-to-digit conversions

The unambiguous API for the `Char` conversion feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.

The current `Char` to numbers conversion functions, which return UTF-16 codes expressed in different numeric types, are often confused with the similar String-to-Int conversion, which returns the numeric value of a string:

```
"4".toInt() // returns 4
'4'.toInt() // returns 52
// and there was no common function that would return the numeric value 4 for Char '4'
```

To avoid this confusion we've decided to separate `Char` conversions into two following sets of clearly named functions:

- Functions to get the integer code of `Char` and to construct `Char` from the given code:

```
fun Char(code: Int): Char
fun Char(code: UShort): Char
val Char.code: Int
```

- Functions to convert `Char` to the numeric value of the digit it represents:

```
fun Char.digitToInt(radix: Int): Int
fun Char.digitToIntOrNull(radix: Int): Int?
```

- An extension function for `Int` to convert the non-negative single digit it represents to the corresponding `Char` representation:

```
fun Int.digitToChar(radix: Int): Char
```

See more details in KEEP.

Serialization updates

Along with Kotlin 1.4.30, we are releasing `kotlinx.serialization` 1.1.0-RC, which includes some new features:

- Inline classes serialization support
- Unsigned primitive type serialization support

Inline classes serialization support

Starting with Kotlin 1.4.30, you can make inline classes serializable:

```
@Serializable
inline class Color(val rgb: Int)
```

The feature requires the new 1.4.30 IR compiler.

The serialization framework does not box serializable inline classes when they are used in other serializable classes.

Learn more in the `kotlinx.serialization` docs.

Unsigned primitive type serialization support

Starting from 1.4.30, you can use standard JSON serializers of kotlinx.serialization for unsigned primitive types: `UInt`, `ULong`, `UByte`, and `UShort`:

```
@Serializable
class Counter(val counted: UByte, val description: String)
fun main() {
    val counted = 239.toUByte()
    println(Json.encodeToString(Counter(counted, "tries")))
}
```

Learn more in the `kotlinx.serialization` docs.

What's new in Kotlin 1.4.20

Released: 23 November 2020

Kotlin 1.4.20 offers a number of new experimental features and provides fixes and improvements for existing features, including those added in 1.4.0.

You can also learn about new features with more examples in this blog post.

For information about the Kotlin release cycle, see Kotlin release process.

Kotlin/JVM

Improvements of Kotlin/JVM are intended to keep it up with the features of modern Java versions:

- Java 15 target
- invokedynamic string concatenation

Java 15 target

Now Java 15 is available as a Kotlin/JVM target.

invokedynamic string concatenation

`invokedynamic` string concatenation is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin 1.4.20 can compile string concatenations into dynamic invocations on JVM 9+ targets, therefore improving the performance.

Currently, this feature is experimental and covers the following cases:

- `String.plus` in the operator (`a + b`), explicit (`a.plus(b)`), and reference (`(a::plus)(b)`) form.
- `toString` on inline and data classes.
- string templates except for ones with a single non-constant argument (see KT-42457).

To enable `invokedynamic` string concatenation, add the `-Xstring-concat` compiler option with one of the following values:

- `indy-with-constants` to perform `invokedynamic` concatenation on strings with StringConcatFactory.makeConcatWithConstants().
- `indy` to perform `invokedynamic` concatenation on strings with StringConcatFactory.makeConcat().
- `inline` to switch back to the classic concatenation via `StringBuilder.append()`.

Kotlin/JS

Kotlin/JS keeps evolving fast, and in 1.4.20 you can find a number experimental features and improvements:

- Gradle DSL changes
- New Wizard templates
- Ignoring compilation errors with IR compiler

Gradle DSL changes

The Gradle DSL for Kotlin/JS receives a number of updates which simplify project setup and customization. This includes webpack configuration adjustments, modifications to the auto-generated `package.json` file, and improved control over transitive dependencies.

Single point for webpack configuration

A new configuration block `commonWebpackConfig` is available for the browser target. Inside it, you can adjust common settings from a single point, instead of having to duplicate configurations for the `webpackTask`, `runTask`, and `testTask`.

To enable CSS support by default for all three tasks, add the following snippet in the `build.gradle(.kts)` of your project:

```
browser {
    commonWebpackConfig {
        cssSupport.enabled = true
    }
    binaries.executable()
}
```

Learn more about configuring webpack bundling.

package.json customization from Gradle

For more control over your Kotlin/JS package management and distribution, you can now add properties to the project file `package.json` via the Gradle DSL.

To add custom fields to your `package.json`, use the `customField` function in the compilation's `packageJson` block:

```
kotlin {
    js(BOTH) {
        compilations["main"].packageJson {
            customField("hello", mapOf("one" to 1, "two" to 2))
        }
    }
}
```

Learn more about `package.json` customization.

Selective yarn dependency resolutions

> Support for selective yarn dependency resolutions is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin 1.4.20 provides a way of configuring Yarn's selective dependency resolutions - the mechanism for overriding dependencies of the packages you depend on.

You can use it through the `YarnRootExtension` inside the `YarnPlugin` in Gradle. To affect the resolved version of a package for your project, use the `resolution` function passing in the package name selector (as specified by Yarn) and the version to which it should resolve.

```
rootProject.plugins.withType<YarnPlugin> {
    rootProject.the<YarnRootExtension>().apply {
        resolution("react", "16.0.0")
        resolution("processor/decamelize", "3.0.0")
    }
}
```

Here, *all* of your npm dependencies which require `react` will receive version `16.0.0`, and `processor` will receive its dependency `decamelize` as version `3.0.0`.

Disabling granular workspaces

> Disabling granular workspaces is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

To speed up build times, the Kotlin/JS Gradle plugin only installs the dependencies which are required for a particular Gradle task. For example, the `webpack-dev-server` package is only installed when you execute one of the `*Run` tasks, and not when you execute the assemble task. Such behavior can potentially bring problems when you run multiple Gradle processes in parallel. When the dependency requirements clash, the two installations of npm packages can cause errors.

To resolve this issue, Kotlin 1.4.20 includes an option to disable these so-called *granular workspaces*. This feature is currently available through the `YarnRootExtension` inside the `YarnPlugin` in Gradle. To use it, add the following snippet to your `build.gradle.kts` file:

```
rootProject.plugins.withType<YarnPlugin> {
    rootProject.the<YarnRootExtension>().disableGranularWorkspaces()
}
```

New Wizard templates

To give you more convenient ways to customize your project during creation, the project wizard for Kotlin comes with new templates for Kotlin/JS applications:

- **Browser Application** - a minimal Kotlin/JS Gradle project that runs in the browser.

- **React Application** - a React app that uses the appropriate `kotlin-wrappers`. It provides options to enable integrations for style-sheets, navigational components, or state containers.
- **Node.js Application** - a minimal project for running in a Node.js runtime. It comes with the option to directly include the experimental `kotlinx-nodejs` package.

Ignoring compilation errors with IR compiler

> *Ignore compilation errors* mode is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

The IR compiler for Kotlin/JS comes with a new experimental mode - *compilation with errors*. In this mode, you can run you code even if it contains errors, for example, if you want to try certain things it when the whole application is not ready yet.

There are two tolerance policies for this mode:

- `SEMANTIC`: the compiler will accept code which is syntactically correct, but doesn't make sense semantically, such as `val x: String = 3`.
- `SYNTAX`: the compiler will accept any code, even if it contains syntax errors.

To allow compilation with errors, add the `-Xerror-tolerance-policy=` compiler option with one of the values listed above.

Learn more about Kotlin/JS IR compiler.

Kotlin/Native

Kotlin/Native's priorities in 1.4.20 are performance and polishing existing features. These are the notable improvements:

- Escape analysis
- Performance improvements and bug fixes
- Opt-in wrapping of Objective-C exceptions
- CocoaPods plugin improvements
- Support for Xcode 12 libraries

Escape analysis

> The escape analysis mechanism is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin/Native receives a prototype of the new escape analysis mechanism. It improves the runtime performance by allocating certain objects on the stack instead of the heap. This mechanism shows a 10% average performance increase on our benchmarks, and we continue improving it so that it speeds up the program even more.

The escape analysis runs in a separate compilation phase for the release builds (with the `-opt` compiler option).

If you want to disable the escape analysis phase, use the `-Xdisable-phases=EscapeAnalysis` compiler option.

Performance improvements and bug fixes

Kotlin/Native receives performance improvements and bug fixes in various components, including the ones added in 1.4.0, for example, the code sharing mechanism.

Opt-in wrapping of Objective-C exceptions

> The Objective-C exception wrapping mechanism is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin/Native now can handle exceptions thrown from Objective-C code in runtime to avoid program crashes.

You can opt in to wrap `NSException`'s into Kotlin exceptions of type `ForeignException`. They hold the references to the original `NSException`'s. This lets you get the information about the root cause and handle it properly.

To enable wrapping of Objective-C exceptions, specify the `-Xforeign-exception-mode objc-wrap` option in the `cinterop` call or add `foreignExceptionMode = objc-wrap` property to `.def` file. If you use CocoaPods integration, specify the option in the `pod {}` build script block of a dependency like this:

```
pod("foo") {
    extraOpts = listOf("-Xforeign-exception-mode", "objc-wrap")
}
```

The default behavior remains unchanged: the program terminates when an exception is thrown from the Objective-C code.

CocoaPods plugin improvements

Kotlin 1.4.20 continues the set of improvements in CocoaPods integration. Namely, you can try the following new features:

- Improved task execution
- Extended DSL
- Updated integration with Xcode

Improved task execution

CocoaPods plugin gets an improved task execution flow. For example, if you add a new CocoaPods dependency, existing dependencies are not rebuilt. Adding an extra target also doesn't affect rebuilding dependencies for existing ones.

Extended DSL

The DSL of adding CocoaPods dependencies to your Kotlin project receives new capabilites.

In addition to local Pods and Pods from the CocoaPods repository, you can add dependencies on the following types of libraries:

- A library from a custom spec repository.
- A remote library from a Git repository.
- A library from an archive (also available by arbitrary HTTP address).
- A static library.
- A library with custom cinterop options.

Learn more about adding CocoaPods dependencies in Kotlin projects. Find examples in the Kotlin with CocoaPods sample.

Updated integration with Xcode

To work correctly with Xcode, Kotlin requires some Podfile changes:

- If your Kotlin Pod has any Git, HTTP, or specRepo Pod dependency, you should also specify it in the Podfile.
- When you add a library from the custom spec, you also should specify the location of specs at the beginning of your Podfile.

Now integration errors have a detailed description in IDEA. So if you have problems with your Podfile, you will immediately know how to fix them.

Learn more about creating Kotlin pods.

Support for Xcode 12 libraries

We have added support for new libraries delivered with Xcode 12. Now you can use them from the Kotlin code.

Kotlin Multiplatform

Updated structure of multiplatform library publications

Starting from Kotlin 1.4.20, there is no longer a separate metadata publication. Metadata artifacts are now included in the *root* publication which stands for the whole library and is automatically resolved to the appropriate platform-specific artifacts when added as a dependency to the common source set.

Learn more about publishing a multiplatform library.

Compatibility with earlier versions

This change of structure breaks the compatibility between projects with hierarchical project structure. If a multiplatform project and a library it depends on both have the hierarchical project structure, then you need to update them to Kotlin 1.4.20 or higher simultaneously. Libraries published with Kotlin 1.4.20 are not available for using from project published with earlier versions.

Projects and libraries without the hierarchical project structure remain compatible.

Standard library

The standard library of Kotlin 1.4.20 offers new extensions for working with files and a better performance.

- Extensions for java.nio.file.Path

- Improved String.replace function performance

Extensions for java.nio.file.Path

> Extensions for `java.nio.file.Path` are Experimental. They may be dropped or changed at any time. Opt-in is required (see details below). Use them only for evaluation purposes. We appreciate your feedback on them in YouTrack.

Now the standard library provides experimental extensions for `java.nio.file.Path`. Working with the modern JVM file API in an idiomatic Kotlin way is now similar to working with `java.io.File` extensions from the `kotlin.io` package.

```
// construct path with the div (/) operator
val baseDir = Path("/base")
val subDir = baseDir / "subdirectory"

// list files in a directory
val kotlinFiles: List<Path> = Path("/home/user").listDirectoryEntries("*.kt")
```

The extensions are available in the `kotlin.io.path` package in the `kotlin-stdlib-jdk7` module. To use the extensions, opt-in to the experimental annotation `@ExperimentalPathApi`.

Improved String.replace function performance

The new implementation of `String.replace()` speeds up the function execution. The case-sensitive variant uses a manual replacement loop based on `indexOf`, while the case-insensitive one uses regular expression matching.

Kotlin Android Extensions

In 1.4.20 the Kotlin Android Extensions plugin becomes deprecated and `Parcelable` implementation generator moves to a separate plugin.

- Deprecation of synthetic views
- New plugin for Parcelable implementation generator

Deprecation of synthetic views

Synthetic views were presented in the Kotlin Android Extensions plugin a while ago to simplify the interaction with UI elements and reduce boilerplate. Now Google offers a native mechanism that does the same - Android Jetpack's view bindings, and we're deprecating synthetic views in favor of those.

We extract the Parcelable implementations generator from `kotlin-android-extensions` and start the deprecation cycle for the rest of it - synthetic views. For now, they will keep working with a deprecation warning. In the future, you'll need to switch your project to another solution. Here are the guidelines that will help you migrate your Android project from synthetics to view bindings.

New plugin for Parcelable implementation generator

The `Parcelable` implementation generator is now available in the new `kotlin-parcelize` plugin. Apply this plugin instead of `kotlin-android-extensions`.

> `kotlin-parcelize` and `kotlin-android-extensions` can't be applied together in one module.

The `@Parcelize` annotation is moved to the `kotlinx.parcelize` package.

Learn more about `Parcelable` implementation generator in the Android documentation.

What's new in Kotlin 1.4.0

Released: 17 August 2020

In Kotlin 1.4.0, we ship a number of improvements in all of its components, with the focus on quality and performance. Below you will find the list of the most important changes in Kotlin 1.4.0.

> For information about the Kotlin release cycle, see Kotlin release process.

Language features and improvements

Kotlin 1.4.0 comes with a variety of different language features and improvements. They include:

- SAM conversions for Kotlin interfaces
- Explicit API mode for library authors
- Mixing named and positional arguments
- Trailing comma
- Callable reference improvements
- break and continue inside when included in loops

SAM conversions for Kotlin interfaces

Before Kotlin 1.4.0, you could apply SAM (Single Abstract Method) conversions only when working with Java methods and Java interfaces from Kotlin. From now on, you can use SAM conversions for Kotlin interfaces as well. To do so, mark a Kotlin interface explicitly as functional with the `fun` modifier.

SAM conversion applies if you pass a lambda as an argument when an interface with only one single abstract method is expected as a parameter. In this case, the compiler automatically converts the lambda to an instance of the class that implements the abstract member function.

```kotlin
fun interface IntPredicate {
    fun accept(i: Int): Boolean
}

val isEven = IntPredicate { it % 2 == 0 }

fun main() {
    println("Is 7 even? - ${isEven.accept(7)}")
}
```

Learn more about Kotlin functional interfaces and SAM conversions.

Explicit API mode for library authors

Kotlin compiler offers *explicit API mode* for library authors. In this mode, the compiler performs additional checks that help make the library's API clearer and more consistent. It adds the following requirements for declarations exposed to the library's public API:

- Visibility modifiers are required for declarations if the default visibility exposes them to the public API. This helps ensure that no declarations are exposed to the public API unintentionally.
- Explicit type specifications are required for properties and functions that are exposed to the public API. This guarantees that API users are aware of the types of API members they use.

Depending on your configuration, these explicit APIs can produce errors (*strict* mode) or warnings (*warning* mode). Certain kinds of declarations are excluded from such checks for the sake of readability and common sense:

- primary constructors
- properties of data classes
- property getters and setters
- `override` methods

Explicit API mode analyzes only the production sources of a module.

To compile your module in the explicit API mode, add the following lines to your Gradle build script:

Kotlin

```kotlin
kotlin {
    // for strict mode
    explicitApi()
    // or
    explicitApi = ExplicitApiMode.Strict

    // for warning mode
    explicitApiWarning()
    // or
    explicitApi = ExplicitApiMode.Warning
}
```

Groovy

```
kotlin {
    // for strict mode
    explicitApi()
    // or
    explicitApi = 'strict'

    // for warning mode
    explicitApiWarning()
    // or
    explicitApi = 'warning'
}
```

When using the command-line compiler, switch to explicit API mode by adding the `-Xexplicit-api` compiler option with the value `strict` or `warning`.

```
-Xexplicit-api={strict|warning}
```

Find more details about the explicit API mode in the KEEP.

Mixing named and positional arguments

In Kotlin 1.3, when you called a function with named arguments, you had to place all the arguments without names (positional arguments) before the first named argument. For example, you could call `f(1, y = 2)`, but you couldn't call `f(x = 1, 2)`.

It was really annoying when all the arguments were in their correct positions but you wanted to specify a name for one argument in the middle. It was especially helpful for making absolutely clear which attribute a boolean or `null` value belongs to.

In Kotlin 1.4, there is no such limitation – you can now specify a name for an argument in the middle of a set of positional arguments. Moreover, you can mix positional and named arguments any way you like, as long as they remain in the correct order.

```
fun reformat(
    str: String,
    uppercaseFirstLetter: Boolean = true,
    wordSeparator: Char = ' '
) {
    // ...
}

//Function call with a named argument in the middle
reformat("This is a String!", uppercaseFirstLetter = false , '-')
```

Trailing comma

With Kotlin 1.4 you can now add a trailing comma in enumerations such as argument and parameter lists, `when` entries, and components of destructuring declarations. With a trailing comma, you can add new items and change their order without adding or removing commas.

This is especially helpful if you use multi-line syntax for parameters or values. After adding a trailing comma, you can then easily swap lines with parameters or values.

```
fun reformat(
    str: String,
    uppercaseFirstLetter: Boolean = true,
    wordSeparator: Character = ' ', //trailing comma
) {
    // ...
}
```

```
val colors = listOf(
    "red",
    "green",
    "blue", //trailing comma
)
```

Callable reference improvements

Kotlin 1.4 supports more cases for using callable references:

- References to functions that include parameters with default values
- Function references in `Unit`-returning functions
- References that adapt based on the number of arguments in a function
- Suspend conversion on callable references

References to functions that include parameters with default values

Now you can use callable references to functions including parameters with default values. If the callable reference to the function `foo` takes no arguments, the default value `0` is used.

```
fun foo(i: Int = 0): String = "$i!"

fun apply(func: () -> String): String = func()

fun main() {
    println(apply(::foo))
}
```

Previously, you had to write an additional overload for either the `apply` or `foo` function.

```
// some new overload
fun applyInt(func: (Int) -> String): String = func(0)
```

Function references in Unit-returning functions

In Kotlin 1.4, you can use callable references to functions returning any type in `Unit`-returning functions. Before Kotlin 1.4, you could only use lambda arguments in this case. Now you can use both lambda arguments and callable references.

```
fun foo(f: () -> Unit) { }
fun returnsInt(): Int = 42

fun main() {
    foo { returnsInt() } // this was the only way to do it  before 1.4
    foo(::returnsInt) // starting from 1.4, this also works
}
```

References that adapt based on the number of arguments in a function

Now you can adapt callable references to functions when passing a variable number of arguments (`vararg`) . You can pass any number of parameters of the same type at the end of the list of passed arguments.

```
fun foo(x: Int, vararg y: String) {}

fun use0(f: (Int) -> Unit) {}
fun use1(f: (Int, String) -> Unit) {}
fun use2(f: (Int, String, String) -> Unit) {}

fun test() {
    use0(::foo)
    use1(::foo)
    use2(::foo)
}
```

Suspend conversion on callable references

In addition to suspend conversion on lambdas, Kotlin now supports suspend conversion on callable references starting from version 1.4.0.

```
fun call() {}
fun takeSuspend(f: suspend () -> Unit) {}

fun test() {
    takeSuspend { call() } // OK before 1.4
    takeSuspend(::call) // In Kotlin 1.4, it also works
}
```

Using break and continue inside when expressions included in loops

In Kotlin 1.3, you could not use unqualified `break` and `continue` inside `when` expressions included in loops. The reason was that these keywords were reserved for possible fall-through behavior in `when` expressions.

That's why if you wanted to use `break` and `continue` inside `when` expressions in loops, you had to label them, which became rather cumbersome.

```
fun test(xs: List<Int>) {
    LOOP@for (x in xs) {
        when (x) {
            2 -> continue@LOOP
            17 -> break@LOOP
            else -> println(x)
        }
    }
}
```

In Kotlin 1.4, you can use `break` and `continue` without labels inside `when` expressions included in loops. They behave as expected by terminating the nearest enclosing loop or proceeding to its next step.

```
fun test(xs: List<Int>) {
    for (x in xs) {
        when (x) {
            2 -> continue
            17 -> break
            else -> println(x)
        }
    }
}
```

The fall-through behavior inside `when` is subject to further design.

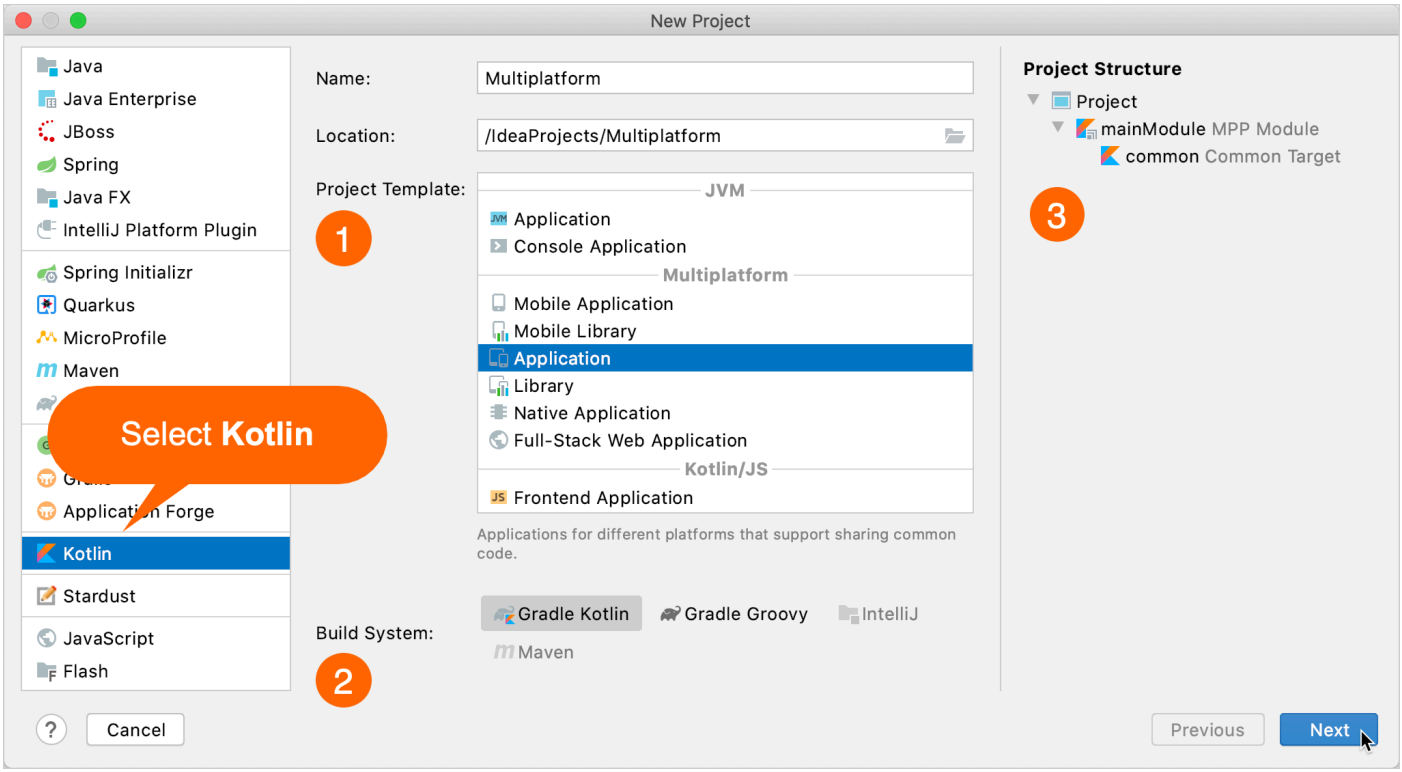
New tools in the IDE

With Kotlin 1.4, you can use the new tools in IntelliJ IDEA to simplify Kotlin development:

- New flexible Project Wizard
- Coroutine Debugger

New flexible Project Wizard

With the flexible new Kotlin Project Wizard, you have a place to easily create and configure different types of Kotlin projects, including multiplatform projects, which can be difficult to configure without a UI.



Kotlin Project Wizard – Multiplatform project

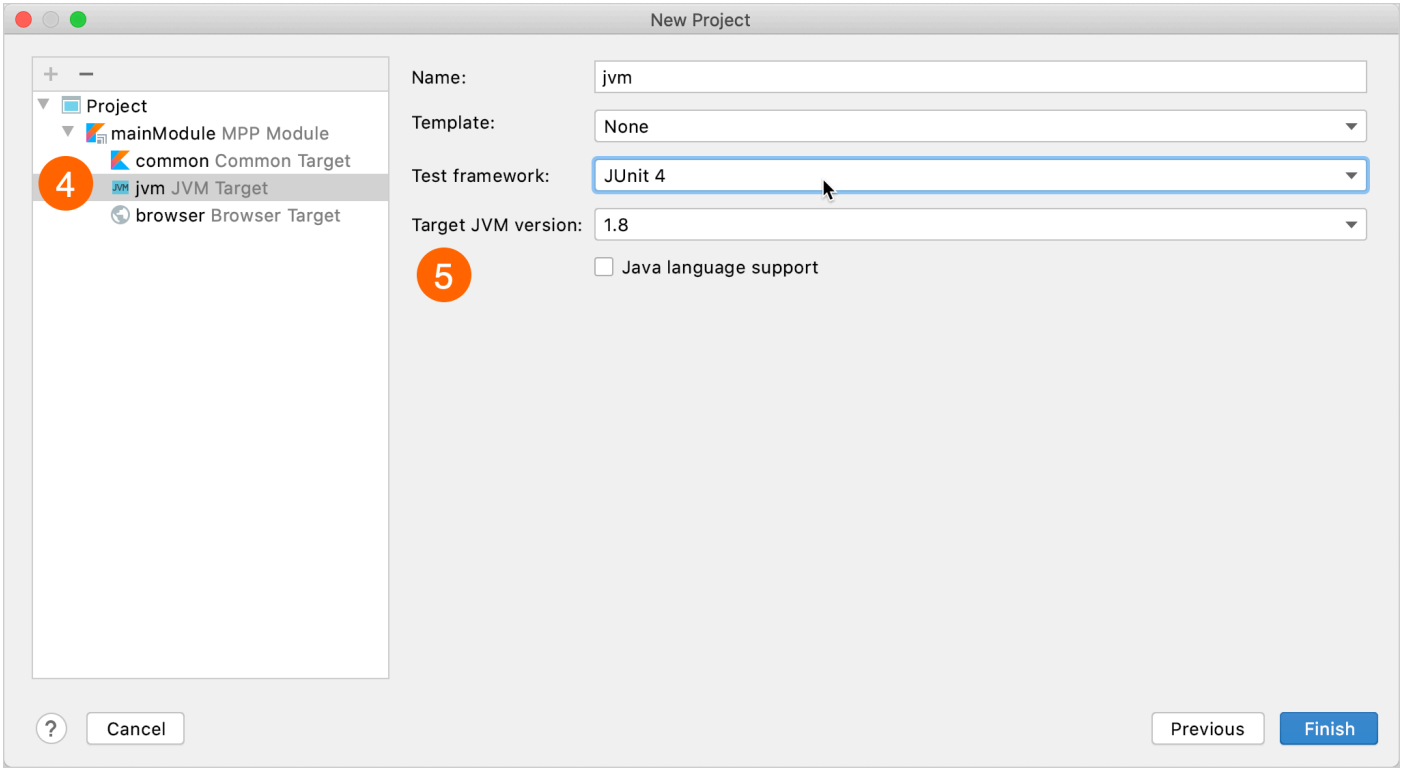
The new Kotlin Project Wizard is both simple and flexible:

1. *Select the project template*, depending on what you're trying to do. More templates will be added in the future.

2. *Select the build system* – Gradle (Kotlin or Groovy DSL), Maven, or IntelliJ IDEA.
   The Kotlin Project Wizard will only show the build systems supported on the selected project template.
3. *Preview the project structure* directly on the main screen.

Then you can finish creating your project or, optionally, *configure the project* on the next screen:

4. *Add/remove modules and targets* supported for this project template.
5. *Configure module and target settings*, for example, the target JVM version, target template, and test framework.



Kotlin Project Wizard - Configure targets

In the future, we are going to make the Kotlin Project Wizard even more flexible by adding more configuration options and templates.

You can try out the new Kotlin Project Wizard by working through these tutorials:

- Create a console application based on Kotlin/JVM
- Create a Kotlin/JS application for React
- Create a Kotlin/Native application

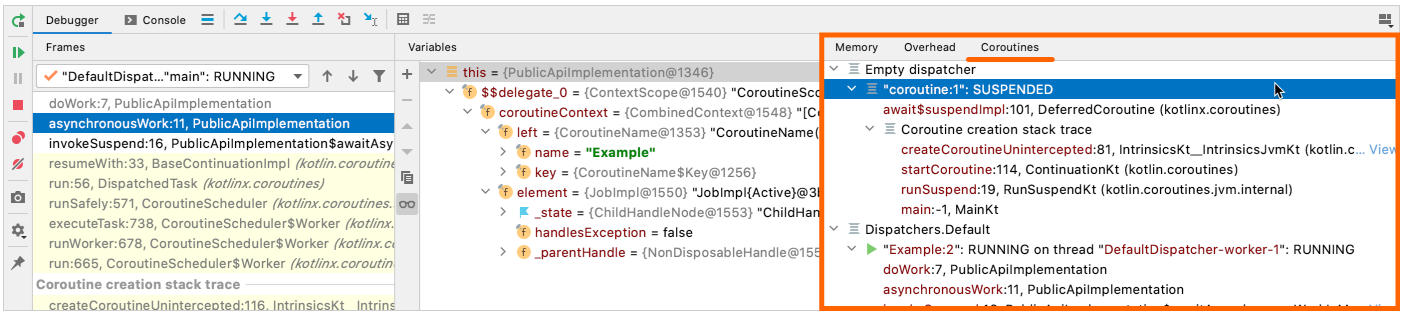
Coroutine Debugger

Many people already use coroutines for asynchronous programming. But when it came to debugging, working with coroutines before Kotlin 1.4, could be a real pain. Since coroutines jumped between threads, it was difficult to understand what a specific coroutine was doing and check its context. In some cases, tracking steps over breakpoints simply didn't work. As a result, you had to rely on logging or mental effort to debug code that used coroutines.

In Kotlin 1.4, debugging coroutines is now much more convenient with the new functionality shipped with the Kotlin plugin.

Debugging works for versions 1.3.8 or later of `kotlinx-coroutines-core`.

The **Debug Tool Window** now contains a new **Coroutines** tab. In this tab, you can find information about both currently running and suspended coroutines. The coroutines are grouped by the dispatcher they are running on.

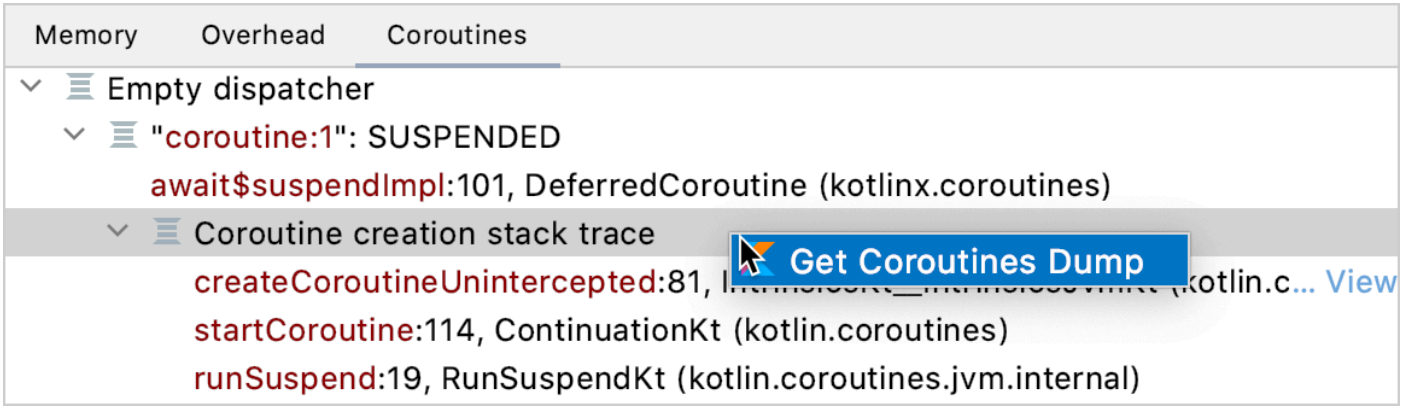


Debugging coroutines

Now you can:

- Easily check the state of each coroutine.
- See the values of local and captured variables for both running and suspended coroutines.
- See a full coroutine creation stack, as well as a call stack inside the coroutine. The stack includes all frames with variable values, even those that would be lost during standard debugging.

If you need a full report containing the state of each coroutine and its stack, right-click inside the **Coroutines** tab, and then click **Get Coroutines Dump**. Currently, the coroutines dump is rather simple, but we're going to make it more readable and helpful in future versions of Kotlin.



Coroutines Dump

Learn more about debugging coroutines in this blog post and IntelliJ IDEA documentation.

New compiler

The new Kotlin compiler is going to be really fast; it will unify all the supported platforms and provide an API for compiler extensions. It's a long-term project, and we've already completed several steps in Kotlin 1.4.0:

- New, more powerful type inference algorithm is enabled by default.
- New JVM and JS IR backends. They will become the default once we stabilize them.

New more powerful type inference algorithm

Kotlin 1.4 uses a new, more powerful type inference algorithm. This new algorithm was already available to try in Kotlin 1.3 by specifying a compiler option, and now it's used by default. You can find the full list of issues fixed in the new algorithm in YouTrack. Here you can find some of the most noticeable improvements:

- More cases where type is inferred automatically
- Smart casts for a lambda's last expression
- Smart casts for callable references
- Better inference for delegated properties
- SAM conversion for Java interfaces with different arguments
- Java SAM interfaces in Kotlin

More cases where type is inferred automatically

The new inference algorithm infers types for many cases where the old algorithm required you to specify them explicitly. For instance, in the following example the type of the lambda parameter `it` is correctly inferred to `String?`:

```
val rulesMap: Map<String, (String?) -> Boolean> = mapOf(
    "weak" to { it != null },
    "medium" to { !it.isNullOrBlank() },
    "strong" to { it != null && "^[a-zA-Z0-9]+$".toRegex().matches(it) }
)

fun main() {
    println(rulesMap.getValue("weak")("abc!"))
    println(rulesMap.getValue("strong")("abc"))
    println(rulesMap.getValue("strong")("abc!"))
}
```

In Kotlin 1.3, you needed to introduce an explicit lambda parameter or replace `to` with a `Pair` constructor with explicit generic arguments to make it work.

Smart casts for a lambda's last expression

In Kotlin 1.3, the last expression inside a lambda wasn't smart cast unless you specified the expected type. Thus, in the following example, Kotlin 1.3 infers `String?` as the type of the `result` variable:

```
val result = run {
    var str = currentValue()
    if (str == null) {
        str = "test"
    }
    str // the Kotlin compiler knows that str is not null here
}
// The type of 'result' is String? in Kotlin 1.3 and String in Kotlin 1.4
```

In Kotlin 1.4, thanks to the new inference algorithm, the last expression inside a lambda gets smart cast, and this new, more precise type is used to infer the resulting lambda type. Thus, the type of the `result` variable becomes `String`.

In Kotlin 1.3, you often needed to add explicit casts (either `!!` or type casts like `as String`) to make such cases work, and now these casts have become unnecessary.

Smart casts for callable references

In Kotlin 1.3, you couldn't access a member reference of a smart cast type. Now in Kotlin 1.4 you can:

```
import kotlin.reflect.KFunction

sealed class Animal
class Cat : Animal() {
    fun meow() {
        println("meow")
    }
}

class Dog : Animal() {
    fun woof() {
        println("woof")
    }
}

fun perform(animal: Animal) {
    val kFunction: KFunction<*> = when (animal) {
        is Cat -> animal::meow
        is Dog -> animal::woof
    }
    kFunction.call()
}

fun main() {
    perform(Cat())
}
```

You can use different member references `animal::meow` and `animal::woof` after the animal variable has been smart cast to specific types `Cat` and `Dog`. After type checks, you can access member references corresponding to subtypes.

Better inference for delegated properties

The type of a delegated property wasn't taken into account while analyzing the delegate expression which follows the `by` keyword. For instance, the following code didn't compile before, but now the compiler correctly infers the types of the `old` and `new` parameters as `String?`:

```
import kotlin.properties.Delegates

fun main() {
    var prop: String? by Delegates.observable(null) { p, old, new ->
        println("$old -> $new")
    }
    prop = "abc"
    prop = "xyz"
}
```

SAM conversion for Java interfaces with different arguments

Kotlin has supported SAM conversions for Java interfaces from the beginning, but there was one case that wasn't supported, which was sometimes annoying when working with existing Java libraries. If you called a Java method that took two SAM interfaces as parameters, both arguments needed to be either lambdas or regular objects. You couldn't pass one argument as a lambda and another as an object.

The new algorithm fixes this issue, and you can pass a lambda instead of a SAM interface in any case, which is the way you'd naturally expect it to work.

```
// FILE: A.java
public class A {
    public static void foo(Runnable r1, Runnable r2) {}
}
```

```
// FILE: test.kt
fun test(r1: Runnable) {
    A.foo(r1) {}  // Works in Kotlin 1.4
}
```

Java SAM interfaces in Kotlin

In Kotlin 1.4, you can use Java SAM interfaces in Kotlin and apply SAM conversions to them.

```
import java.lang.Runnable

fun foo(r: Runnable) {}

fun test() {
    foo { } // OK
}
```

In Kotlin 1.3, you would have had to declare the function `foo` above in Java code to perform a SAM conversion.

Unified backends and extensibility

In Kotlin, we have three backends that generate executables: Kotlin/JVM, Kotlin/JS, and Kotlin/Native. Kotlin/JVM and Kotlin/JS don't share much code since they were developed independently of each other. Kotlin/Native is based on a new infrastructure built around an intermediate representation (IR) for Kotlin code.

We are now migrating Kotlin/JVM and Kotlin/JS to the same IR. As a result, all three backends share a lot of logic and have a unified pipeline. This allows us to implement most features, optimizations, and bug fixes only once for all platforms. Both new IR-based back-ends are in Alpha.

A common backend infrastructure also opens the door for multiplatform compiler extensions. You will be able to plug into the pipeline and add custom processing and transformations that will automatically work for all platforms.

We encourage you to use our new JVM IR and JS IR backends, which are currently in Alpha, and share your feedback with us.

Kotlin/JVM

Kotlin 1.4.0 includes a number of JVM-specific improvements, such as:

- New JVM IR backend
- New modes for generating default methods in interfaces
- Unified exception type for null checks
- Type annotations in the JVM bytecode

New JVM IR backend

Along with Kotlin/JS, we are migrating Kotlin/JVM to the unified IR backend, which allows us to implement most features and bug fixes once for all platforms. You will also be able to benefit from this by creating multiplatform extensions that will work for all platforms.

Kotlin 1.4.0 does not provide a public API for such extensions yet, but we are working closely with our partners, including Jetpack Compose, who are already building their compiler plugins using our new backend.

We encourage you to try out the new Kotlin/JVM backend, which is currently in Alpha, and to file any issues and feature requests to our issue tracker. This will help us to unify the compiler pipelines and bring compiler extensions like Jetpack Compose to the Kotlin community more quickly.

To enable the new JVM IR backend, specify an additional compiler option in your Gradle build script:

```
kotlinOptions.useIR = true
```

If you enable Jetpack Compose, you will automatically be opted in to the new JVM backend without needing to specify the compiler option in `kotlinOptions`.

When using the command-line compiler, add the compiler option `-Xuse-ir`.

You can use code compiled by the new JVM IR backend only if you've enabled the new backend. Otherwise, you will get an error. Considering this, we don't recommend that library authors switch to the new backend in production.

New modes for generating default methods

When compiling Kotlin code to targets JVM 1.8 and above, you could compile non-abstract methods of Kotlin interfaces into Java's `default` methods. For this purpose, there was a mechanism that includes the `@JvmDefault` annotation for marking such methods and the `-Xjvm-default` compiler option that enables processing of this annotation.

In 1.4.0, we've added a new mode for generating default methods: `-Xjvm-default=all` compiles *all* non-abstract methods of Kotlin interfaces to `default` Java methods. For compatibility with the code that uses the interfaces compiled without `default`, we also added `all-compatibility` mode.

For more information about default methods in the Java interop, see the interoperability documentation and this blog post.

Unified exception type for null checks

Starting from Kotlin 1.4.0, all runtime null checks will throw a `java.lang.NullPointerException` instead of `KotlinNullPointerException`, `IllegalStateException`, `IllegalArgumentException`, and `TypeCastException`. This applies to: the `!!` operator, parameter null checks in the method preamble, platform-typed expression null checks, and the `as` operator with a non-nullable type. This doesn't apply to `lateinit` null checks and explicit library function calls like `checkNotNull` or `requireNotNull`.

This change increases the number of possible null check optimizations that can be performed either by the Kotlin compiler or by various kinds of bytecode processing tools, such as the Android R8 optimizer.

Note that from a developer's perspective, things won't change that much: the Kotlin code will throw exceptions with the same error messages as before. The type of exception changes, but the information passed stays the same.

Type annotations in the JVM bytecode

Kotlin can now generate type annotations in the JVM bytecode (target version 1.8+), so that they become available in Java reflection at runtime. To emit the type annotation in the bytecode, follow these steps:

1. Make sure that your declared annotation has a proper annotation target (Java's `ElementType.TYPE_USE` or Kotlin's `AnnotationTarget.TYPE`) and retention (`AnnotationRetention.RUNTIME`).
2. Compile the annotation class declaration to JVM bytecode target version 1.8+. You can specify it with `-jvm-target=1.8` compiler option.
3. Compile the code that uses the annotation to JVM bytecode target version 1.8+ (`-jvm-target=1.8`) and add the `-Xemit-jvm-type-annotations` compiler option.

Note that the type annotations from the standard library aren't emitted in the bytecode for now because the standard library is compiled with the target version 1.6.

So far, only the basic cases are supported:

- Type annotations on method parameters, method return types and property types;
- Invariant projections of type arguments, such as `Smth<@Ann Foo>`, `Array<@Ann Foo>`.

In the following example, the `@Foo` annotation on the `String` type can be emitted to the bytecode and then used by the library code:

```
@Target(AnnotationTarget.TYPE)
annotation class Foo

class A {
    fun foo(): @Foo String = "OK"
}
```

Kotlin/JS

On the JS platform, Kotlin 1.4.0 provides the following improvements:

- New Gradle DSL
- New JS IR backend

New Gradle DSL

The `kotlin.js` Gradle plugin comes with an adjusted Gradle DSL, which provides a number of new configuration options and is more closely aligned to the DSL used by the `kotlin-multiplatform` plugin. Some of the most impactful changes include:

- Explicit toggles for the creation of executable files via `binaries.executable()`. Read more about the executing Kotlin/JS and its environment here.
- Configuration of webpack's CSS and style loaders from within the Gradle configuration via `cssSupport`. Read more about using CSS and style loaders here.

- Improved management for npm dependencies, with mandatory version numbers or semver version ranges, as well as support for *development*, *peer*, and *optional* npm dependencies using `devNpm`, `optionalNpm` and `peerNpm`. Read more about dependency management for npm packages directly from Gradle here.
- Stronger integrations for Dukat, the generator for Kotlin external declarations. External declarations can now be generated at build time, or can be manually generated via a Gradle task.

New JS IR backend

The IR backend for Kotlin/JS, which currently has Alpha stability, provides some new functionality specific to the Kotlin/JS target which is focused around the generated code size through dead code elimination, and improved interoperation with JavaScript and TypeScript, among others.

To enable the Kotlin/JS IR backend, set the key `kotlin.js.compiler=ir` in your `gradle.properties`, or pass the `IR` compiler type to the `js` function of your Gradle build script:

```
kotlin {
    js(IR) { // or: LEGACY, BOTH
        // ...
    }
    binaries.executable()
}
```

For more detailed information about how to configure the new backend, check out the Kotlin/JS IR compiler documentation.

With the new @JsExport annotation and the ability to generate TypeScript definitions **from Kotlin code**, the Kotlin/JS IR compiler backend improves JavaScript & TypeScript interoperability. This also makes it easier to integrate Kotlin/JS code with existing tooling, to create **hybrid applications** and leverage code-sharing functionality in multiplatform projects.

Learn more about the available features in the Kotlin/JS IR compiler backend.

Kotlin/Native

In 1.4.0, Kotlin/Native got a significant number of new features and improvements, including:

- Support for suspending functions in Swift and Objective-C
- Objective-C generics support by default
- Exception handling in Objective-C/Swift interop
- Generate release .dSYMs on Apple targets by default
- Performance improvements
- Simplified management of CocoaPods dependencies

Support for Kotlin's suspending functions in Swift and Objective-C

In 1.4.0, we add the basic support for suspending functions in Swift and Objective-C. Now, when you compile a Kotlin module into an Apple framework, suspending functions are available in it as functions with callbacks (`completionHandler` in the Swift/Objective-C terminology). When you have such functions in the generated framework's header, you can call them from your Swift or Objective-C code and even override them.

For example, if you write this Kotlin function:

```
suspend fun queryData(id: Int): String = ...
```

...then you can call it from Swift like so:

```
queryData(id: 17) { result, error in
   if let e = error {
       print("ERROR: \(e)")
   } else {
       print(result!)
   }
}
```

Learn more about using suspending functions in Swift and Objective-C.

Objective-C generics support by default

Previous versions of Kotlin provided experimental support for generics in Objective-C interop. Since 1.4.0, Kotlin/Native generates Apple frameworks with generics from Kotlin code by default. In some cases, this may break existing Objective-C or Swift code calling Kotlin frameworks. To have the framework header written without generics, add the `-Xno-objc-generics` compiler option.

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.all {
            freeCompilerArgs += "-Xno-objc-generics"
        }
    }
}
```

Please note that all specifics and limitations listed in the documentation on interoperability with Objective-C are still valid.

Exception handling in Objective-C/Swift interop

In 1.4.0, we slightly change the Swift API generated from Kotlin with respect to the way exceptions are translated. There is a fundamental difference in error handling between Kotlin and Swift. All Kotlin exceptions are unchecked, while Swift has only checked errors. Thus, to make Swift code aware of expected exceptions, Kotlin functions should be marked with a `@Throws` annotation specifying a list of potential exception classes.

When compiling to Swift or the Objective-C framework, functions that have or are inheriting `@Throws` annotation are represented as `NSError*`-producing methods in Objective-C and as `throws` methods in Swift.

Previously, any exceptions other than `RuntimeException` and `Error` were propagated as `NSError`. Now this behavior changes: now `NSError` is thrown only for exceptions that are instances of classes specified as parameters of `@Throws` annotation (or their subclasses). Other Kotlin exceptions that reach Swift/Objective-C are considered unhandled and cause program termination.

Generate release .dSYMs on Apple targets by default

Starting with 1.4.0, the Kotlin/Native compiler produces debug symbol files (`.dSYM`s) for release binaries on Darwin platforms by default. This can be disabled with the `-Xadd-light-debug=disable` compiler option. On other platforms, this option is disabled by default. To toggle this option in Gradle, use:

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.all {
            freeCompilerArgs += "-Xadd-light-debug={enable|disable}"
        }
    }
}
```

Learn more about crash report symbolication.

Performance improvements

Kotlin/Native has received a number of performance improvements that speed up both the development process and execution. Here are some examples:

- To improve the speed of object allocation, we now offer the mimalloc memory allocator as an alternative to the system allocator. mimalloc works up to two times faster on some benchmarks. Currently, the usage of mimalloc in Kotlin/Native is experimental; you can switch to it using the `-Xallocator=mimalloc` compiler option.
- We've reworked how C interop libraries are built. With the new tooling, Kotlin/Native produces interop libraries up to 4 times as fast as before, and artifacts are 25% to 30% the size they used to be.
- Overall runtime performance has improved because of optimizations in GC. This improvement will be especially apparent in projects with a large number of long-lived objects. `HashMap` and `HashSet` collections now work faster by escaping redundant boxing.
- In 1.3.70 we introduced two new features for improving the performance of Kotlin/Native compilation: caching project dependencies and running the compiler from the Gradle daemon. Since that time, we've managed to fix numerous issues and improve the overall stability of these features.

Simplified management of CocoaPods dependencies

Previously, once you integrated your project with the dependency manager CocoaPods, you could build an iOS, macOS, watchOS, or tvOS part of your project only in Xcode, separate from other parts of your multiplatform project. These other parts could be built in IntelliJ IDEA.

Moreover, every time you added a dependency on an Objective-C library stored in CocoaPods (Pod library), you had to switch from IntelliJ IDEA to Xcode, call `pod install`, and run the Xcode build there.

Now you can manage Pod dependencies right in IntelliJ IDEA while enjoying the benefits it provides for working with code, such as code highlighting and completion. You can also build the whole Kotlin project with Gradle, without having to switch to Xcode. This means you only have to go to Xcode when you need to write Swift/Objective-C code or run your application on a simulator or device.

Now you can also work with Pod libraries stored locally.

Depending on your needs, you can add dependencies between:

- A Kotlin project and Pod libraries stored remotely in the CocoaPods repository or stored locally on your machine.
- A Kotlin Pod (Kotlin project used as a CocoaPods dependency) and an Xcode project with one or more targets.

Complete the initial configuration, and when you add a new dependency to `cocoapods`, just re-import the project in IntelliJ IDEA. The new dependency will be added automatically. No additional steps are required.

Learn how to add dependencies.

Kotlin Multiplatform

Support for multiplatform projects is in Alpha. It may change incompatibly and require manual migration in the future. We appreciate your feedback on it in YouTrack.

Kotlin Multiplatform reduces time spent writing and maintaining the same code for different platforms while retaining the flexibility and benefits of native programming. We continue investing our effort in multiplatform features and improvements:

- Sharing code in several targets with the hierarchical project structure
- Leveraging native libs in the hierarchical structure
- Specifying kotlinx dependencies only once

Multiplatform projects require Gradle 6.0 or later.

Sharing code in several targets with the hierarchical project structure

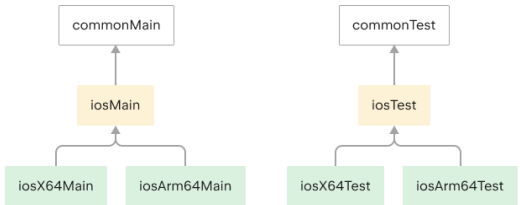
With the new hierarchical project structure support, you can share code among several platforms in a multiplatform project.

Previously, any code added to a multiplatform project could be placed either in a platform-specific source set, which is limited to one target and can't be reused by any other platform, or in a common source set, like `commonMain` or `commonTest`, which is shared across all the platforms in the project. In the common source set, you could only call a platform-specific API by using an `expect` declaration that needs platform-specific `actual` implementations.

This made it easy to share code on all platforms, but it was not so easy to share between only some of the targets, especially similar ones that could potentially reuse a lot of the common logic and third-party APIs.

For example, in a typical multiplatform project targeting iOS, there are two iOS-related targets: one for iOS ARM64 devices, and the other for the x64 simulator. They have separate platform-specific source sets, but in practice, there is rarely a need for different code for the device and simulator, and their dependencies are much alike. So iOS-specific code could be shared between them.

Apparently, in this setup, it would be desirable to have a *shared source set for two iOS targets*, with Kotlin/Native code that could still directly call any of the APIs that are common to both the iOS device and the simulator.



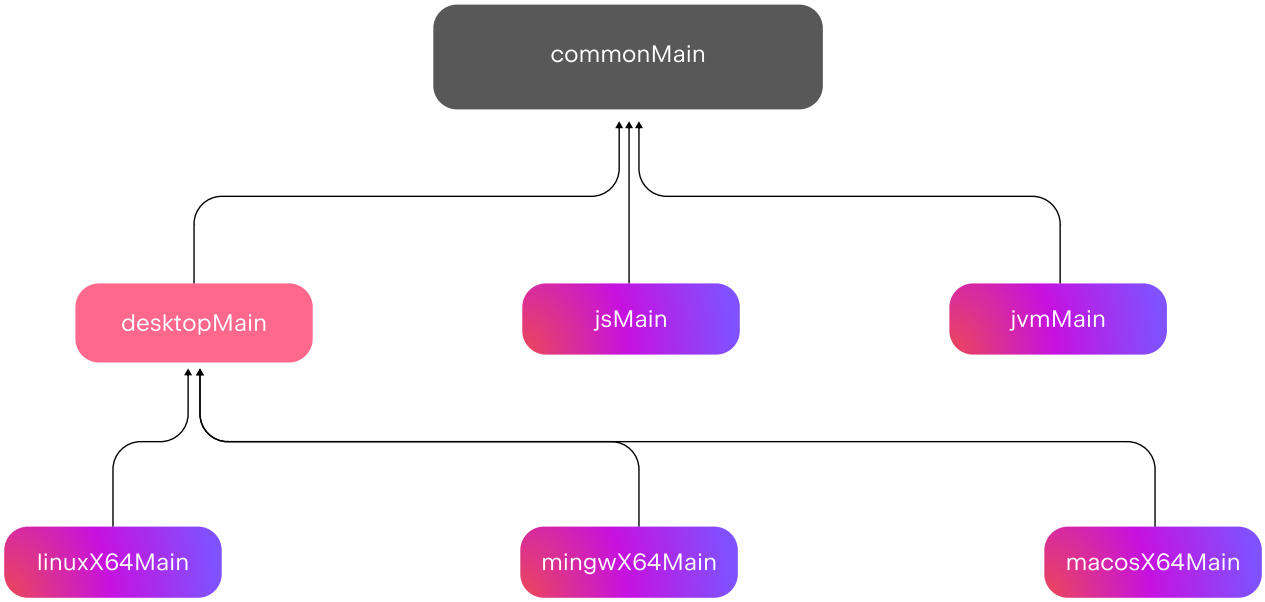
Code shared for iOS targets

Now you can do this with the hierarchical project structure support, which infers and adapts the API and language features available in each source set based on which targets consume them.

For common combinations of targets, you can create a hierarchical structure with target shortcuts. For example, create two iOS targets and the shared source set shown above with the `ios()` shortcut:

```
kotlin {
    ios() // iOS device and simulator targets; iosMain and iosTest source sets
}
```

For other combinations of targets, create a hierarchy manually by connecting the source sets with the `dependsOn` relation.



Hierarchical structure

Kotlin

```
kotlin{
    sourceSets {
        val desktopMain by creating {
            dependsOn(commonMain)
        }
        val linuxX64Main by getting {
            dependsOn(desktopMain)
        }
        val mingwX64Main by getting {
            dependsOn(desktopMain)
        }
        val macosX64Main by getting {
            dependsOn(desktopMain)
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        desktopMain {
            dependsOn(commonMain)
        }
        linuxX64Main {
            dependsOn(desktopMain)
        }
        mingwX64Main {
            dependsOn(desktopMain)
        }
        macosX64Main {
            dependsOn(desktopMain)
        }
    }
}
```

Thanks to the hierarchical project structure, libraries can also provide common APIs for a subset of targets. Learn more about sharing code in libraries.

Leveraging native libs in the hierarchical structure

You can use platform-dependent libraries, such as Foundation, UIKit, and POSIX, in source sets shared among several native targets. This can help you share more native code without being limited by platform-specific dependencies.

No additional steps are required – everything is done automatically. IntelliJ IDEA will help you detect common declarations that you can use in the shared code.

Learn more about usage of platform-dependent libraries.

Specifying dependencies only once

From now on, instead of specifying dependencies on different variants of the same library in shared and platform-specific source sets where it is used, you should specify a dependency only once in the shared source set.

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0")
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0'
            }
        }
    }
}
```

Don't use kotlinx library artifact names with suffixes specifying the platform, such as `-common`, `-native`, or similar, as they are NOT supported anymore. Instead, use the library base artifact name, which in the example above is `kotlinx-coroutines-core`.

However, the change doesn't currently affect:

- The `stdlib` library – starting from Kotlin 1.4.0, the stdlib dependency is added automatically.
- The `kotlin.test` library – you should still use `test-common` and `test-annotations-common`. These dependencies will be addressed later.

If you need a dependency only for a specific platform, you can still use platform-specific variants of standard and kotlinx libraries with such suffixes as `-jvm` or `-js`, for example `kotlinx-coroutines-core-jvm`.

Learn more about configuring dependencies.

Gradle project improvements

Besides Gradle project features and improvements that are specific to Kotlin Multiplatform, Kotlin/JVM, Kotlin/Native, and Kotlin/JS, there are several changes applicable to all Kotlin Gradle projects:

- Dependency on the standard library is now added by default
- Kotlin projects require a recent version of Gradle
- Improved support for Kotlin Gradle DSL in the IDE

Dependency on the standard library added by default

You no longer need to declare a dependency on the `stdlib` library in any Kotlin Gradle project, including a multiplatform one. The dependency is added by default.

The automatically added standard library will be the same version of the Kotlin Gradle plugin, since they have the same versioning.

For platform-specific source sets, the corresponding platform-specific variant of the library is used, while a common standard library is added to the rest. The Kotlin Gradle plugin will select the appropriate JVM standard library depending on the `kotlinOptions.jvmTarget` compiler option of your Gradle build script.

Learn how to change the default behavior.

Minimum Gradle version for Kotlin projects

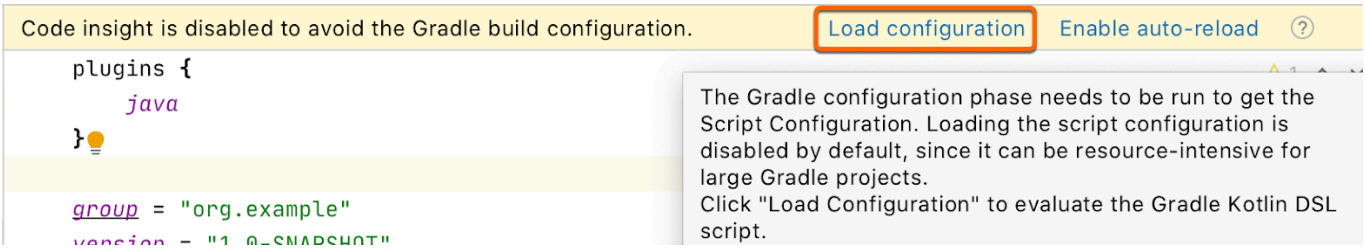
To enjoy the new features in your Kotlin projects, update Gradle to the latest version. Multiplatform projects require Gradle 6.0 or later, while other Kotlin projects work with Gradle 5.4 or later.

Improved *.gradle.kts support in the IDE

In 1.4.0, we continued improving the IDE support for Gradle Kotlin DSL scripts (`*.gradle.kts` files). Here is what the new version brings:

- *Explicit loading of script configurations* for better performance. Previously, the changes you make to the build script were loaded automatically in the background. To improve the performance, we've disabled the automatic loading of build script configuration in 1.4.0. Now the IDE loads the changes only when you explicitly apply them.

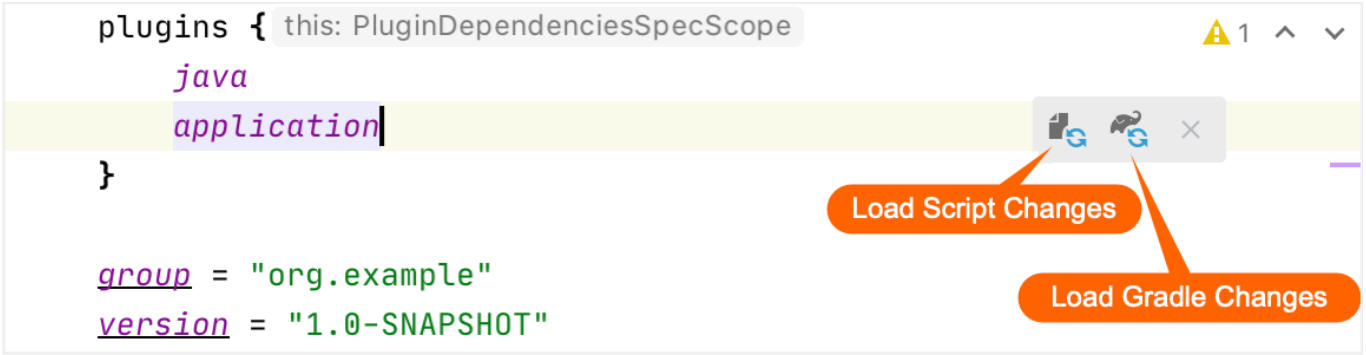
  In Gradle versions earlier than 6.0, you need to manually load the script configuration by clicking **Load Configuration** in the editor.



*.gradle.kts – Load Configuration

  In Gradle 6.0 and above, you can explicitly apply changes by clicking **Load Gradle Changes** or by reimporting the Gradle project.

  We've added one more action in IntelliJ IDEA 2020.1 with Gradle 6.0 and above – **Load Script Configurations**, which loads changes to the script configurations without updating the whole project. This takes much less time than reimporting the whole project.



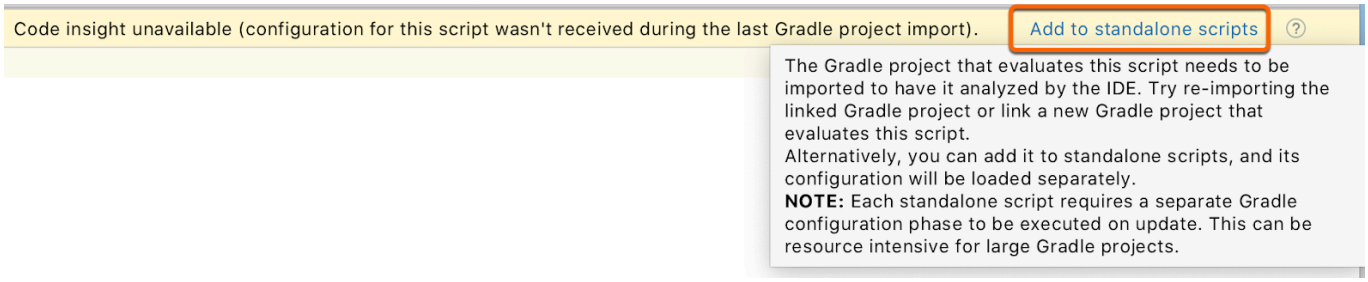
*.gradle.kts – Load Script Changes and Load Gradle Changes

  You should also **Load Script Configurations** for newly created scripts or when you open a project with new Kotlin plugin for the first time.

  With Gradle 6.0 and above, you are now able to load all scripts at once as opposed to the previous implementation where they were loaded individually. Since each request requires the Gradle configuration phase to be executed, this could be resource-intensive for large Gradle projects.

  Currently, such loading is limited to `build.gradle.kts` and `settings.gradle.kts` files (please vote for the related issue). To enable highlighting for `init.gradle.kts` or applied script plugins, use the old mechanism –

adding them to standalone scripts. Configuration for that scripts will be loaded separately when you need it. You can also enable auto-reload for such scripts.



*.gradle.kts – Add to standalone scripts

- *Better error reporting*. Previously you could only see errors from the Gradle Daemon in separate log files. Now the Gradle Daemon returns all the information about errors directly and shows it in the Build tool window. This saves you both time and effort.

Standard library

Here is the list of the most significant changes to the Kotlin standard library in 1.4.0:

- Common exception processing API
- New functions for arrays and collections
- Functions for string manipulations
- Bit operations
- Delegated properties improvements
- Converting from KType to Java Type
- Proguard configurations for Kotlin reflection
- Improving the existing API
- module-info descriptors for stdlib artifacts
- Deprecations
- Exclusion of the deprecated experimental coroutines

Common exception processing API

The following API elements have been moved to the common library:

- `Throwable.stackTraceToString()` extension function, which returns the detailed description of this throwable with its stack trace, and `Throwable.printStackTrace()`, which prints this description to the standard error output.
- `Throwable.addSuppressed()` function, which lets you specify the exceptions that were suppressed in order to deliver the exception, and the `Throwable.suppressedExceptions` property, which returns a list of all the suppressed exceptions.
- `@Throws` annotation, which lists exception types that will be checked when the function is compiled to a platform method (on JVM or native platforms).

New functions for arrays and collections

Collections

In 1.4.0, the standard library includes a number of useful functions for working with **collections**:

- `setOfNotNull()`, which makes a set consisting of all the non-null items among the provided arguments.

```
fun main() {
    val set = setOfNotNull(null, 1, 2, 0, null)
    println(set)
}
```

- `shuffled()` for sequences.

```
fun main() {
    val numbers = (0 until 50).asSequence()
    val result = numbers.map { it * 2 }.shuffled().take(5)
    println(result.toList()) //five random even numbers below 100
}
```

- `*Indexed()` counterparts for `onEach()` and `flatMap()`. The operation that they apply to the collection elements has the element index as a parameter.

```
fun main() {
    listOf("a", "b", "c", "d").onEachIndexed {
        index, item -> println(index.toString() + ":" + item)
    }

   val list = listOf("hello", "kot", "lin", "world")
          val kotlin = list.flatMapIndexed { index, item ->
              if (index in 1..2) item.toList() else emptyList()
          }
          println(kotlin)
}
```

- `*OrNull()` counterparts `randomOrNull()`, `reduceOrNull()`, and `reduceIndexedOrNull()`. They return `null` on empty collections.

```
fun main() {
     val empty = emptyList<Int>()
     empty.reduceOrNull { a, b -> a + b }
     //empty.reduce { a, b -> a + b } // Exception: Empty collection can't be reduced.
}
```

- `runningFold()`, its synonym `scan()`, and `runningReduce()` apply the given operation to the collection elements sequentially, similarly to `fold()` and `reduce()`; the difference is that these new functions return the whole sequence of intermediate results.

```
fun main() {
    val numbers = mutableListOf(0, 1, 2, 3, 4, 5)
    val runningReduceSum = numbers.runningReduce { sum, item -> sum + item }
    val runningFoldSum = numbers.runningFold(10) { sum, item -> sum + item }
    println(runningReduceSum.toString())
    println(runningFoldSum.toString())
}
```

- `sumOf()` takes a selector function and returns a sum of its values for all elements of a collection. `sumOf()` can produce sums of the types `Int`, `Long`, `Double`, `UInt`, and `ULong`. On the JVM, `BigInteger` and `BigDecimal` are also available.

```
data class OrderItem(val name: String, val price: Double, val count: Int)

fun main() {
    val order = listOf<OrderItem>(
        OrderItem("Cake", price = 10.0, count = 1),
        OrderItem("Coffee", price = 2.5, count = 3),
        OrderItem("Tea", price = 1.5, count = 2))

    val total = order.sumOf { it.price * it.count } // Double
    val count = order.sumOf { it.count } // Int
    println("You've ordered $count items that cost $total in total")
}
```

- The `min()` and `max()` functions have been renamed to `minOrNull()` and `maxOrNull()` to comply with the naming convention used across the Kotlin collections API. An `*OrNull` suffix in the function name means that it returns `null` if the receiver collection is empty. The same applies to `minBy()`, `maxBy()`, `minWith()`, `maxWith()` – in 1.4, they have `*OrNull()` synonyms.

- The new `minOf()` and `maxOf()` extension functions return the minimum and the maximum value of the given selector function on the collection items.

```
data class OrderItem(val name: String, val price: Double, val count: Int)

fun main() {
    val order = listOf<OrderItem>(
        OrderItem("Cake", price = 10.0, count = 1),
        OrderItem("Coffee", price = 2.5, count = 3),
        OrderItem("Tea", price = 1.5, count = 2))
    val highestPrice = order.maxOf { it.price }
    println("The most expensive item in the order costs $highestPrice")
}
```

  There are also `minOfWith()` and `maxOfWith()`, which take a `Comparator` as an argument, and `*OrNull()` versions of all four functions that return `null` on empty collections.

- New overloads for `flatMap` and `flatMapTo` let you use transformations with return types that don't match the receiver type, namely:
  - Transformations to `Sequence` on `Iterable`, `Array`, and `Map`
  - Transformations to `Iterable` on `Sequence`

```
fun main() {
    val list = listOf("kot", "lin")
    val lettersList = list.flatMap { it.asSequence() }
    val lettersSeq = list.asSequence().flatMap { it.toList() }
    println(lettersList)
    println(lettersSeq.toList())
}
```

- `removeFirst()` and `removeLast()` shortcuts for removing elements from mutable lists, and `*orNull()` counterparts of these functions.

Arrays

To provide a consistent experience when working with different container types, we've also added new functions for **arrays**:

- `shuffle()` puts the array elements in a random order.
- `onEach()` performs the given action on each array element and returns the array itself.
- `associateWith()` and `associateWithTo()` build maps with the array elements as keys.
- `reverse()` for array subranges reverses the order of the elements in the subrange.
- `sortDescending()` for array subranges sorts the elements in the subrange in descending order.
- `sort()` and `sortWith()` for array subranges are now available in the common library.

```
fun main() {
    var language = ""
    val letters = arrayOf("k", "o", "t", "l", "i", "n")
    val fileExt = letters.onEach { language += it }
       .filterNot { it in "aeuio" }.take(2)
       .joinToString(prefix = ".", separator = "")
    println(language) // "kotlin"
    println(fileExt) // ".kt"

    letters.shuffle()
    letters.reverse(0, 3)
    letters.sortDescending(2, 5)
    println(letters.contentToString()) // [k, o, t, l, i, n]
}
```

Additionally, there are new functions for conversions between `CharArray` / `ByteArray` and `String`:

- `ByteArray.decodeToString()` and `String.encodeToByteArray()`
- `CharArray.concatToString()` and `String.toCharArray()`

```
fun main() {
        val str = "kotlin"
    val array = str.toCharArray()
    println(array.concatToString())
}
```

ArrayDeque

We've also added the `ArrayDeque` class – an implementation of a double-ended queue. A double-ended queue lets you add or remove elements both at the beginning or end of the queue in an amortized constant time. You can use a double-ended queue by default when you need a queue or a stack in your code.

```
fun main() {
    val deque = ArrayDeque(listOf(1, 2, 3))

    deque.addFirst(0)
    deque.addLast(4)
    println(deque) // [0, 1, 2, 3, 4]

    println(deque.first()) // 0
    println(deque.last()) // 4

    deque.removeFirst()
    deque.removeLast()
    println(deque) // [1, 2, 3]
}
```

The `ArrayDeque` implementation uses a resizable array underneath: it stores the contents in a circular buffer, an `Array`, and resizes this `Array` only when it becomes full.

Functions for string manipulations

The standard library in 1.4.0 includes a number of improvements in the API for string manipulation:

- `StringBuilder` has useful new extension functions: `set()`, `setRange()`, `deleteAt()`, `deleteRange()`, `appendRange()`, and others.

  ```
  fun main() {
              val sb = StringBuilder("Bye Kotlin 1.3.72")
          sb.deleteRange(0, 3)
          sb.insertRange(0, "Hello", 0 ,5)
          sb.set(15, '4')
          sb.setRange(17, 19, "0")
          print(sb.toString())
      //sampleEnd
      }
  ```

- Some existing functions of `StringBuilder` are available in the common library. Among them are `append()`, `insert()`, `substring()`, `setLength()`, and more.
- New functions `Appendable.appendLine()` and `StringBuilder.appendLine()` have been added to the common library. They replace the JVM-only `appendln()` functions of these classes.

  ```
  fun main() {
      println(buildString {
          appendLine("Hello,")
          appendLine("world")
      })
  }
  ```

Bit operations

New functions for bit manipulations:

- `countOneBits()`
- `countLeadingZeroBits()`
- `countTrailingZeroBits()`

- `takeHighestOneBit()`
- `takeLowestOneBit()`
- `rotateLeft()` and `rotateRight()` (experimental)

```
fun main() {
    val number = "1010000".toInt(radix = 2)
    println(number.countOneBits())
    println(number.countTrailingZeroBits())
    println(number.takeHighestOneBit().toString(2))
}
```

Delegated properties improvements

In 1.4.0, we have added new features to improve your experience with delegated properties in Kotlin:

- Now a property can be delegated to another property.
- A new interface `PropertyDelegateProvider` helps create delegate providers in a single declaration.
- `ReadWriteProperty` now extends `ReadOnlyProperty` so you can use both of them for read-only properties.

Aside from the new API, we've made some optimizations that reduce the resulting bytecode size. These optimizations are described in this blog post.

Learn more about delegated properties.

Converting from KType to Java Type

A new extension property `KType.javaType` (currently experimental) in the stdlib helps you obtain a `java.lang.reflect.Type` from a Kotlin type without using the whole `kotlin-reflect` dependency.

```
import kotlin.reflect.javaType
import kotlin.reflect.typeOf

@OptIn(ExperimentalStdlibApi::class)
inline fun <reified T> accessReifiedTypeArg() {
   val kType = typeOf<T>()
   println("Kotlin type: $kType")
   println("Java type: ${kType.javaType}")
}

@OptIn(ExperimentalStdlibApi::class)
fun main() {
   accessReifiedTypeArg<String>()
   // Kotlin type: kotlin.String
   // Java type: class java.lang.String

   accessReifiedTypeArg<List<String>>()
   // Kotlin type: kotlin.collections.List<kotlin.String>
   // Java type: java.util.List<java.lang.String>
}
```

Proguard configurations for Kotlin reflection

Starting from 1.4.0, we have embedded Proguard/R8 configurations for Kotlin Reflection in `kotlin-reflect.jar`. With this in place, most Android projects using R8 or Proguard should work with kotlin-reflect without needing

any additional configuration. You no longer need to copy-paste the Proguard rules for kotlin-reflect internals. But note that you still need to explicitly list all the APIs you're going to reflect on.

Improving the existing API

- Several functions now work on null receivers, for example:
  - `toBoolean()` on strings
  - `contentEquals()`, `contentHashcode()`, `contentToString()` on arrays
- `NaN`, `NEGATIVE_INFINITY`, and `POSITIVE_INFINITY` in `Double` and `Float` are now defined as `const`, so you can use them as annotation arguments.
- New constants `SIZE_BITS` and `SIZE_BYTES` in `Double` and `Float` contain the number of bits and bytes used to represent an instance of the type in binary form.
- The `maxOf()` and `minOf()` top-level functions can accept a variable number of arguments (`vararg`).

module-info descriptors for stdlib artifacts

Kotlin 1.4.0 adds `module-info.java` module information to default standard library artifacts. This lets you use them with jlink tool, which generates custom Java runtime images containing only the platform modules that are required for your app. You could already use jlink with Kotlin standard library artifacts, but you had to use separate artifacts to do so – the ones with the "modular" classifier – and the whole setup wasn't straightforward.
In Android, make sure you use the Android Gradle plugin version 3.2 or higher, which can correctly process jar files with module-info.

Deprecations

toShort() and toByte() of Double and Float

We've deprecated the functions `toShort()` and `toByte()` on `Double` and `Float` because they could lead to unexpected results because of the narrow value range and smaller variable size.

To convert floating-point numbers to `Byte` or `Short`, use the two-step conversion: first, convert them to `Int`, and then convert them again to the target type.

contains(), indexOf(), and lastIndexOf() on floating-point arrays

We've deprecated the `contains()`, `indexOf()`, and `lastIndexOf()` extension functions of `FloatArray` and `DoubleArray` because they use the IEEE 754 standard equality, which contradicts the total order equality in some corner cases. See this issue for details.

min() and max() collection functions

We've deprecated the `min()` and `max()` collection functions in favor of `minOrNull()` and `maxOrNull()`, which more properly reflect their behavior – returning `null` on empty collections. See this issue for details.

Exclusion of the deprecated experimental coroutines

The `kotlin.coroutines.experimental` API was deprecated in favor of kotlin.coroutines in 1.3.0. In 1.4.0, we're completing the deprecation cycle for `kotlin.coroutines.experimental` by removing it from the standard

library. For those who still use it on the JVM, we've provided a compatibility artifact `kotlin-coroutines-experimental-compat.jar` with all the experimental coroutines APIs. We've published it to Maven, and we include it in the Kotlin distribution alongside the standard library.

Stable JSON serialization

With Kotlin 1.4.0, we are shipping the first stable version of kotlinx.serialization - 1.0.0-RC. Now we are pleased to declare the JSON serialization API in `kotlinx-serialization-core` (previously known as `kotlinx-serialization-runtime`) stable. Libraries for other serialization formats remain experimental, along with some advanced parts of the core library.

We have significantly reworked the API for JSON serialization to make it more consistent and easier to use. From now on, we'll continue developing the JSON serialization API in a backward-compatible manner. However, if you have used previous versions of it, you'll need to rewrite some of your code when migrating to 1.0.0-RC. To help you with this, we also offer the Kotlin Serialization Guide – the complete set of documentation for `kotlinx.serialization`. It will guide you through the process of using the most important features and it can help you address any issues that you might face.

Note: `kotlinx-serialization` 1.0.0-RC only works with Kotlin compiler 1.4. Earlier compiler versions are not compatible.

Scripting and REPL

In 1.4.0, scripting in Kotlin benefits from a number of functional and performance improvements along with other updates. Here are some of the key changes:

- New dependencies resolution API
- New REPL API
- Compiled scripts cache
- Artifacts renaming

To help you become more familiar with scripting in Kotlin, we've prepared a project with examples. It contains examples of the standard scripts (`*.main.kts`) and examples of uses of the Kotlin Scripting API and custom script definitions. Please give it a try and share your feedback using our issue tracker.

New dependencies resolution API

In 1.4.0, we've introduced a new API for resolving external dependencies (such as Maven artifacts), along with implementations for it. This API is published in the new artifacts `kotlin-scripting-dependencies` and `kotlin-scripting-dependencies-maven`. The previous dependency resolution functionality in `kotlin-script-util` library is now deprecated.

New REPL API

The new experimental REPL API is now a part of the Kotlin Scripting API. There are also several implementations of it in the published artifacts, and some have advanced functionality, such as code completion. We use this API in the Kotlin Jupyter kernel and now you can try it in your own custom shells and REPLs.

Compiled scripts cache

The Kotlin Scripting API now provides the ability to implement a compiled scripts cache, significantly speeding up subsequent executions of unchanged scripts. Our default advanced script implementation `kotlin-main-kts` already has its own cache.

Artifacts renaming

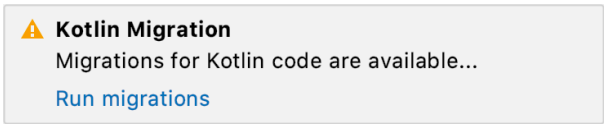
In order to avoid confusion about artifact names, we've renamed `kotlin-scripting-jsr223-embeddable` and `kotlin-scripting-jvm-host-embeddable` to just `kotlin-scripting-jsr223` and `kotlin-scripting-jvm-host`. These artifacts depend on the `kotlin-compiler-embeddable` artifact, which shades the bundled third-party libraries to avoid usage conflicts. With this renaming, we're making the usage of `kotlin-compiler-embeddable` (which is safer in general) the default for scripting artifacts. If, for some reason, you need artifacts that depend on the unshaded `kotlin-compiler`, use the artifact versions with the `-unshaded` suffix, such as `kotlin-scripting-jsr223-unshaded`. Note that this renaming affects only the scripting artifacts that are supposed to be used directly; names of other artifacts remain unchanged.

Migrating to Kotlin 1.4.0

The Kotlin plugin's migration tools help you migrate your projects from earlier versions of Kotlin to 1.4.0.

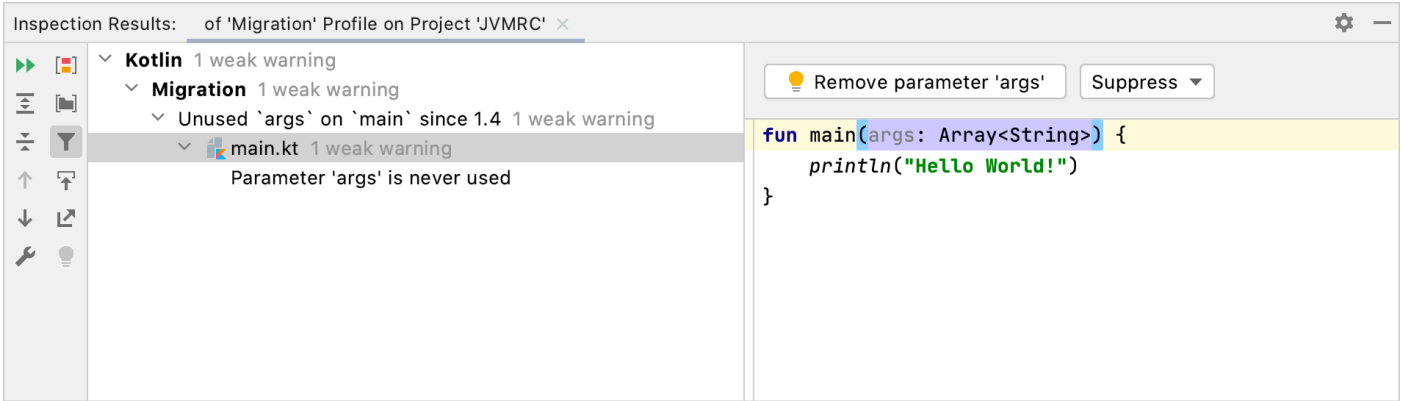
Just change the Kotlin version to `1.4.0` and re-import your Gradle or Maven project. The IDE will then ask you about migration.

If you agree, it will run migration code inspections that will check your code and suggest corrections for anything that doesn't work or that is not recommended in 1.4.0.



Run migration

Code inspections have different severity levels, to help you decide which suggestions to accept and which to ignore.



Migration inspections

Kotlin 1.4.0 is a feature release and therefore can bring incompatible changes to the language. Find the detailed list of such changes in the Compatibility Guide for Kotlin 1.4.

Compatibility guide for Kotlin 1.4.x

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.3 to Kotlin 1.4.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language and stdlib

Unexpected behavior with in infix operator and ConcurrentHashMap

Issue: KT-18053

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.4 will outlaw auto operator `contains` coming from the implementors of `java.util.Map` written in Java

Deprecation cycle:

- < 1.4: introduce warning for problematic operators at call-site
- >= 1.4: raise this warning to an error, `-XXLanguage:-ProhibitConcurrentHashMapContains` can be used to temporarily revert to pre-1.4 behavior

Prohibit access to protected members inside public inline members

Issue: KT-21178

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.4 will prohibit access to protected members from public inline members.

Deprecation cycle:

- < 1.4: introduce warning at call-site for problematic cases
- 1.4: raise this warning to an error, `-XXLanguage:-ProhibitProtectedCallFromInline` can be used to temporarily revert to pre-1.4 behavior

Contracts on calls with implicit receivers

Issue: KT-28672

Component: Core Language

Incompatible change type: behavioral

Short summary: smart casts from contracts will be available on calls with implicit receivers in 1.4

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-ContractsOnCallsWithImplicitReceiver` can be used to temporarily revert to pre-1.4 behavior

Inconsistent behavior of floating-point number comparisons

Issues: KT-22723

Component: Core language

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, Kotlin compiler will use IEEE 754 standard to compare floating-point numbers

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-ProperIeee754Comparisons` can be used to temporarily revert to pre-1.4 behavior

No smart cast on the last expression in a generic lambda

Issue: KT-15020

Component: Core Language

Incompatible change type: behavioral

Short summary: smart casts for last expressions in lambdas will be correctly applied since 1.4

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Do not depend on the order of lambda arguments to coerce result to Unit

Issue: KT-36045

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, lambda arguments will be resolved independently without implicit coercion to `Unit`

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Wrong common supertype between raw and integer literal type leads to unsound code

Issue: KT-35681

Components: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, common supertype between raw `Comparable` type and integer literal type will be more specific

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type safety problem because several equal type variables are instantiated with a different types

Issue: KT-35679

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, Kotlin compiler will prohibit instantiating equal type variables with different types

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type safety problem because of incorrect subtyping for intersection types

Issues: KT-22474

Component: Core language

Incompatible change type: source

Short summary: in Kotlin 1.4, subtyping for intersection types will be refined to work more correctly

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

No type mismatch with an empty when expression inside lambda

Issue: KT-17995

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, there will be a type mismatch for empty `when` expression if it's used as the last expression in a lambda

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Return type Any inferred for lambda with early return with integer literal in one of possible return values

Issue: KT-20226

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, integer type returning from a lambda will be more specific for cases when there is early return

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Proper capturing of star projections with recursive types

Issue: KT-33012

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, more candidates will become applicable because capturing for recursive types will work more correctly

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Common supertype calculation with non-proper type and flexible one leads to incorrect results

Issue: KT-37054

Component: Core language

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, common supertype between flexible types will be more specific protecting from runtime errors

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type safety problem because of lack of captured conversion against nullable type argument

Issue: KT-35487

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, subtyping between captured and nullable types will be more correct protecting from runtime errors

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Preserve intersection type for covariant types after unchecked cast

Issue: KT-37280

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, uchecked casts of covariant types produce the intersection type for smart casts, not the type of the unchecked cast.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type variable leaks from builder inference because of using this expression

Issue: KT-32126

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, using `this` inside builder functions like `sequence {}` is prohibited if there are no other proper constraints

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Wrong overload resolution for contravariant types with nullable type arguments

Issue: KT-31670

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, if two overloads of a function that takes contravariant type arguments differ only by the nullability of the type (such as `In<T>` and `In<T?>`), the nullable type is considered more specific.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Builder inference with non-nested recursive constraints

Issue: KT-34975

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, builder functions such as `sequence {}` with type that depends on a recursive constraint inside the passed lambda cause a compiler error.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Eager type variable fixation leads to a contradictory constraint system

Issue: KT-25175

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, the type inference in certain cases works less eagerly allowing to find the constraint system that is not contradictory.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Prohibit tailrec modifier on open functions

Issue: KT-18541

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, functions can't have `open` and `tailrec` modifiers at the same time.

Deprecation cycle:

- < 1.4: report a warning on functions that have `open` and `tailrec` modifiers together (error in the progressive mode).
- >= 1.4: raise this warning to an error.

The INSTANCE field of a companion object more visible than the companion object class itself

Issue: KT-11567

Component: Kotlin/JVM

Incompatible change type: source

Short summary: since Kotlin 1.4, if a companion object is private, then its field `INSTANCE` will be also private

Deprecation cycle:

- < 1.4: the compiler generates object `INSTANCE` with a deprecated flag
- >= 1.4: companion object `INSTANCE` field has proper visibility

Outer finally block inserted before return is not excluded from the catch interval of the inner try block without finally

Issue: KT-31923

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, the catch interval will be computed properly for nested `try/catch` blocks

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-ProperFinally` can be used to temporarily revert to pre-1.4 behavior

Use the boxed version of an inline class in return type position for covariant and generic-specialized overrides

Issues: KT-30419

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, functions using covariant and generic-specialized overrides will return boxed values of inline classes

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Do not declare checked exceptions in JVM bytecode when using delegation to Kotlin interfaces

Issue: KT-35834

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will not generate checked exceptions during interface delegation to Kotlin interfaces

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-DoNotGenerateThrowsForDelegatedKotlinMembers` can be used to temporarily revert to pre-1.4 behavior

Changed behavior of signature-polymorphic calls to methods with a single vararg parameter to avoid wrapping the argument into another array

Issue: KT-35469

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will not wrap the argument into another array on a signature-polymorphic call

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Incorrect generic signature in annotations when KClass is used as a generic parameter

Issue: KT-35207

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will fix incorrect type mapping in annotations when KClass is used as a generic parameter

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Forbid spread operator in signature-polymorphic calls

Issue: KT-35226

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will prohibit the use of spread operator (*) on signature-polymorphic calls

Deprecation cycle:

- < 1.4: report a warning on the use of a spread operator in signature-polymorphic calls
- >= 1.5: raise this warning to an error, `-XXLanguage:-ProhibitSpreadOnSignaturePolymorphicCall` can be used to temporarily revert to pre-1.4 behavior

Change initialization order of default values for tail-recursive optimized functions

Issue: KT-31540

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since Kotlin 1.4, the initialization order for tail-recursive functions will be the same as for regular functions

Deprecation cycle:

- < 1.4: report a warning at declaration-site for problematic functions
- >= 1.4: behavior changed, `-XXLanguage:-ProperComputationOrderOfTailrecDefaultParameters` can be used to temporarily revert to pre-1.4 behavior

Do not generate ConstantValue attribute for non-const vals

Issue: KT-16615

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since Kotlin 1.4, the compiler will not generate the `ConstantValue` attribute for non-`const` `val`s

Deprecation cycle:

- < 1.4: report a warning through an IntelliJ IDEA inspection
- >= 1.4: behavior changed, `-XXLanguage:-NoConstantValueAttributeForNonConstVals` can be used to temporarily revert to pre-1.4 behavior

Generated overloads for @JvmOverloads on open methods should be final

Issue: KT-33240

Components: Kotlin/JVM

Incompatible change type: source

Short summary: overloads for functions with `@JvmOverloads` will be generated as `final`

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-GenerateJvmOverloadsAsFinal` can be used to temporarily revert to pre-1.4 behavior

Lambdas returning kotlin.Result now return boxed value instead of unboxed

Issue: KT-39198

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, lambdas returning values of `kotlin.Result` type will return boxed value instead of unboxed

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Unify exceptions from null checks

Issue: KT-22275

Component: Kotlin/JVM

Incompatible change type: behavior

Short summary: Starting from Kotlin 1.4, all runtime null checks will throw a `java.lang.NullPointerException`

Deprecation cycle:

- < 1.4: runtime null checks throw different exceptions, such as `KotlinNullPointerException`, `IllegalStateException`, `IllegalArgumentException`, and `TypeCastException`
- >= 1.4: all runtime null checks throw a `java.lang.NullPointerException`. `-Xno-unified-null-checks` can be used to temporarily revert to pre-1.4 behavior

Comparing floating-point values in array/list operations contains, indexOf, lastIndexOf: IEEE 754 or total order

Issue: KT-28753

Component: kotlin-stdlib (JVM)

Incompatible change type: behavioral

Short summary: the `List` implementation returned from `Double/FloatArray.asList()` will implement `contains`, `indexOf`, and `lastIndexOf`, so that they use total order equality

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Gradually change the return type of collection min and max functions to non-nullable

Issue: KT-38854

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: return type of collection `min` and `max` functions will be changed to non-nullable in 1.6

Deprecation cycle:

- 1.4: introduce `...OrNull` functions as synonyms and deprecate the affected API (see details in the issue)
- 1.5.x: raise the deprecation level of the affected API to error
- >=1.6: reintroduce the affected API but with non-nullable return type

Deprecate appendln in favor of appendLine

Issue: KT-38754

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: `StringBuilder.appendln()` will be deprecated in favor of `StringBuilder.appendLine()`

Deprecation cycle:

- 1.4: introduce `appendLine` function as a replacement for `appendln` and deprecate `appendln`
- >=1.5: raise the deprecation level to error

Deprecate conversions of floating-point types to Short and Byte

Issue: KT-30360

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: since Kotlin 1.4, conversions of floating-point types to `Short` and `Byte` will be deprecated

Deprecation cycle:

- 1.4: deprecate `Double.toShort()/toByte()` and `Float.toShort()/toByte()` and propose replacement
- >=1.5: raise the deprecation level to error

Fail fast in Regex.findAll on an invalid startIndex

Issue: KT-28356

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, `findAll` will be improved to check that `startIndex` is in the range of the valid position indices of the input char sequence at the moment of entering `findAll`, and throw `IndexOutOfBoundsException` if it's not

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Remove deprecated kotlin.coroutines.experimental

Issue: KT-36083

Component: kotlin-stdlib

Incompatible change type: source

Short summary: since Kotlin 1.4, the deprecated `kotlin.coroutines.experimental` API is removed from stdlib

Deprecation cycle:

- < 1.4: `kotlin.coroutines.experimental` is deprecated with the `ERROR` level
- >= 1.4: `kotlin.coroutines.experimental` is removed from stdlib. On the JVM, a separate compatibility artifact is provided (see details in the issue).

Remove deprecated mod operator

Issue: KT-26654

Component: kotlin-stdlib

Incompatible change type: source

Short summary: since Kotlin 1.4, `mod` operator on numeric types is removed from stdlib

Deprecation cycle:

- < 1.4: `mod` is deprecated with the `ERROR` level
- >= 1.4: `mod` is removed from stdlib

Hide Throwable.addSuppressed member and prefer extension instead

Issue: KT-38777

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: `Throwable.addSuppressed()` extension function is now preferred over the `Throwable.addSuppressed()` member function

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

capitalize should convert digraphs to title case

Issue: KT-38817

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: `String.capitalize()` function now capitalizes digraphs from the Serbo-Croatian Gaj's Latin alphabet in the title case (`Dž` instead of `DŽ`)

Deprecation cycle:

- < 1.4: digraphs are capitalized in the upper case (`DŽ`)
- >= 1.4: digraphs are capitalized in the title case (`Dž`)

Tools

Compiler arguments with delimiter characters must be passed in double quotes on Windows

Issue: KT-41309

Component: CLI

Incompatible change type: behavioral

Short summary: on Windows, `kotlinc.bat` arguments that contain delimiter characters (whitespace, `=` , `;` , `,`) now require double quotes (`"`)

Deprecation cycle:

- < 1.4: all compiler arguments are passed without quotes
- >= 1.4: compiler arguments that contain delimiter characters (whitespace, `=` , `;` , `,`) require double quotes (`"`)

KAPT: Names of synthetic $annotations() methods for properties have changed

Issue: KT-36926

Component: KAPT

Incompatible change type: behavioral

Short summary: names of synthetic `$annotations()` methods generated by KAPT for properties have changed in 1.4

Deprecation cycle:

- < 1.4: names of synthetic `$annotations()` methods for properties follow the template `<propertyName>@annotations()`
- >= 1.4: names of synthetic `$annotations()` methods for properties include the `get` prefix: `get<PropertyName>@annotations()`

What's new in Kotlin 1.3

Released: 29 October 2018

For information about the Kotlin release cycle, see Kotlin release process.

Coroutines release

After some long and extensive battle testing, coroutines are now released! It means that from Kotlin 1.3 the language support and the API are fully stable. Check out the new coroutines overview page.

Kotlin 1.3 introduces callable references on suspend-functions and support of coroutines in the reflection API.

Kotlin/Native

Kotlin 1.3 continues to improve and polish the Native target. See the Kotlin/Native overview for details.

Multiplatform projects

In 1.3, we've completely reworked the model of multiplatform projects in order to improve expressiveness and flexibility, and to make sharing common code easier. Also, Kotlin/Native is now supported as one of the targets!

The key differences to the old model are:

- In the old model, common and platform-specific code needed to be placed in separate modules, linked by `expectedBy` dependencies. Now, common and platform-specific code is placed in different source roots of the same module, making projects easier to configure.
- There is now a large number of preset platform configurations for different supported platforms.
- The dependencies configuration has been changed; dependencies are now specified separately for each source root.

- Source sets can now be shared between an arbitrary subset of platforms (for example, in a module that targets JS, Android and iOS, you can have a source set that is shared only between Android and iOS).
- Publishing multiplatform libraries is now supported.

For more information, please refer to the multiplatform programming documentation.

Contracts

The Kotlin compiler does extensive static analysis to provide warnings and reduce boilerplate. One of the most notable features is smartcasts — with the ability to perform a cast automatically based on the performed type checks:

```
fun foo(s: String?) {
    if (s ≠ null) s.length // Compiler automatically casts 's' to 'String'
}
```

However, as soon as these checks are extracted in a separate function, all the smartcasts immediately disappear:

```
fun String?.isNotNull(): Boolean = this ≠ null

fun foo(s: String?) {
    if (s.isNotNull()) s.length // No smartcast :(
}
```

To improve the behavior in such cases, Kotlin 1.3 introduces experimental mechanism called *contracts*.

Contracts allow a function to explicitly describe its behavior in a way which is understood by the compiler. Currently, two wide classes of cases are supported:

- Improving smartcasts analysis by declaring the relation between a function's call outcome and the passed arguments values:

```
fun require(condition: Boolean) {
    // This is a syntax form which tells the compiler:
    // "if this function returns successfully, then the passed 'condition' is true"
    contract { returns() implies condition }
    if (!condition) throw IllegalArgumentException(...)
}

fun foo(s: String?) {
    require(s is String)
    // s is smartcast to 'String' here, because otherwise
    // 'require' would have thrown an exception
}
```

- Improving the variable initialization analysis in the presence of higher-order functions:

```
fun synchronize(lock: Any?, block: () → Unit) {
    // It tells the compiler:
    // "This function will invoke 'block' here and now, and exactly one time"
    contract { callsInPlace(block, EXACTLY_ONCE) }
}

fun foo() {
    val x: Int
    synchronize(lock) {
        x = 42 // Compiler knows that lambda passed to 'synchronize' is called
               // exactly once, so no reassignment is reported
    }
    println(x) // Compiler knows that lambda will be definitely called, performing
               // initialization, so 'x' is considered to be initialized here
}
```

Contracts in stdlib

`stdlib` already makes use of contracts, which leads to improvements in the analyses described above. This part of contracts is **stable**, meaning that you can benefit from the improved analysis right now without any additional opt-ins:

```
fun bar(x: String?) {
    if (!x.isNullOrEmpty()) {
        println("length of '$x' is ${x.length}") // Yay, smartcast to not-null!
    }
}
fun main() {
    bar(null)
    bar("42")
}
```

Custom contracts

It is possible to declare contracts for your own functions, but this feature is **experimental,** as the current syntax is in a state of early prototype and will most probably be changed. Also please note that currently the Kotlin compiler does not verify contracts, so it's the responsibility of the programmer to write correct and sound contracts.

Custom contracts are introduced by a call to `contract` stdlib function, which provides DSL scope:

```
fun String?.isNullOrEmpty(): Boolean {
    contract {
        returns(false) implies (this@isNullOrEmpty ≠ null)
    }
    return this == null || isEmpty()
}
```

See the details on the syntax as well as the compatibility notice in the KEEP.

Capturing when subject in a variable

In Kotlin 1.3, it is now possible to capture the `when` subject into a variable:

```
fun Request.getBody() =
        when (val response = executeRequest()) {
            is Success -> response.body
            is HttpError -> throw HttpException(response.status)
        }
```

While it was already possible to extract this variable just before `when`, `val` in `when` has its scope properly restricted to the body of `when`, and so preventing namespace pollution. See the full documentation on `when` here.

@JvmStatic and @JvmField in companions of interfaces

With Kotlin 1.3, it is possible to mark members of a `companion` object of interfaces with annotations `@JvmStatic` and `@JvmField`. In the classfile, such members will be lifted to the corresponding interface and marked as `static`.

For example, the following Kotlin code:

```
interface Foo {
    companion object {
        @JvmField
        val answer: Int = 42

        @JvmStatic
        fun sayHello() {
            println("Hello, world!")
        }
    }
}
```

It is equivalent to this Java code:

```
interface Foo {
    public static int answer = 42;
    public static void sayHello() {
        // ...
    }
}
```

Nested declarations in annotation classes

In Kotlin 1.3, it is possible for annotations to have nested classes, interfaces, objects, and companions:

```
annotation class Foo {
    enum class Direction { UP, DOWN, LEFT, RIGHT }

    annotation class Bar

    companion object {
        fun foo(): Int = 42
        val bar: Int = 42
    }
}
```

Parameterless main

By convention, the entry point of a Kotlin program is a function with a signature like `main(args: Array<String>)`, where `args` represent the command-line arguments passed to the program. However, not every application supports command-line arguments, so this parameter often ends up not being used.

Kotlin 1.3 introduced a simpler form of `main` which takes no parameters. Now "Hello, World" in Kotlin is 19 characters shorter!

```
fun main() {
    println("Hello, world!")
}
```

Functions with big arity

In Kotlin, functional types are represented as generic classes taking a different number of parameters: `Function0<R>`, `Function1<P0, R>`, `Function2<P0, P1, R>`, ... This approach has a problem in that this list is finite, and it currently ends with `Function22`.

Kotlin 1.3 relaxes this limitation and adds support for functions with bigger arity:

```
fun trueEnterpriseComesToKotlin(block: (Any, Any, ... /* 42 more */, Any) -> Any) {
    block(Any(), Any(), ..., Any())
}
```

Progressive mode

Kotlin cares a lot about stability and backward compatibility of code: Kotlin compatibility policy says that breaking changes (e.g., a change which makes the code that used to compile fine, not compile anymore) can be introduced only in the major releases (**1.2**, **1.3**, etc.).

We believe that a lot of users could use a much faster cycle where critical compiler bug fixes arrive immediately, making the code more safe and correct. So, Kotlin 1.3 introduces the *progressive* compiler mode, which can be enabled by passing the argument `-progressive` to the compiler.

In the progressive mode, some fixes in language semantics can arrive immediately. All these fixes have two important properties:

- They preserve backward compatibility of source code with older compilers, meaning that all the code which is compilable by the progressive compiler will be compiled fine by non-progressive one.
- They only make code *safer* in some sense — e.g., some unsound smartcast can be forbidden, behavior of the generated code may be changed to be more predictable/stable, and so on.

Enabling the progressive mode can require you to rewrite some of your code, but it shouldn't be too much — all the fixes enabled under progressive are carefully handpicked, reviewed, and provided with tooling migration assistance. We expect that the progressive mode will be a nice choice for any actively maintained codebases which are updated to the latest language versions quickly.

Inline classes

Inline classes are in Alpha. They may change incompatibly and require manual migration in the future. We appreciate your feedback on it in YouTrack. See details in the reference.

Kotlin 1.3 introduces a new kind of declaration — `inline class`. Inline classes can be viewed as a restricted version of the usual classes, in particular, inline classes must have exactly one property:

```
inline class Name(val s: String)
```

The Kotlin compiler will use this restriction to aggressively optimize runtime representation of inline classes and substitute their instances with the value of the underlying property where possible removing constructor calls, GC pressure, and enabling other optimizations:

```
inline class Name(val s: String)
fun main() {
    // In the next line no constructor call happens, and
    // at the runtime 'name' contains just string "Kotlin"
    val name = Name("Kotlin")
    println(name.s)
}
```

See reference for inline classes for details.

Unsigned integers

Unsigned integers are in Beta. Their implementation is almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you will have to make.

Kotlin 1.3 introduces unsigned integer types:

- `kotlin.UByte` : an unsigned 8-bit integer, ranges from 0 to 255
- `kotlin.UShort` : an unsigned 16-bit integer, ranges from 0 to 65535
- `kotlin.UInt` : an unsigned 32-bit integer, ranges from 0 to 2^32 - 1
- `kotlin.ULong` : an unsigned 64-bit integer, ranges from 0 to 2^64 - 1

Most of the functionality of signed types are supported for unsigned counterparts too:

```
fun main() {
// You can define unsigned types using literal suffixes
val uint = 42u
val ulong = 42uL
val ubyte: UByte = 255u

// You can convert signed types to unsigned and vice versa via stdlib extensions:
val int = uint.toInt()
val byte = ubyte.toByte()
val ulong2 = byte.toULong()

// Unsigned types support similar operators:
val x = 20u + 22u
val y = 1u shl 8
val z = "128".toUByte()
val range = 1u..5u
println("ubyte: $ubyte, byte: $byte, ulong2: $ulong2")
println("x: $x, y: $y, z: $z, range: $range")
}
```

See reference for details.

@JvmDefault

`@JvmDefault` is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin targets a wide range of the Java versions, including Java 6 and Java 7, where default methods in the interfaces are not allowed. For your convenience, the Kotlin compiler works around that limitation, but this workaround isn't compatible with the `default` methods, introduced in Java 8.

This could be an issue for Java-interoperability, so Kotlin 1.3 introduces the `@JvmDefault` annotation. Methods annotated with this annotation will be generated as `default` methods for JVM:

```
interface Foo {
    // Will be generated as 'default' method
    @JvmDefault
    fun foo(): Int = 42
}
```

Warning! Annotating your API with `@JvmDefault` has serious implications on binary compatibility. Make sure to carefully read the reference page before using `@JvmDefault` in production.

Standard library

Multiplatform random

Prior to Kotlin 1.3, there was no uniform way to generate random numbers on all platforms — we had to resort to platform-specific solutions like `java.util.Random` on JVM. This release fixes this issue by introducing the class `kotlin.random.Random`, which is available on all platforms:

```
import kotlin.random.Random

fun main() {
    val number = Random.nextInt(42)  // number is in range [0, limit)
    println(number)
}
```

isNullOrEmpty and orEmpty extensions

`isNullOrEmpty` and `orEmpty` extensions for some types are already present in stdlib. The first one returns `true` if the receiver is `null` or empty, and the second one falls back to an empty instance if the receiver is `null`. Kotlin 1.3 provides similar extensions on collections, maps, and arrays of objects.

Copy elements between two existing arrays

The `array.copyInto(targetArray, targetOffset, startIndex, endIndex)` functions for the existing array types, including the unsigned arrays, make it easier to implement array-based containers in pure Kotlin.

```
fun main() {
    val sourceArr = arrayOf("k", "o", "t", "l", "i", "n")
    val targetArr = sourceArr.copyInto(arrayOfNulls<String>(6), 3, startIndex = 3, endIndex = 6)
    println(targetArr.contentToString())

    sourceArr.copyInto(targetArr, startIndex = 0, endIndex = 3)
    println(targetArr.contentToString())
}
```

associateWith

It is quite a common situation to have a list of keys and want to build a map by associating each of these keys with some value. It was possible to do it before with the `associate { it to getValue(it) }` function, but now we're introducing a more efficient and easy to explore alternative: `keys.associateWith { getValue(it) }`.

```
fun main() {
    val keys = 'a'..'f'
    val map = keys.associateWith { it.toString().repeat(5).capitalize() }
    map.forEach { println(it) }
}
```

ifEmpty and ifBlank functions

Collections, maps, object arrays, char sequences, and sequences now have an `ifEmpty` function, which allows specifying a fallback value that will be used instead of the receiver if it is empty:

```
fun main() {
    fun printAllUppercase(data: List<String>) {
        val result = data
        .filter { it.all { c -> c.isUpperCase() } }
            .ifEmpty { listOf("<no uppercase>") }
        result.forEach { println(it) }
    }

    printAllUppercase(listOf("foo", "Bar"))
    printAllUppercase(listOf("FOO", "BAR"))
}
```

Char sequences and strings in addition have an `ifBlank` extension that does the same thing as `ifEmpty` but checks for a string being all whitespace instead of empty.

```
fun main() {
    val s = "    \n"
    println(s.ifBlank { "<blank>" })
    println(s.ifBlank { null })
}
```

Sealed classes in reflection

We've added a new API to `kotlin-reflect` that can be used to enumerate all the direct subtypes of a `sealed` class, namely `KClass.sealedSubclasses`.

Smaller changes

- `Boolean` type now has companion.
- `Any?.hashCode()` extension that returns 0 for `null`.
- `Char` now provides `MIN_VALUE` and `MAX_VALUE` constants.
- `SIZE_BYTES` and `SIZE_BITS` constants in primitive type companions.

Tooling

Code style support in IDE

Kotlin 1.3 introduces support for the recommended code style in IntelliJ IDEA. Check out this page for the migration guidelines.

kotlinx.serialization

kotlinx.serialization is a library which provides multiplatform support for (de)serializing objects in Kotlin. Previously, it was a separate project, but since Kotlin 1.3, it ships with the Kotlin compiler distribution on par with the other compiler plugins. The main difference is that you don't need to manually watch out for the Serialization IDE Plugin being compatible with the Kotlin IDE plugin version you're using: now the Kotlin IDE plugin already includes serialization!

See here for details.

Even though kotlinx.serialization now ships with the Kotlin Compiler distribution, it is still considered to be an experimental feature in Kotlin 1.3.

Scripting update

Scripting is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.

Kotlin 1.3 continues to evolve and improve scripting API, introducing some experimental support for scripts customization, such as adding external properties, providing static or dynamic dependencies, and so on.

For additional details, please consult the KEEP-75.

Scratches support

Kotlin 1.3 introduces support for runnable Kotlin *scratch files*. *Scratch file* is a kotlin script file with the .kts extension that you can run and get evaluation results directly in the editor.

Consult the general Scratches documentation for details.

Compatibility guide for Kotlin 1.3.x

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.2 to Kotlin 1.3.

Basic terms

In this document we introduce several kinds of compatibility:

- *Source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *Binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *Behavioral*: a change is said to be behavioral-incompatible if one and the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (e.g. from Java) is out of the scope of this document.

Incompatible changes

Evaluation order of constructor arguments regarding <clinit> call

Issue: KT-19532

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: evaluation order with respect to class initialization is changed in 1.3

Deprecation cycle:

- <1.3: old behavior (see details in the Issue)
- >= 1.3: behavior changed, `-Xnormalize-constructor-calls=disable` can be used to temporarily revert to pre-1.3 behavior. Support for this flag is going to be removed in the next major release.

Missing getter-targeted annotations on annotation constructor parameters

Issue: KT-25287

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: getter-target annotations on annotations constructor parameters will be properly written to classfiles in 1.3

Deprecation cycle:

- <1.3: getter-target annotations on annotation constructor parameters are not applied
- >=1.3: getter-target annotations on annotation constructor parameters are properly applied and written to the generated code

Missing errors in class constructor's @get: annotations

Issue: KT-19628

Component: Core language

Incompatible change type: Source

Short summary: errors in getter-target annotations will be reported properly in 1.3

Deprecation cycle:

- <1.2: compilation errors in getter-target annotations were not reported, causing incorrect code to be compiled fine.
- 1.2.x: errors reported only by tooling, the compiler still compiles such code without any warnings
- >=1.3: errors reported by the compiler too, causing erroneous code to be rejected

Nullability assertions on access to Java types annotated with @NotNull

Issue: KT-20830

Component: Kotlin/JVM

Incompatible change type: Behavioral

Short summary: nullability assertions for Java-types annotated with not-null annotations will be generated more aggressively, causing code which passes `null` here to fail faster.

Deprecation cycle:

- <1.3: the compiler could miss such assertions when type inference was involved, allowing potential `null` propagation during compilation against binaries (see Issue for details).
- >=1.3: the compiler generates missed assertions. This can cause code which was (erroneously) passing `null`s here fail faster.
  `-XXLanguage:-StrictJavaNullabilityAssertions` can be used to temporarily return to the pre-1.3 behavior. Support for this flag will be removed in the next major release.

Unsound smartcasts on enum members

Issue: KT-20772

Component: Core language

Incompatible change type: Source

Short summary: a smartcast on a member of one enum entry will be correctly applied to only this enum entry

Deprecation cycle:

- <1.3: a smartcast on a member of one enum entry could lead to an unsound smartcast on the same member of other enum entries.
- >=1.3: smartcast will be properly applied only to the member of one enum entry.
  `-XXLanguage:-SoundSmartcastForEnumEntries` will temporarily return old behavior. Support for this flag will be removed in the next major release.

val backing field reassignment in getter

Issue: KT-16681

Components: Core language

Incompatible change type: Source

Short summary: reassignment of the backing field of `val`-property in its getter is now prohibited

Deprecation cycle:

- <1.2: Kotlin compiler allowed to modify backing field of `val` in its getter. Not only it violates Kotlin semantic, but also generates ill-behaved JVM bytecode which reassigns `final` field.
- 1.2.X: deprecation warning is reported on code which reassigns backing field of `val`
- >=1.3: deprecation warnings are elevated to errors

Array capturing before the for-loop where it is iterated

Issue: KT-21354

Component: Kotlin/JVM

Incompatible change type: Source

Short summary: if an expression in for-loop range is a local variable updated in a loop body, this change affects loop execution. This is inconsistent with iterating over other containers, such as ranges, character sequences, and collections.

Deprecation cycle:

- <1.2: described code patterns are compiled fine, but updates to local variable affect loop execution
- 1.2.X: deprecation warning reported if a range expression in a for-loop is an array-typed local variable which is assigned in a loop body
- 1.3: change behavior in such cases to be consistent with other containers

Nested classifiers in enum entries

Issue: KT-16310

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, nested classifiers (classes, object, interfaces, annotation classes, enum classes) in enum entries are prohibited

Deprecation cycle:

- <1.2: nested classifiers in enum entries are compiled fine, but may fail with exception at runtime
- 1.2.X: deprecation warnings reported on the nested classifiers
- >=1.3: deprecation warnings elevated to errors

Data class overriding copy

Issue: KT-19618

Components: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, data classes are prohibited to override `copy()`

Deprecation cycle:

- <1.2: data classes overriding `copy()` are compiled fine but may fail at runtime/expose strange behavior
- 1.2.X: deprecation warnings reported on data classes overriding `copy()`
- >=1.3: deprecation warnings elevated to errors

Inner classes inheriting Throwable that capture generic parameters from the outer class

Issue: KT-17981

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, inner classes are not allowed to inherit `Throwable`

Deprecation cycle:

- <1.2: inner classes inheriting `Throwable` are compiled fine. If such inner classes happen to capture generic parameters, it could lead to strange code patterns which fail at runtime.
- 1.2.X: deprecation warnings reported on inner classes inheriting `Throwable`
- >=1.3: deprecation warnings elevated to errors

Visibility rules regarding complex class hierarchies with companion objects

Issues: KT-21515, KT-25333

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, rules of visibility by short names are stricter for complex class hierarchies involving companion objects and nested classifiers.

Deprecation cycle:

- <1.2: old visibility rules (see Issue for details)
- 1.2.X: deprecation warnings reported on short names which are not going to be accessible anymore. Tooling suggests automated migration by adding full name.
- >=1.3: deprecation warnings elevated to errors. Offending code should add full qualifiers or explicit imports

Non-constant vararg annotation parameters

Issue: KT-23153

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, setting non-constant values as vararg annotation parameters is prohibited

Deprecation cycle:

- <1.2: the compiler allows to pass non-constant value for vararg annotation parameter, but actually drops that value during bytecode generation, leading to non-obvious behavior
- 1.2.X: deprecation warnings reported on such code patterns
- >=1.3: deprecation warnings elevated to errors

Local annotation classes

Issue: KT-23277

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 local annotation classes are not supported

Deprecation cycle:

- <1.2: the compiler compiled local annotation classes fine
- 1.2.X: deprecation warnings reported on local annotation classes
- >=1.3: deprecation warnings elevated to errors

Smartcasts on local delegated properties

Issue: KT-22517

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 smartcasts on local delegated properties are not allowed

Deprecation cycle:

- <1.2: the compiler allowed to smartcast local delegated property, which could lead to unsound smartcast in case of ill-behaved delegates
- 1.2.X: smartcasts on local delegated properries are reported as deprecated (the compiler issues warnings)
- >=1.3: deprecation warnings elevated to errors

mod operator convention

Issues: KT-24197

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 declaration of `mod` operator is prohibited, as well as calls which resolve to such declarations

Deprecation cycle:

- 1.1.X, 1.2.X: report warnings on declarations of `operator mod`, as well as on calls which resolve to it
- 1.3.X: elevate warnings to error, but still allow to resolve to `operator mod` declarations
- 1.4.X: do not resolve calls to `operator mod` anymore

Passing single element to vararg in named form

Issues: KT-20588, KT-20589. See also KT-20171

Component: Core language

Incompatible change type: Source

Short summary: in Kotlin 1.3, assigning single element to vararg is deprecated and should be replaced with consecutive spread and array construction.

Deprecation cycle:

- <1.2: assigning one value element to vararg in named form compiles fine and is treated as assigning *single* element to array, causing non-obvious behavior when assigning array to vararg
- 1.2.X: deprecation warnings are reported on such assignments, users are suggested to switch to consecutive spread and array construction.
- 1.3.X: warnings are elevated to errors
- >= 1.4: change semantic of assigning single element to vararg, making assignment of array equivalent to the assignment of a spread of an array

Retention of annotations with target EXPRESSION

Issue: KT-13762

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, only `SOURCE` retention is allowed for annotations with target `EXPRESSION`

Deprecation cycle:

- <1.2: annotations with target `EXPRESSION` and retention other than `SOURCE` are allowed, but silently ignored at use-sites
- 1.2.X: deprecation warnings are reported on declarations of such annotations
- >=1.3: warnings are elevated to errors

Annotations with target PARAMETER shouldn't be applicable to parameter's type

Issue: KT-9580

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, error about wrong annotation target will be properly reported when annotation with target `PARAMETER` is applied to parameter's type

Deprecation cycle:

- <1.2: aforementioned code patterns are compiled fine; annotations are silently ignored and not present in the bytecode
- 1.2.X: deprecation warnings are reported on such usages
- >=1.3: warnings are elevated to errors

Array.copyOfRange throws an exception when indices are out of bounds instead of enlarging the returned array

Issue: KT-19489

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, ensure that the `toIndex` argument of `Array.copyOfRange`, which represents the exclusive end of the range being copied, is not greater than the array size and throw `IllegalArgumentException` if it is.

Deprecation cycle:

- <1.3: in case `toIndex` in the invocation of `Array.copyOfRange` is greater than the array size, the missing elements in range fill be filled with `null` s, violating soundness of the Kotlin type system.
- >=1.3: check that `toIndex` is in the array bounds, and throw exception if it isn't

Progressions of ints and longs with a step of Int.MIN_VALUE and Long.MIN_VALUE are outlawed and won't be allowed to be instantiated

Issue: KT-17176

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, prohibit step value for integer progressions being the minimum negative value of its integer type (`Long` or `Int`), so that calling `IntProgression.fromClosedRange(0, 1, step = Int.MIN_VALUE)` will throw `IllegalArgumentException`

Deprecation cycle:

- <1.3: it was possible to create an `IntProgression` with `Int.MIN_VALUE` step, which yields two values `[0, -2147483648]` , which is non-obvious behavior
- >=1.3: throw `IllegalArgumentException` if the step is the minimum negative value of its integer type

Check for index overflow in operations on very long sequences

Issue: KT-16097

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, make sure `index` , `count` and similar methods do not overflow for long sequences. See the Issue for the full list of affected methods.

Deprecation cycle:

- <1.3: calling such methods on very long sequences could produce negative results due to integer overflow
- >=1.3: detect overflow in such methods and throw exception immediately

Unify split by an empty match regex result across the platforms

Issue: KT-21049

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, unify behavior of `split` method by empty match regex across all platforms

Deprecation cycle:

- <1.3: behavior of described calls is different when comparing JS, JRE 6, JRE 7 versus JRE 8+
- >=1.3: unify behavior across the platforms

Discontinued deprecated artifacts in the compiler distribution

Issue: KT-23799

Component: other

Incompatible change type: Binary

Short summary: Kotlin 1.3 discontinues the following deprecated binary artifacts:

- `kotlin-runtime`: use `kotlin-stdlib` instead
- `kotlin-stdlib-jre7/8`: use `kotlin-stdlib-jdk7/8` instead
- `kotlin-jslib` in the compiler distribution: use `kotlin-stdlib-js` instead

Deprecation cycle:

- 1.2.X: the artifacts were marked as deprecated, the compiler reported warning on usage of those artifacts
- >=1.3: the artifacts are discontinued

Annotations in stdlib

Issue: KT-21784

Component: kotlin-stdlib (JVM)

Incompatible change type: Binary

Short summary: Kotlin 1.3 removes annotations from the package `org.jetbrains.annotations` from stdlib and moves them to the separate artifacts shipped with the compiler: `annotations-13.0.jar` and `mutability-annotations-compat.jar`

Deprecation cycle:

- <1.3: annotations were shipped with the stdlib artifact
- >=1.3: annotations ship in separate artifacts

What's new in Kotlin 1.2

Released: 28 November 2017

Table of contents

- Multiplatform projects
- Other language features
- Standard library
- JVM backend
- JavaScript backend

For information about the Kotlin release cycle, see Kotlin release process.

Multiplatform projects (experimental)

Multiplatform projects are a new **experimental** feature in Kotlin 1.2, allowing you to reuse code between target platforms supported by Kotlin – JVM, JavaScript, and (in the future) Native. In a multiplatform project, you have three kinds of modules:

- A *common* module contains code that is not specific to any platform, as well as declarations without implementation of platform-dependent APIs.
- A *platform* module contains implementations of platform-dependent declarations in the common module for a specific platform, as well as other platform-dependent code.
- A regular module targets a specific platform and can either be a dependency of platform modules or depend on platform modules.

When you compile a multiplatform project for a specific platform, the code for both the common and platform-specific parts is generated.

A key feature of the multiplatform project support is the possibility to express dependencies of common code on platform-specific parts through *expected* and *actual* declarations. An *expected* declaration specifies an API (class, interface, annotation, top-level declaration etc.). An *actual* declaration is either a platform-dependent implementation of the API or a type alias referring to an existing implementation of the API in an external library. Here's an example:

In the common code:

```
// expected platform-specific API:
expect fun hello(world: String): String

fun greet() {
    // usage of the expected API:
    val greeting = hello("multiplatform world")
    println(greeting)
}

expect class URL(spec: String) {
    open fun getHost(): String
    open fun getPath(): String
}
```

In the JVM platform code:

```
actual fun hello(world: String): String =
    "Hello, $world, on the JVM platform!"

// using existing platform-specific implementation:
actual typealias URL = java.net.URL
```

See the multiplatform programming documentation for details and steps to build a multiplatform project.

Other language features

Array literals in annotations

Starting with Kotlin 1.2, array arguments for annotations can be passed with the new array literal syntax instead of the `arrayOf` function:

```
@CacheConfig(cacheNames = ["books", "default"])
public class BookRepositoryImpl {
    // ...
}
```

The array literal syntax is constrained to annotation arguments.

Lateinit top-level properties and local variables

The `lateinit` modifier can now be used on top-level properties and local variables. The latter can be used, for example, when a lambda passed as a constructor argument to one object refers to another object which has to be defined later:

```
class Node<T>(val value: T, val next: () -> Node<T>)

fun main(args: Array<String>) {
    // A cycle of three nodes:
    lateinit var third: Node<Int>

    val second = Node(2, next = { third })
    val first = Node(1, next = { second })

    third = Node(3, next = { first })

    val nodes = generateSequence(first) { it.next() }
    println("Values in the cycle: ${nodes.take(7).joinToString { it.value.toString() }}, ...")
}
```

Check whether a lateinit var is initialized

You can now check whether a lateinit var has been initialized using `isInitialized` on the property reference:

```
class Foo {
    lateinit var lateinitVar: String

    fun initializationLogic() {
        println("isInitialized before assignment: " + this::lateinitVar.isInitialized)
        lateinitVar = "value"
        println("isInitialized after assignment: " + this::lateinitVar.isInitialized)
    }
}

fun main(args: Array<String>) {
        Foo().initializationLogic()
}
```

Inline functions with default functional parameters

Inline functions are now allowed to have default values for their inlined functional parameters:

```
inline fun <E> Iterable<E>.strings(transform: (E) -> String = { it.toString() }) =
    map { transform(it) }

val defaultStrings = listOf(1, 2, 3).strings()
val customStrings = listOf(1, 2, 3).strings { "($it)" }

fun main(args: Array<String>) {
    println("defaultStrings = $defaultStrings")
    println("customStrings = $customStrings")
}
```

Information from explicit casts is used for type inference

The Kotlin compiler can now use information from type casts in type inference. If you're calling a generic method that returns a type parameter `T` and casting the return value to a specific type `Foo`, the compiler now understands that `T` for this call needs to be bound to the type `Foo`.

This is particularly important for Android developers, since the compiler can now correctly analyze generic `findViewById` calls in Android API level 26:

```
val button = findViewById(R.id.button) as Button
```

Smart cast improvements

When a variable is assigned from a safe call expression and checked for null, the smart cast is now applied to the safe call receiver as well:

```
fun countFirst(s: Any): Int {
    val firstChar = (s as? CharSequence)?.firstOrNull()
    if (firstChar ≠ null)
    return s.count { it == firstChar } // s: Any is smart cast to CharSequence

    val firstItem = (s as? Iterable<*>)?.firstOrNull()
    if (firstItem ≠ null)
    return s.count { it == firstItem } // s: Any is smart cast to Iterable<*>
    return -1
}

fun main(args: Array<String>) {
  val string = "abacaba"
  val countInString = countFirst(string)
  println("called on \"$string\": $countInString")

  val list = listOf(1, 2, 3, 1, 2)
  val countInList = countFirst(list)
  println("called on $list: $countInList")
}
```

Also, smart casts in a lambda are now allowed for local variables that are only modified before the lambda:

```
fun main(args: Array<String>) {
    val flag = args.size == 0
    var x: String? = null
    if (flag) x = "Yahoo!"

    run {
        if (x ≠ null) {
            println(x.length) // x is smart cast to String
        }
    }
}
```

Support for ::foo as a shorthand for this::foo

A bound callable reference to a member of `this` can now be written without explicit receiver, `::foo` instead of `this::foo`. This also makes callable references more convenient to use in lambdas where you refer to a member of the outer receiver.

Breaking change: sound smart casts after try blocks

Earlier, Kotlin used assignments made inside a `try` block for smart casts after the block, which could break type- and null-safety and lead to runtime failures. This release fixes this issue, making the smart casts more strict, but breaking some code that relied on such smart casts.

To switch to the old smart casts behavior, pass the fallback flag `-Xlegacy-smart-cast-after-try` as the compiler argument. It will become deprecated in Kotlin 1.3.

Deprecation: data classes overriding copy

When a data class derived from a type that already had the `copy` function with the same signature, the `copy` implementation generated for the data class used the defaults from the supertype, leading to counter-intuitive behavior, or failed at runtime if there were no default parameters in the supertype.

Inheritance that leads to a `copy` conflict has become deprecated with a warning in Kotlin 1.2 and will be an error in Kotlin 1.3.

Deprecation: nested types in enum entries

Inside enum entries, defining a nested type that is not an `inner class` has been deprecated due to issues in the initialization logic. This causes a warning in Kotlin 1.2 and will become an error in Kotlin 1.3.

Deprecation: single named argument for vararg

For consistency with array literals in annotations, passing a single item for a vararg parameter in the named form (`foo(items = i)`) has been deprecated. Please use the spread operator with the corresponding array factory functions:

```
foo(items = *arrayOf(1))
```

There is an optimization that removes redundant arrays creation in such cases, which prevents performance degradation. The single-argument form produces warnings in Kotlin 1.2 and is to be dropped in Kotlin 1.3.

Deprecation: inner classes of generic classes extending Throwable

Inner classes of generic types that inherit from `Throwable` could violate type-safety in a throw-catch scenario and thus have been deprecated, with a warning in Kotlin 1.2 and an error in Kotlin 1.3.

Deprecation: mutating backing field of a read-only property

Mutating the backing field of a read-only property by assigning `field = ...` in the custom getter has been deprecated, with a warning in Kotlin 1.2 and an error in Kotlin 1.3.

Standard library

Kotlin standard library artifacts and split packages

The Kotlin standard library is now fully compatible with the Java 9 module system, which forbids split packages (multiple jar files declaring classes in the same package). In order to support that, new artifacts `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` are introduced, which replace the old `kotlin-stdlib-jre7` and `kotlin-stdlib-jre8`.

The declarations in the new artifacts are visible under the same package names from the Kotlin point of view, but have different package names for Java. Therefore, switching to the new artifacts will not require any changes to your source code.

Another change made to ensure compatibility with the new module system is removing the deprecated declarations in the `kotlin.reflect` package from the `kotlin-reflect` library. If you were using them, you need to switch to using the declarations in the `kotlin.reflect.full` package, which is supported since Kotlin 1.1.

windowed, chunked, zipWithNext

New extensions for `Iterable<T>`, `Sequence<T>`, and `CharSequence` cover such use cases as buffering or batch processing (`chunked`), sliding window and computing sliding average (`windowed`), and processing pairs of subsequent items (`zipWithNext`):

```
fun main(args: Array<String>) {
    val items = (1..9).map { it * it }

    val chunkedIntoLists = items.chunked(4)
    val points3d = items.chunked(3) { (x, y, z) -> Triple(x, y, z) }
    val windowed = items.windowed(4)
    val slidingAverage = items.windowed(4) { it.average() }
    val pairwiseDifferences = items.zipWithNext { a, b -> b - a }

    println("items: $items\n")

    println("chunked into lists: $chunkedIntoLists")
    println("3D points: $points3d")
    println("windowed by 4: $windowed")
    println("sliding average by 4: $slidingAverage")
    println("pairwise differences: $pairwiseDifferences")
}
```

fill, replaceAll, shuffle/shuffled

A set of extension functions was added for manipulating lists: `fill`, `replaceAll` and `shuffle` for `MutableList`, and `shuffled` for read-only `List`:

```
fun main(args: Array<String>) {
    val items = (1..5).toMutableList()

    items.shuffle()
    println("Shuffled items: $items")

    items.replaceAll { it * 2 }
    println("Items doubled: $items")

    items.fill(5)
    println("Items filled with 5: $items")
}
```

Math operations in kotlin-stdlib

Satisfying the longstanding request, Kotlin 1.2 adds the `kotlin.math` API for math operations that is common for JVM and JS and contains the following:

- Constants: `PI` and `E`
- Trigonometric: `cos`, `sin`, `tan` and inverse of them: `acos`, `asin`, `atan`, `atan2`
- Hyperbolic: `cosh`, `sinh`, `tanh` and their inverse: `acosh`, `asinh`, `atanh`
- Exponentation: `pow` (an extension function), `sqrt`, `hypot`, `exp`, `expm1`
- Logarithms: `log`, `log2`, `log10`, `ln`, `ln1p`
- Rounding:
  - `ceil`, `floor`, `truncate`, `round` (half to even) functions
  - `roundToInt`, `roundToLong` (half to integer) extension functions
- Sign and absolute value:
  - `abs` and `sign` functions
  - `absoluteValue` and `sign` extension properties
  - `withSign` extension function
- `max` and `min` of two values
- Binary representation:
  - `ulp` extension property
  - `nextUp`, `nextDown`, `nextTowards` extension functions
  - `toBits`, `toRawBits`, `Double.fromBits` (these are in the `kotlin` package)

The same set of functions (but without constants) is also available for `Float` arguments.

Operators and conversions for BigInteger and BigDecimal

Kotlin 1.2 introduces a set of functions for operating with `BigInteger` and `BigDecimal` and creating them from other numeric types. These are:

- `toBigInteger` for `Int` and `Long`
- `toBigDecimal` for `Int`, `Long`, `Float`, `Double`, and `BigInteger`
- Arithmetic and bitwise operator functions:
  - Binary operators `+`, `-`, `*`, `/`, `%` and infix functions `and`, `or`, `xor`, `shl`, `shr`
  - Unary operators `-`, `++`, `--`, and a function `inv`

Floating point to bits conversions

New functions were added for converting `Double` and `Float` to and from their bit representations:

- `toBits` and `toRawBits` returning `Long` for `Double` and `Int` for `Float`
- `Double.fromBits` and `Float.fromBits` for creating floating point numbers from the bit representation

Regex is now serializable

The `kotlin.text.Regex` class has become `Serializable` and can now be used in serializable hierarchies.

Closeable.use calls Throwable.addSuppressed if available

The `Closeable.use` function calls `Throwable.addSuppressed` when an exception is thrown during closing the resource after some other exception.

To enable this behavior you need to have `kotlin-stdlib-jdk7` in your dependencies.

JVM backend

Constructor calls normalization

Ever since version 1.0, Kotlin supported expressions with complex control flow, such as try-catch expressions and inline function calls. Such code is valid according to the Java Virtual Machine specification. Unfortunately, some bytecode processing tools do not handle such code quite well when such expressions are present in the arguments of constructor calls.

To mitigate this problem for the users of such bytecode processing tools, we've added a command-line compiler option (`-Xnormalize-constructor-calls=MODE`) that tells the compiler to generate more Java-like bytecode for such constructs. Here `MODE` is one of:

- `disable` (default) – generate bytecode in the same way as in Kotlin 1.0 and 1.1.
- `enable` – generate Java-like bytecode for constructor calls. This can change the order in which the classes are loaded and initialized.
- `preserve-class-initialization` – generate Java-like bytecode for constructor calls, ensuring that the class initialization order is preserved. This can affect overall performance of your application; use it only if you have some complex state shared between multiple classes and updated on class initialization.

The "manual" workaround is to store the values of sub-expressions with control flow in variables, instead of evaluating them directly inside the call arguments. It's similar to `-Xnormalize-constructor-calls=enable`.

Java-default method calls

Before Kotlin 1.2, interface members overriding Java-default methods while targeting JVM 1.6 produced a warning on super calls: `Super calls to Java default methods are deprecated in JVM target 1.6. Recompile with '-jvm-target 1.8'`. In Kotlin 1.2, there's an **error** instead, thus requiring any such code to be compiled with JVM target 1.8.

Breaking change: consistent behavior of x.equals(null) for platform types

Calling `x.equals(null)` on a platform type that is mapped to a Java primitive (`Int!`, `Boolean!`, `Short`!, `Long!`, `Float!`, `Double!`, `Char!`) incorrectly returned `true` when `x` was null. Starting with Kotlin 1.2, calling `x.equals(...)` on a null value of a platform type **throws an NPE** (but `x == ...` does not).

To return to the pre-1.2 behavior, pass the flag `-Xno-exception-on-explicit-equals-for-boxed-null` to the compiler.

Breaking change: fix for platform null escaping through an inlined extension receiver

Inline extension functions that were called on a null value of a platform type did not check the receiver for null and would thus allow null to escape into the other code. Kotlin 1.2 forces this check at the call sites, throwing an exception if the receiver is null.

To switch to the old behavior, pass the fallback flag `-Xno-receiver-assertions` to the compiler.

JavaScript backend

TypedArrays support enabled by default

The JS typed arrays support that translates Kotlin primitive arrays, such as `IntArray`, `DoubleArray`, into JavaScript typed arrays, that was previously an opt-in feature, has been enabled by default.

Tools

Warnings as errors

The compiler now provides an option to treat all warnings as errors. Use `-Werror` on the command line, or the following Gradle snippet:

```
compileKotlin {
    kotlinOptions.allWarningsAsErrors = true
}
```

What's new in Kotlin 1.1

Released: 15 February 2016

Table of contents

- Coroutines
- Other language features
- Standard library
- JVM backend
- JavaScript backend

For information about the Kotlin release cycle, see Kotlin release process.

JavaScript

Starting with Kotlin 1.1, the JavaScript target is no longer considered experimental. All language features are supported, and there are many new tools for integration with the frontend development environment. See below for a more detailed list of changes.

Coroutines (experimental)

The key new feature in Kotlin 1.1 is *coroutines*, bringing the support of `async` / `await`, `yield`, and similar programming patterns. The key feature of Kotlin's design is that the implementation of coroutine execution is part of the libraries, not the language, so you aren't bound to any specific programming paradigm or concurrency library.

A coroutine is effectively a light-weight thread that can be suspended and resumed later. Coroutines are supported through *suspending functions*: a call to such a function can potentially suspend a coroutine, and to start a new coroutine we usually use an anonymous suspending functions (i.e. suspending lambdas).

Let's look at `async` / `await` which is implemented in an external library, kotlinx.coroutines:

```
// runs the code in the background thread pool
fun asyncOverlay() = async(CommonPool) {
    // start two async operations
    val original = asyncLoadImage("original")
    val overlay = asyncLoadImage("overlay")
    // and then apply overlay to both results
    applyOverlay(original.await(), overlay.await())
}

// launches new coroutine in UI context
launch(UI) {
    // wait for async overlay to complete
    val image = asyncOverlay().await()
    // and then show it in UI
    showImage(image)
}
```

Here, `async { ... }` starts a coroutine and, when we use `await()`, the execution of the coroutine is suspended while the operation being awaited is executed, and is resumed (possibly on a different thread) when the operation being awaited completes.

The standard library uses coroutines to support *lazily generated sequences* with `yield` and `yieldAll` functions. In such a sequence, the block of code that returns sequence elements is suspended after each element has been retrieved, and resumed when the next element is requested. Here's an example:

```
import kotlin.coroutines.experimental.*

fun main(args: Array<String>) {
    val seq = buildSequence {
      for (i in 1..5) {
          // yield a square of i
          yield(i * i)
      }
      // yield a range
      yieldAll(26..28)
    }

    // print the sequence
    println(seq.toList())
}
```

Run the code above to see the result. Feel free to edit it and run again!

For more information, please refer to the coroutines documentation and tutorial.

Note that coroutines are currently considered an **experimental feature**, meaning that the Kotlin team is not committing to supporting the backwards compatibility of this feature after the final 1.1 release.

Other language features

Type aliases

A type alias allows you to define an alternative name for an existing type. This is most useful for generic types such as collections, as well as for function types. Here is an example:

```
typealias OscarWinners = Map<String, String>

fun countLaLaLand(oscarWinners: OscarWinners) =
        oscarWinners.count { it.value.contains("La La Land") }

// Note that the type names (initial and the type alias) are interchangeable:
fun checkLaLaLandIsTheBestMovie(oscarWinners: Map<String, String>) =
        oscarWinners["Best picture"] == "La La Land"

fun oscarWinners(): OscarWinners {
    return mapOf(
            "Best song" to "City of Stars (La La Land)",
            "Best actress" to "Emma Stone (La La Land)",
            "Best picture" to "Moonlight" /* ... */)
}

fun main(args: Array<String>) {
    val oscarWinners = oscarWinners()

    val laLaLandAwards = countLaLaLand(oscarWinners)
    println("LaLaLandAwards = $laLaLandAwards (in our small example), but actually it's 6.")

    val laLaLandIsTheBestMovie = checkLaLaLandIsTheBestMovie(oscarWinners)
    println("LaLaLandIsTheBestMovie = $laLaLandIsTheBestMovie")
}
```

See the type aliases documentation and KEEP for more details.

Bound callable references

You can now use the `::` operator to get a member reference pointing to a method or property of a specific object instance. Previously this could only be expressed with a lambda. Here's an example:

```
val numberRegex = "\\d+".toRegex()
val numbers = listOf("abc", "123", "456").filter(numberRegex::matches)

fun main(args: Array<String>) {
    println("Result is $numbers")
}
```

Read the documentation and KEEP for more details.

Sealed and data classes

Kotlin 1.1 removes some of the restrictions on sealed and data classes that were present in Kotlin 1.0. Now you can define subclasses of a top-level sealed class on the top level in the same file, and not just as nested classes of the sealed class. Data classes can now extend other classes. This can be used to define a hierarchy of expression classes nicely and cleanly:

```
sealed class Expr

data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()

fun eval(expr: Expr): Double = when (expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
}
val e = eval(Sum(Const(1.0), Const(2.0)))

fun main(args: Array<String>) {
    println("e is $e") // 3.0
}
```

Read the sealed classes documentation or KEEPs for sealed class and data class for more detail.

Destructuring in lambdas

You can now use the destructuring declaration syntax to unpack the arguments passed to a lambda. Here's an example:

```
fun main(args: Array<String>) {
    val map = mapOf(1 to "one", 2 to "two")
    // before
    println(map.mapValues { entry ->
      val (key, value) = entry
      "$key -> $value!"
    })
    // now
    println(map.mapValues { (key, value) -> "$key -> $value!" })
}
```

Read the destructuring declarations documentation and KEEP for more details.

Underscores for unused parameters

For a lambda with multiple parameters, you can use the `_` character to replace the names of the parameters you don't use:

```
fun main(args: Array<String>) {
    val map = mapOf(1 to "one", 2 to "two")

    map.forEach { _, value -> println("$value!") }
}
```

This also works in destructuring declarations:

```
data class Result(val value: Any, val status: String)

fun getResult() = Result(42, "ok").also { println("getResult() returns $it") }

fun main(args: Array<String>) {
    val (_, status) = getResult()
    println("status is '$status'")
}
```

Read the KEEP for more details.

Underscores in numeric literals

Just as in Java 8, Kotlin now allows to use underscores in numeric literals to separate groups of digits:

```
val oneMillion = 1_000_000
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010

fun main(args: Array<String>) {
    println(oneMillion)
    println(hexBytes.toString(16))
    println(bytes.toString(2))
}
```

Read the KEEP for more details.

Shorter syntax for properties

For properties with the getter defined as an expression body, the property type can now be omitted:

```
    data class Person(val name: String, val age: Int) {
    val isAdult get() = age ⩾ 20 // Property type inferred to be 'Boolean'
}
fun main(args: Array<String>) {
    val akari = Person("Akari", 26)
    println("$akari.isAdult = ${akari.isAdult}")
}
```

Inline property accessors

You can now mark property accessors with the `inline` modifier if the properties don't have a backing field. Such accessors are compiled in the same way as inline functions.

```
public val <T> List<T>.lastIndex: Int
    inline get() = this.size - 1

fun main(args: Array<String>) {
    val list = listOf('a', 'b')
    // the getter will be inlined
    println("Last index of $list is ${list.lastIndex}")
}
```

You can also mark the entire property as `inline` - then the modifier is applied to both accessors.

Read the inline functions documentation and KEEP for more details.

Local delegated properties

You can now use the delegated property syntax with local variables. One possible use is defining a lazily evaluated local variable:

```
import java.util.Random

fun needAnswer() = Random().nextBoolean()

fun main(args: Array<String>) {
    val answer by lazy {
        println("Calculating the answer...")
        42
    }
    if (needAnswer()) {                    // returns the random value
        println("The answer is $answer.")   // answer is calculated at this point
    }
    else {
        println("Sometimes no answer is the answer...")
    }
}
```

Read the KEEP for more details.

Interception of delegated property binding

For delegated properties, it is now possible to intercept delegate to property binding using the `provideDelegate` operator. For example, if we want to check the property name before binding, we can write something like this:

```
class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(thisRef: MyUI, prop: KProperty<*>): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        ... // property creation
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ... }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ... }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

The `provideDelegate` method will be called for each property during the creation of a `MyUI` instance, and it can perform the necessary validation right away.

Read the delegated properties documentation for more details.

Generic enum value access

It is now possible to enumerate the values of an enum class in a generic way.

```
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}

fun main(args: Array<String>) {
    printAllValues<RGB>() // prints RED, GREEN, BLUE
}
```

Scope control for implicit receivers in DSLs

The `@DslMarker` annotation allows to restrict the use of receivers from outer scopes in a DSL context. Consider the canonical HTML builder example:

```
table {
    tr {
        td { + "Text" }
    }
}
```

In Kotlin 1.0, code in the lambda passed to `td` has access to three implicit receivers: the one passed to `table`, to `tr` and to `td`. This allows you to call methods that make no sense in the context - for example to call `tr` inside `td` and thus to put a `<tr>` tag in a `<td>`.

In Kotlin 1.1, you can restrict that, so that only methods defined on the implicit receiver of `td` will be available inside the lambda passed to `td`. You do that by defining your annotation marked with the `@DslMarker` meta-annotation and applying it to the base class of the tag classes.

Read the type safe builders documentation and KEEP for more details.

rem operator

The `mod` operator is now deprecated, and `rem` is used instead. See this issue for motivation.

Standard library

String to number conversions

There is a bunch of new extensions on the String class to convert it to a number without throwing an exception on invalid number: `String.toIntOrNull(): Int?`, `String.toDoubleOrNull(): Double?` etc.

```
val port = System.getenv("PORT")?.toIntOrNull() ?: 80
```

Also integer conversion functions, like `Int.toString()`, `String.toInt()`, `String.toIntOrNull()`, each got an overload with `radix` parameter, which allows to specify the base of conversion (2 to 36).

onEach()

`onEach` is a small, but useful extension function for collections and sequences, which allows to perform some action, possibly with side-effects, on each element of the collection/sequence in a chain of operations. On iterables it behaves like `forEach` but also returns the iterable instance further. And on sequences it returns a wrapping sequence, which applies the given action lazily as the elements are being iterated.

```
inputDir.walk()
        .filter { it.isFile && it.name.endsWith(".txt") }
        .onEach { println("Moving $it to $outputDir") }
        .forEach { moveFile(it, File(outputDir, it.toRelativeString(inputDir))) }
```

also(), takeIf(), and takeUnless()

These are three general-purpose extension functions applicable to any receiver.

`also` is like `apply` : it takes the receiver, does some action on it, and returns that receiver. The difference is that in the block inside `apply` the receiver is available as `this` , while in the block inside `also` it's available as `it` (and you can give it another name if you want). This comes handy when you do not want to shadow `this` from the outer scope:

```
class Block {
    lateinit var content: String
}

fun Block.copy() = Block().also {
    it.content = this.content
}

// using 'apply' instead
fun Block.copy1() = Block().apply {
    this.content = this@copy1.content
}

fun main(args: Array<String>) {
    val block = Block().apply { content = "content" }
    val copy = block.copy()
    println("Testing the content was copied:")
    println(block.content == copy.content)
}
```

`takeIf` is like `filter` for a single value. It checks whether the receiver meets the predicate, and returns the receiver, if it does or `null` if it doesn't. Combined with an elvis operator (?:) and early returns it allows writing constructs like:

```
val outDirFile = File(outputDir.path).takeIf { it.exists() } ?: return false
// do something with existing outDirFile
```

```
fun main(args: Array<String>) {
    val input = "Kotlin"
    val keyword = "in"

    val index = input.indexOf(keyword).takeIf { it ≥ 0 } ?: error("keyword not found")
    // do something with index of keyword in input string, given that it's found

    println("'$keyword' was found in '$input'")
    println(input)
    println(" ".repeat(index) + "^")
}
```

`takeUnless` is the same as `takeIf` , but it takes the inverted predicate. It returns the receiver when it *doesn't* meet the predicate and `null` otherwise. So one of the examples above could be rewritten with `takeUnless` as following:

```
val index = input.indexOf(keyword).takeUnless { it < 0 } ?: error("keyword not found")
```

It is also convenient to use when you have a callable reference instead of the lambda:

```
private fun testTakeUnless(string: String) {
    val result = string.takeUnless(String::isEmpty)

    println("string = \"$string\"; result = \"$result\"")
}

fun main(args: Array<String>) {
    testTakeUnless("")
    testTakeUnless("abc")
}
```

groupingBy()

This API can be used to group a collection by key and fold each group simultaneously. For example, it can be used to count the number of words starting with each letter:

```
fun main(args: Array<String>) {
    val words = "one two three four five six seven eight nine ten".split(' ')
    val frequencies = words.groupingBy { it.first() }.eachCount()
    println("Counting first letters: $frequencies.")

    // The alternative way that uses 'groupBy' and 'mapValues' creates an intermediate map,
    // while 'groupingBy' way counts on the fly.
    val groupBy = words.groupBy { it.first() }.mapValues { (_, list) -> list.size }
    println("Comparing the result with using 'groupBy': ${groupBy == frequencies}.")
}
```

Map.toMap() and Map.toMutableMap()

These functions can be used for easy copying of maps:

```
class ImmutablePropertyBag(map: Map<String, Any>) {
    private val mapCopy = map.toMap()
}
```

Map.minus(key)

The operator `plus` provides a way to add key-value pair(s) to a read-only map producing a new map, however there was not a simple way to do the opposite: to remove a key from the map you have to resort to less straightforward ways to like `Map.filter()` or `Map.filterKeys()`. Now the operator `minus` fills this gap. There are 4 overloads available: for removing a single key, a collection of keys, a sequence of keys and an array of keys.

```
fun main(args: Array<String>) {
    val map = mapOf("key" to 42)
    val emptyMap = map - "key"

    println("map: $map")
    println("emptyMap: $emptyMap")
}
```

minOf() and maxOf()

These functions can be used to find the lowest and greatest of two or three given values, where values are primitive numbers or `Comparable` objects. There is also an overload of each function that take an additional `Comparator` instance if you want to compare objects that are not comparable themselves.

```
fun main(args: Array<String>) {
    val list1 = listOf("a", "b")
    val list2 = listOf("x", "y", "z")
    val minSize = minOf(list1.size, list2.size)
    val longestList = maxOf(list1, list2, compareBy { it.size })

    println("minSize = $minSize")
    println("longestList = $longestList")
}
```

Array-like List instantiation functions

Similar to the `Array` constructor, there are now functions that create `List` and `MutableList` instances and initialize each element by calling a lambda:

```
fun main(args: Array<String>) {
    val squares = List(10) { index -> index * index }
    val mutable = MutableList(10) { 0 }

    println("squares: $squares")
    println("mutable: $mutable")
}
```

Map.getValue()

This extension on `Map` returns an existing value corresponding to the given key or throws an exception, mentioning which key was not found. If the map was produced with `withDefault`, this function will return the default value instead of throwing an exception.

```
fun main(args: Array<String>) {
//sampleStart
    val map = mapOf("key" to 42)
    // returns non-nullable Int value 42
    val value: Int = map.getValue("key")

    val mapWithDefault = map.withDefault { k -> k.length }
    // returns 4
    val value2 = mapWithDefault.getValue("key2")

    // map.getValue("anotherKey") // ← this will throw NoSuchElementException

    println("value is $value")
    println("value2 is $value2")
}
```

Abstract collections

These abstract classes can be used as base classes when implementing Kotlin collection classes. For implementing read-only collections there are `AbstractCollection`, `AbstractList`, `AbstractSet` and `AbstractMap`, and for mutable collections there are `AbstractMutableCollection`, `AbstractMutableList`

, `AbstractMutableSet` and `AbstractMutableMap`. On JVM, these abstract mutable collections inherit most of their functionality from JDK's abstract collections.

Array manipulation functions

The standard library now provides a set of functions for element-by-element operations on arrays: comparison (`contentEquals` and `contentDeepEquals`), hash code calculation (`contentHashCode` and `contentDeepHashCode`), and conversion to a string (`contentToString` and `contentDeepToString`). They're supported both for the JVM (where they act as aliases for the corresponding functions in `java.util.Arrays`) and for JS (where the implementation is provided in the Kotlin standard library).

```
fun main(args: Array<String>) {
    val array = arrayOf("a", "b", "c")
    println(array.toString())  // JVM implementation: type-and-hash gibberish
    println(array.contentToString())  // nicely formatted as list
}
```

JVM Backend

Java 8 bytecode support

Kotlin has now the option of generating Java 8 bytecode (`-jvm-target 1.8` command line option or the corresponding options in Maven/Gradle). For now this doesn't change the semantics of the bytecode (in particular, default methods in interfaces and lambdas are generated exactly as in Kotlin 1.0), but we plan to make further use of this later.

Java 8 standard library support

There are now separate versions of the standard library supporting the new JDK APIs added in Java 7 and 8. If you need access to the new APIs, use `kotlin-stdlib-jre7` and `kotlin-stdlib-jre8` maven artifacts instead of the standard `kotlin-stdlib` . These artifacts are tiny extensions on top of `kotlin-stdlib` and they bring it to your project as a transitive dependency.

Parameter names in the bytecode

Kotlin now supports storing parameter names in the bytecode. This can be enabled using the `-java-parameters` command line option.

Constant inlining

The compiler now inlines values of `const val` properties into the locations where they are used.

Mutable closure variables

The box classes used for capturing mutable closure variables in lambdas no longer have volatile fields. This change improves performance, but can lead to new race conditions in some rare usage scenarios. If you're affected by this, you need to provide your own synchronization for accessing the variables.

javax.script support

Kotlin now integrates with the javax.script API (JSR-223). The API allows to evaluate snippets of code at runtime:

```
val engine = ScriptEngineManager().getEngineByExtension("kts")!!
engine.eval("val x = 3")
println(engine.eval("x + 2"))  // Prints out 5
```

See here for a larger example project using the API.

kotlin.reflect.full

To prepare for Java 9 support, the extension functions and properties in the `kotlin-reflect.jar` library have been moved to the package `kotlin.reflect.full`. The names in the old package (`kotlin.reflect`) are deprecated and will be removed in Kotlin 1.2. Note that the core reflection interfaces (such as `KClass`) are part of the Kotlin standard library, not `kotlin-reflect`, and are not affected by the move.

JavaScript backend

Unified standard library

A much larger part of the Kotlin standard library can now be used from code compiled to JavaScript. In particular, key classes such as collections (`ArrayList`, `HashMap` etc.), exceptions (`IllegalArgumentException` etc.) and a few others (`StringBuilder`, `Comparator`) are now defined under the `kotlin` package. On the JVM, the names are type aliases for the corresponding JDK classes, and on the JS, the classes are implemented in the Kotlin standard library.

Better code generation

JavaScript backend now generates more statically checkable code, which is friendlier to JS code processing tools, like minifiers, optimisers, linters, etc.

The external modifier

If you need to access a class implemented in JavaScript from Kotlin in a typesafe way, you can write a Kotlin declaration using the `external` modifier. (In Kotlin 1.0, the `@native` annotation was used instead.) Unlike the JVM target, the JS one permits to use external modifier with classes and properties. For example, here's how you can declare the DOM `Node` class:

```
external class Node {
    val firstChild: Node

    fun appendChild(child: Node): Node

    fun removeChild(child: Node): Node

    // etc
}
```

Improved import handling

You can now describe declarations which should be imported from JavaScript modules more precisely. If you add the `@JsModule("<module-name>")` annotation on an external declaration it will be properly imported to a module system (either CommonJS or AMD) during the compilation. For example, with CommonJS the declaration will be imported via `require(...)` function. Additionally, if you want to import a declaration either as a module or as a global JavaScript object, you can use the `@JsNonModule` annotation.

For example, here's how you can import JQuery into a Kotlin module:

```
external interface JQuery {
    fun toggle(duration: Int = definedExternally): JQuery
    fun click(handler: (Event) -> Unit): JQuery
}

@JsModule("jquery")
@JsNonModule
@JsName("$")
external fun jquery(selector: String): JQuery
```

In this case, JQuery will be imported as a module named `jquery`. Alternatively, it can be used as a $-object, depending on what module system Kotlin compiler is configured to use.

You can use these declarations in your application like this:

```
fun main(args: Array<String>) {
    jquery(".toggle-button").click {
        jquery(".toggle-panel").toggle(300)
    }
}
```

Participate in the Kotlin Early Access Preview

You can participate in the Kotlin Early Access Preview (EAP) to try out the latest Kotlin features before they are released.

Before every language (*2.x.0*) and tooling (*2.x.20*) release, we ship Early Access Preview (EAP) builds for you to test in real projects and share early feedback. Kotlin EAP builds typically include the following stages:

EAP build	Description
Beta1	Introduces the first set of upcoming features, improvements, and other significant changes. Gives you an early opportunity to evaluate new functionality and share feedback.
Beta2	Usually adds more features and refinements based on the feedback we've received. Is feature-complete and continues the preview of the upcoming release, further maturing previously introduced features.
RC	The first release candidate. Focuses on stabilizing the changes delivered in Beta1 and Beta2, as well as fixing regressions found during testing.

EAP build	Description
RC2	Includes important fixes to finalize the release and confirm readiness.

We'd appreciate it if you report any bugs you find to our issue tracker, YouTrack. In most cases, we can fix them before the final release, so you won't need to wait for the next Kotlin release to see your issues resolved.

By participating in the Early Access Preview and reporting bugs, you contribute to Kotlin and help us make it better for everyone in the growing Kotlin community.

If you have any questions or want to join the discussion, you're welcome to join the #eap channel in Kotlin Slack. In this channel, you can also get notifications about new EAP builds.

Configure your project for the Kotlin EAP version

> By participating in the EAP, you expressly acknowledge that the EAP version may not be reliable, may not work as intended, and may contain errors.
>
> We don't guarantee compatibility between EAP and final versions of the same release.

How the EAP can help you be more productive with Kotlin

- **Prepare for the Stable release**. If you work on a complex multimodule project, participating in the EAP may streamline your experience when you adopt the Stable release version. The sooner you update to the Stable version, the sooner you can take advantage of its performance improvements and new language features.

  The migration of huge and complex projects might take a while, not only because of their size, but also because some specific use cases may not have been covered by the Kotlin team yet. By participating in the EAP and continuously testing new versions of Kotlin, you can provide us with early feedback about your specific use cases. This will help us address as many issues as possible and ensure you can safely update to the Stable version when it's released. Check out how Slack benefits from testing Android, Kotlin, and Gradle pre-release versions.

- **Keep your library up-to-date**. If you're a library author, updating to the new Kotlin version is extremely important. Using older versions could block your users from updating Kotlin in their projects. Working with EAP versions allows you to support the latest Kotlin versions in your library almost immediately with the Stable release, which makes your users happier and your library more popular.

- **Share the experience**. If you're a Kotlin enthusiast and enjoy contributing to the Kotlin ecosystem by creating educational content, trying new features in the Kotlin EAP allows you to be among the first to share the experience of using the new cool features with the community.

Build details

No preview versions are currently available.

Configure your build for EAP

To configure your build to use the EAP version of Kotlin, you need to:

-

Specify the EAP version of Kotlin. Available EAP versions are listed here.

- Change the versions of dependencies to EAP ones. The EAP version of Kotlin may not work with the libraries of the previously released version.

The following procedures describe how to configure your build in Gradle and Maven:

- Configure in Gradle
- Configure in Maven

Configure in Gradle

This section describes how you can:

- Adjust the Kotlin version
- Adjust versions in dependencies

Adjust the Kotlin version

In the `plugins` block within `build.gradle(.kts)`, change the `KOTLIN-EAP-VERSION` to the actual EAP version, such as `2.4.20-RC3`. Available EAP versions are listed here.

Alternatively, you can specify the EAP version in the `pluginManagement` block in `settings.gradle(.kts)` – see Gradle documentation for details.

Here is an example for the Multiplatform project.

Kotlin

```
plugins {
    java
    kotlin("multiplatform") version "KOTLIN-EAP-VERSION"
}

repositories {
    mavenCentral()
}
```

Groovy

```
plugins {
    id 'java'
    id 'org.jetbrains.kotlin.multiplatform' version 'KOTLIN-EAP-VERSION'
}

repositories {
    mavenCentral()
}
```

Adjust versions in dependencies

If you use kotlinx libraries in your project, your versions of the libraries may not be compatible with the EAP version of Kotlin.

To resolve this issue, you need to specify the version of a compatible library in dependencies. For a list of compatible libraries, see EAP build details.

> In most cases we create libraries only for the first EAP version of a specific release and these libraries work with the subsequent EAP versions for this release.
>
> If there are incompatible changes in next EAP versions, we release a new version of the library.

Here is an example.

For the **kotlinx.coroutines** library, add the version number – `1.11.0` – that is compatible with `2.4.20-RC3`.

Kotlin

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0")
}
```

Groovy

```
dependencies {
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.11.0"
}
```

Configure in Maven

In the sample Maven project definition, replace `KOTLIN-EAP-VERSION` with the actual version, such as `2.4.20-RC3`. Available EAP versions are listed here.

```
<project ...>
    <properties>
        <kotlin.version>KOTLIN-EAP-VERSION</kotlin.version>
    </properties>

    <repositories>
        <repository>
           <id>mavenCentral</id>
           <url>https://repo1.maven.org/maven2/</url>
        </repository>
    </repositories>

    <pluginRepositories>
       <pluginRepository>
          <id>mavenCentral</id>
          <url>https://repo1.maven.org/maven2/</url>
       </pluginRepository>
    </pluginRepositories>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-stdlib</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.jetbrains.kotlin</groupId>
                <artifactId>kotlin-maven-plugin</artifactId>
                <version>${kotlin.version}</version>
                ...
            </plugin>
        </plugins>
    </build>
</project>
```

If you run into any problems

- Report an issue to our issue tracker, YouTrack.
- Find help in the #eap channel in Kotlin Slack (get an invite).
- Roll back to the latest stable version: change it in your build script file.

Learning materials overview

You can use the following materials and resources for learning Kotlin:

- Basic syntax – get a quick overview of the Kotlin syntax.
- Kotlin tour – learn the fundamentals of the Kotlin programming language.
- Idioms – learn how to write idiomatic Kotlin code for popular cases.
    - Java to Kotlin migration guide: Strings – learn how to perform typical tasks with strings in Java and Kotlin.

- Java to Kotlin migration guide: Collections — learn how to perform typical tasks with collections in Java and Kotlin.
- Java to Kotlin migration guide: Nullability — learn how to handle nullability in Java and Kotlin.

- Kotlin Koans – complete exercises to learn the Kotlin syntax. Each exercise is created as a failing unit test and your job is to make it pass. Recommended for developers with Java experience.
- Kotlin Core track by JetBrains Academy – learn all the Kotlin essentials while creating working applications step by step.
- Kotlin books – find books we've reviewed and recommend for learning Kotlin.
- Kotlin tips – watch short videos where the Kotlin team shows you how to use Kotlin in a more efficient and idiomatic way, so you can have more fun when writing code.
- Advent of Code puzzles – learn idiomatic Kotlin and test your language skills by completing short and fun tasks.
- Kotlin hands-on tutorials – complete long-form tutorials to fully grasp a technology. These tutorials guide you through a self-contained project related to a specific topic.
- Kotlin for Java Developers – learn the similarities and differences between Java and Kotlin in this course on Coursera.
- Kotlin documentation in PDF format – read our documentation offline.

Kotlin hands-on

A series of hands-on tutorials where you can create applications with Kotlin using a variety of different technologies and targeting multiple platforms. The exercises are divided into a series of steps, walking you through each section.

Creating a web app with Spring Boot and Kotlin

Build a simple application with database functionality using Spring Boot and Kotlin, and learn about the benefits of using Kotlin for backend development from a syntax perspective.

Start

Building web applications with React and Kotlin/JS

Create a React Application using Kotlin/JS, and see how you can leverage Kotlin's type system, library ecosystem, and interoperability features.

Start

Building web applications with Spring Boot and Kotlin

Build a simple blog application by combining the power of Spring Boot and Kotlin.

Start

Creating HTTP APIs with Ktor

Create a backend API for your application that responds to HTTP requests.

Start

Creating a WebSocket chat with Ktor

Create a simple Chat application using Ktor including both a JVM server and a JVM client.

Start

Creating an interactive website with Ktor

Learn how to serve files, use templating engines such as Freemarker and the kotlinx.html DSL, and work with form input from Ktor.

Start

Introduction to Kotlin coroutines and channels

Learn about coroutines in Kotlin and how you can communicate between them using channels.

Start

Introduction to Kotlin/Native

Create a simple HTTP client that can run natively on multiple platforms using Kotlin/Native and libcurl.

Start

Kotlin Multiplatform: networking and data storage

Learn how to create a mobile application for Android and iOS using Kotlin Multiplatform with Ktor and SQLDelight.

Start

Targeting iOS and Android with Kotlin Multiplatform

Learn how to create a mobile application that can target both iOS and Android using Kotlin Multiplatform.

Start

Kotlin tips

Kotlin Tips is a series of short videos where members of the Kotlin team show how to use Kotlin in a more efficient and idiomatic way to have more fun when writing code.

Subscribe to our YouTube channel to not miss new Kotlin Tips videos.

null + null in Kotlin

What happens when you add `null + null` in Kotlin, and what does it return? Sebastian Aigner addresses this mystery in our latest quick tip. Along the way, he also shows why there's no reason to be scared of nullables:

Watch video online.

Deduplicating collection items

Got a Kotlin collection that contains duplicates? Need a collection with only unique items? Let Sebastian Aigner show you how to remove duplicates from your lists, or turn them into sets in this Kotlin tip:

Watch video online.

The suspend and inline mystery

How come functions like `repeat()`, `map()` and `filter()` accept suspending functions in their lambdas, even though their signatures aren't coroutines-aware? In this episode of Kotlin Tips, Sebastian Aigner solves the riddle: it has something to do with the inline modifier:

Watch video online.

Unshadowing declarations with their fully qualified name

Shadowing means having two declarations in a scope have the same name. So, how do you pick? In this episode of Kotlin Tips, Sebastian Aigner shows you a simple Kotlin trick to call exactly the function that you need, using the power of fully qualified names:

Watch video online.

Return and throw with the Elvis operator

Elvis has entered the building once more! Sebastian Aigner explains why the operator is named after the famous singer, and how you can use `?:` in Kotlin to return or throw. The magic behind the scenes? The Nothing type.

Watch video online.

Destructuring declarations

With destructuring declarations in Kotlin, you can create multiple variables from a single object, all at once. In this video, Sebastian Aigner shows you a selection of things that can be destructured – pairs, lists, maps, and more. And what about your own objects? Kotlin's component functions provide an answer for those as well:

Watch video online.

Operator functions with nullable values

In Kotlin, you can override operators like addition and subtraction for your classes and supply your own logic. But what if you want to allow null values, both on their left and right sides? In this video, Sebastian Aigner answers this question:

Watch video online.

Timing code

Watch Sebastian Aigner give a quick overview of the `measureTimedValue()` function, and learn how you can time your code:

Watch video online.

Improving loops

In this video, Sebastian Aigner will demonstrate how to improve loops to make your code more readable, understandable, and concise:

Watch video online.

Strings

In this episode, Kate Petrova shows three tips to help you work with Strings in Kotlin:

Watch video online.

Doing more with the Elvis operator

In this video, Sebastian Aigner will show how to add more logic to the Elvis operator, such as logging to the right part of the operator:

Watch video online.

Kotlin collections

In this episode, Kate Petrova shows three tips to help you work with Kotlin Collections:

Watch video online.

What's next?

- See the complete list of Kotlin Tips in our YouTube playlist
- Learn how to write idiomatic Kotlin code for popular cases

Kotlin Koans

Kotlin Koans are a series of exercises designed primarily for Java developers to help you become familiar with the Kotlin syntax. Each exercise is created as a failing unit test, and your job is to make it pass.

> You can check out the task content in our GitHub repository.

You can complete the Kotlin Koans tasks in one of the following ways:

- You can play with Koans online.

- You can perform the tasks right inside IntelliJ IDEA or Android Studio by installing the JetBrains Academy plugin and choosing the Kotlin Koans course.

Whatever way you choose to solve koans, you can see the solution for each task:

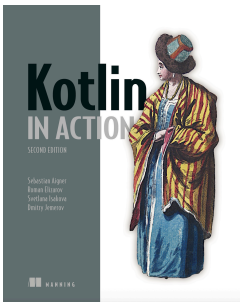
- In the online version, click **Show answer**.
- For the JetBrains Academy plugin, try to complete the task first and then choose **Peek solution** if your answer is incorrect.

We recommend you check the solution after implementing the task to compare your answer with the proposed one. Make sure you don't cheat!

Kotlin books

More and more authors write books for learning Kotlin in different languages. We are very thankful to all of them and appreciate all their efforts in helping us increase a number of professional Kotlin developers.

Here are just a few books we've reviewed and recommend you for learning Kotlin. You can find more books on our community website.



Kotlin in Action

Kotlin in Action teaches you to use the Kotlin language for production-quality applications. Written for developers familiar with Java or other object-oriented languages, this example-rich book goes further than most language books. The second edition is enriched with an extensive section on the Kotlin coroutines library.

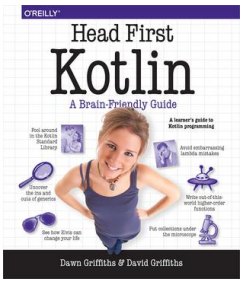
The book is written by Sebastian Aigner, Roman Elizarov, Svetlana Isakova, and Dmitry Jemerov, who are all either current or former members of the Kotlin team.



Atomic Kotlin

Atomic Kotlin is for both beginning and experienced programmers!

From Bruce Eckel, author of the multi-award-winning Thinking in C++ and Thinking in Java, and Svetlana Isakova, Kotlin Developer Advocate at JetBrains, comes a book that breaks the language concepts into small, easy-to-digest "atoms", along with a free course consisting of exercises supported by hints and solutions directly inside IntelliJ IDEA!

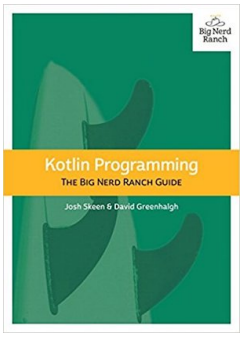


Head First Kotlin

Head First Kotlin is a complete introduction to coding in Kotlin. This hands-on book helps you learn the Kotlin language with a unique method that goes beyond syntax and how-to manuals and teaches you how to think like a great Kotlin developer.

You'll learn everything from language fundamentals to collections, generics, lambdas, and higher-order functions. Along the way, you'll get to play with both object-oriented and functional programming.

If you want to really understand Kotlin, this is the book for you.



Kotlin Programming: The Big Nerd Ranch Guide

Kotlin Programming: The Big Nerd Ranch Guide

In this book you will learn to work effectively with the Kotlin language through carefully considered examples designed to teach you Kotlin's elegant style and features.

Starting from first principles, you will work your way to advanced usage of Kotlin, empowering you to create programs that are more reliable with less code.



Programming Kotlin

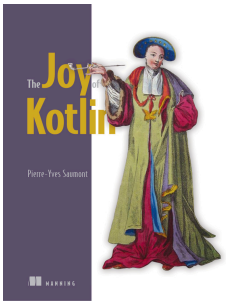
Programming Kotlin is written by Venkat Subramaniam.

Programmers don't just use Kotlin, they love it. Even Google has adopted it as a first-class language for Android development.

With Kotlin, you can intermix imperative, functional, and object-oriented styles of programming and benefit from the approach that's most suitable for the problem at hand.

Learn to use the many features of this highly concise, fluent, elegant, and expressive statically typed language with easy-to-understand examples.

Learn to write maintainable, high-performing JVM and Android applications, create DSLs, program asynchronously, and much more.



The Joy of Kotlin

The Joy of Kotlin teaches you the right way to code in Kotlin.

In this insight-rich book, you'll master the Kotlin language while exploring coding techniques that will make you a better developer no matter what language you use. Kotlin natively supports a functional style of programming, so seasoned author Pierre-Yves Saumont begins by reviewing the FP principles of immutability, referential transparency, and the separation between functions and effects.

Then, you'll move deeper into using Kotlin in the real world, as you learn to handle errors and data properly, encapsulate shared state mutations, and work with laziness.

This book will change the way you code — and give you back some of the joy you had when you first started.

Advent of Code puzzles in idiomatic Kotlin

Advent of Code is an annual December event, where holiday-themed puzzles are published every day from December 1 to December 25. With the permission of Eric Wastl, creator of Advent of Code, we'll show how to solve these puzzles using the idiomatic Kotlin style:

- Advent of Code 2025
- Advent of Code 2024
- Advent of Code 2023

- Advent of Code 2022
- Advent of Code 2021
- Advent of Code 2020

Get ready for Advent of Code

We'll take you through the basic tips on how to get up and running with solving Advent of Code challenges with Kotlin:

- Use this GitHub template to create projects
- Check out the welcome video by Kotlin Developer Advocate, Sebastian Aigner:

Watch video online.

Advent of Code 2022

Day 1: Calorie counting

Learn about the Kotlin Advent of Code template and convenience functions for working with strings and collections in Kotlin, such as `maxOf()` and `sumOf()`. See how extension functions can help you structure your solution in a nice manner.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 1 | Kotlin

Day 2: Rock paper scissors

Understand operations on the `Char` type in Kotlin, see how the `Pair` type and the `to` constructor work well with pattern matching. Understand how to order your own objects using the `compareTo()` function.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 2 | Kotlin

Day 3: Rucksack reorganization

Learn how the kotlinx.benchmark library helps you understand the performance characteristics of your code. See how set operations like `intersect` can help you select overlapping data, and see performance comparisons between different implementations of the same solution.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 3 | Kotlin

Day 4: Camp cleanup

See how `infix` and `operator` functions can make your code more expressive, and how extension functions for the `String` and `IntRange` types make it easy to parse input.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

 Advent of Code 2022 Day 4 | Kotlin

Day 5: Supply stacks

Learn about constructing more complex objects with factory functions, how to use regular expressions, and the double-ended `ArrayDeque` type.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

 Advent of Code 2022 Day 5 | Kotlin

Day 6: Tuning trouble

See more in-depth performance investigations with the kotlinx.benchmark library, comparing the characteristics of 16 different variations of the same solution.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

 Advent of Code 2022 Day 6 | Kotlin

Day 7: No space left on device

Learn how to model tree structures, and see a demo of generating Kotlin code programmatically.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

 Advent of Code 2022 Day 7 | Kotlin

Day 8: Treetop tree house

See the `sequence` builder in action, and how far a first draft of a program and an idiomatic Kotlin solution can differ (with special guest Roman Elizarov!).

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

 Advent of Code 2022 Day 8 | Kotlin

Day 9: Rope bridge

See the `run` function, labeled returns, and convenient standard library functions like `coerceIn`, or `zipWithNext`. See how you can construct lists of given sizes using the `List` and `MutableList` constructors, and get a peek at a Kotlin-based visualization of the problem statement.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 9 | Kotlin

Day 10: Cathode-ray tube

Learn how ranges and the `in` operator make checking ranges natural, how function parameters can be turned into receivers, and a brief exploration of the `tailrec` modifier.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 10 | Kotlin

Day 11: Monkey in the middle

See how you can move from mutable, imperative code to a more functional approach that makes use of immutable and read-only data structures. Learn about context receivers and how our guest built his own visualization library just for Advent of Code.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 11 | Kotlin

Day 12: Hill Climbing algorithm

Use queues, `ArrayDeque`, function references, and the `tailrec` modifier to solve path finding problems with Kotlin.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 12 | Kotlin

Advent of Code 2021

Read our blog post about Advent of Code 2021

Day 1: Sonar sweep

Apply windowed and count functions to work with pairs and triplets of integers.

- Read the puzzle description on Advent of Code

- Check out the solution from Anton Arhipov on the Kotlin Blog or watch the video:

 Advent of Code 2021 in Kotlin, Day 1: Sonar Sweep

Day 2: Dive!

Learn about destructuring declarations and the `when` expression.

- Read the puzzle description on Advent of Code
- Check out the solution from Pasha Finkelshteyn on GitHub or watch the video:

 Advent of Code 2021 in Kotlin, Day 2: Dive!

Day 3: Binary diagnostic

Explore different ways to work with binary numbers.

- Read the puzzle description on Advent of Code
- Check out the solution from Sebastian Aigner on Kotlin Blog or watch the video:

 Advent of Code 2021 in Kotlin, Day 3: Binary Diagnostic

Day 4: Giant squid

Learn how to parse the input and introduce some domain classes for more convenient processing.

- Read the puzzle description on Advent of Code
- Check out the solution from Anton Arhipov on the GitHub or watch the video:

 Advent of Code 2021 in Kotlin, Day 4: Giant Squid

Advent of Code 2020

 You can find all the solutions for the Advent of Code 2020 puzzles in our GitHub repository.

Day 1: Report repair

Explore input handling, iterating over a list, different ways of building a map, and using the `let` function to simplify your code.

- Read the puzzle description on Advent of Code
- Check out the solution from Svetlana Isakova on the Kotlin Blog or watch the video:

 Learn Kotlin With the Kotlin Team: Advent of Code 2020 #1

Day 2: Password philosophy

Explore string utility functions, regular expressions, operations on collections, and how the `let` function can be helpful to transform your expressions.

- Read the puzzle description on Advent of Code

- Check out the solution from Svetlana Isakova on the Kotlin Blog or watch the video:

 Learn Kotlin with The Kotlin Team: Advent of Code 2020 #2

Day 3: Toboggan trajectory

Compare imperative and more functional code styles, work with pairs and the `reduce()` function, edit code in the column selection mode, and fix integer overflows.

- Read the puzzle description on Advent of Code
- Check out the solution from Mikhail Dvorkin on GitHub or watch the video:

 Learn Kotlin with the Kotlin Team: Advent of Code 2020 #3

Day 4: Passport processing

Apply the `when` expression and explore different ways of how to validate the input: utility functions, working with ranges, checking set membership, and matching a particular regular expression.

- Read the puzzle description on Advent of Code
- Check out the solution from Sebastian Aigner on the Kotlin Blog or watch the video:

 Learn Kotlin with the Kotlin Team: Advent of Code 2020 #4

Day 5: Binary boarding

Use the Kotlin standard library functions (`replace()` , `toInt()` , `find()`) to work with the binary representation of numbers, explore powerful local functions, and learn how to use the `max()` function in Kotlin 1.5.

- Read the puzzle description on Advent of Code
- Check out the solution from Svetlana Isakova on the Kotlin Blog or watch the video:

 Learn Kotlin with the Kotlin Team: Advent of Code 2020 #5

Day 6: Custom customs

Learn how to group and count characters in strings and collections using the standard library functions: `map()` , `reduce()` , `sumOf()` , `intersect()` , and `union()` .

- Read the puzzle description on Advent of Code
- Check out the solution from Anton Arhipov on the Kotlin Blog or watch the video:

 Learn Kotlin with the Kotlin Team: Advent of Code 2020 #6

Day 7: Handy haversacks

Learn how to use regular expressions, use Java's `compute()` method for HashMaps from Kotlin for dynamic calculations of the value in the map, use the `forEachLine()` function to read files, and compare two types of search algorithms: depth-first and breadth-first.

- Read the puzzle description on Advent of Code
- Check out the solution from Pasha Finkelshteyn on the Kotlin Blog or watch the video:

Learn Kotlin with the Kotlin Team: Advent of Code 2020 #7

Day 8: Handheld halting

Apply sealed classes and lambdas to represent instructions, apply Kotlin sets to discover loops in the program execution, use sequences and the `sequence { }` builder function to construct a lazy collection, and try the experimental `measureTimedValue()` function to check performance metrics.

- Read the puzzle description on Advent of Code
- Check out the solution from Sebastian Aigner on the Kotlin Blog or watch the video:

Learn Kotlin with the Kotlin Team: Advent of Code 2020 #8

Day 9: Encoding error

Explore different ways to manipulate lists in Kotlin using the `any()`, `firstOrNull()`, `firstNotNullOfOrNull()`, `windowed()`, `takeIf()`, and `scan()` functions, which exemplify an idiomatic Kotlin style.

- Read the puzzle description on Advent of Code
- Check out the solution from Svetlana Isakova on the Kotlin Blog or watch the video:

Learn Kotlin with the Kotlin Team: Advent of Code 2020 #9

What's next?

- Complete more tasks with Kotlin Koans
- Create working applications with the free Kotlin Core track by JetBrains Academy

Learning Kotlin with JetBrains Academy plugin

With the JetBrains Academy plugin, available both in Android Studio and IntelliJ IDEA, you can learn Kotlin through code practicing tasks.

Take a look at the Learner Start Guide, which will get you started with the Kotlin Koans course inside IntelliJ IDEA. Solve interactive coding challenges and get instant feedback right inside the IDE.

If you want to use the JetBrains Academy plugin for teaching, read Teaching Kotlin with JetBrains Academy plugin.

Teaching Kotlin with JetBrains Academy plugin

With the JetBrains Academy plugin, available both in Android Studio and IntelliJ IDEA, you can teach Kotlin through code practicing tasks.

Take a look at the Educator Start Guide to learn how to create a simple Kotlin course that includes a set of programming tasks and integrated tests.

If you want to use the JetBrains Academy plugin to learn Kotlin, read Learning Kotlin with JetBrains Academy plugin.

FAQ

What is Kotlin?

Kotlin is an open-source statically typed programming language that targets the JVM, Android, JavaScript, Wasm, and Native. It's developed by JetBrains. The project started in 2010 and was open source from very early on. The first official 1.0 release was in February 2016.

What is the current version of Kotlin?

The currently released version is 2.4.20, published on September 7, 2026.
You can find more information on GitHub.

Is Kotlin free?

Yes. Kotlin is free, has been free, and will remain free. It is developed under the Apache 2.0 license, and the source code is available on GitHub.

Is Kotlin an object-oriented language or a functional one?

Kotlin has both object-oriented and functional constructs. You can use it in both OO and FP styles, or mix elements of the two. With first-class support for features such as higher-order functions, function types, and lambdas, Kotlin is a great choice if you're doing or exploring functional programming.

What advantages does Kotlin give me over the Java programming language?

Kotlin is more concise. Rough estimates indicate approximately a 40% cut in the number of lines of code. It's also more type-safe — for example, support for non-nullable types makes applications less prone to NPE's. Other features including smart casting, higher-order functions, extension functions, and lambdas with receivers provide the ability to write expressive code as well as facilitating creation of DSL.

Is Kotlin compatible with the Java programming language?

Yes. Kotlin is 100% interoperable with the Java programming language, and major emphasis has been placed on making sure that your existing codebase can interact properly with Kotlin. You can easily call Kotlin code from Java and Java code from Kotlin. This makes adoption much easier and lower-risk. There's also an automated Java-to-Kotlin converter built into the IDE that simplifies migration of existing code.

What can I use Kotlin for?

Kotlin can be used for any kind of development, be it backend, client-side web, Android, or multiplatform library. With Kotlin/Native currently in the works, support for other platforms such as embedded systems, macOS, and iOS. People are using Kotlin for mobile and server-side applications, client-side with JavaScript or JavaFX, and data science, just to name a few possibilities.

Can I use Kotlin for Android development?

Yes. Kotlin is supported as a first-class language on Android. There are hundreds of applications already using Kotlin for Android, such as Basecamp, Pinterest, and more. For more information, check out the resource on Android development.

Can I use Kotlin for backend development?

Yes. Kotlin is 100% compatible with the JVM, and as such you can use any existing frameworks such as Spring Boot, vert.x, or JSF. In addition, there are specific frameworks written in Kotlin, such as Ktor. For more information, check out the Backend development overview.

Can I use Kotlin for web development?

Yes. For backend web development, Kotlin works well with frameworks such as Ktor and Spring, enabling you to build server-side applications efficiently. Additionally, you can use Kotlin/Wasm for client-side web development. Learn how to get started with Kotlin/Wasm.

Can I use Kotlin for desktop development?

Yes. You can use any Java UI framework such as JavaFx, Swing, or other. In addition, there are Kotlin-specific frameworks such as TornadoFX.

Can I use Kotlin for native development?

Yes. Kotlin/Native is available as a part of Kotlin. It compiles Kotlin to native code that can run without a VM. You can try it on popular desktop and mobile platforms and even some IoT devices. For more information, check out the Kotlin/Native documentation.

What IDEs support Kotlin?

Kotlin has full out-of-the-box support in IntelliJ IDEA and Android Studio with an official Kotlin plugin developed by JetBrains. You can also install the official Kotlin by JetBrains extension for Visual Studio Code, powered by Kotlin Language Server, which is in Alpha.

Other IDEs and code editors only have Kotlin community-supported plugins.

You can also try Kotlin Playground for writing, running, and sharing Kotlin code in your browser. In addition, a command line compiler is available, which provides straightforward support for compiling and running applications.

What build tools support Kotlin?

On the JVM side, the main build tools include Gradle and Maven. There are also some build tools available that target client-side JavaScript.

What does Kotlin compile down to?

When targeting the JVM, Kotlin produces Java-compatible bytecode.

When targeting JavaScript, Kotlin transpiles to ES5.1 and generates code which is compatible with module systems including AMD and CommonJS.

When targeting native, Kotlin will produce platform-specific code (via LLVM).

Which versions of JVM does Kotlin target?

Kotlin lets you choose the version of JVM for execution. By default, the Kotlin/JVM compiler produces Java 8 compatible bytecode. If you want to make use of optimizations available in newer versions of Java, you can explicitly specify the target Java version from 9 to 26. Note that in this case, the resulting bytecode might not run on lower versions. Starting with Kotlin 1.5, the compiler does not support producing bytecode compatible with Java versions below 8.

Is Kotlin hard?

Kotlin is inspired by existing languages such as Java, C#, JavaScript, Scala, and Groovy. We've tried to ensure that Kotlin is easy to learn, so that people can easily jump on board, reading and writing Kotlin in a matter of days. Learning idiomatic Kotlin and using some more of its advanced features can take a little longer, but overall it is not a complicated language.
For more information, check out our learning materials.

What companies are using Kotlin?

There are too many companies using Kotlin to list, but some more visible companies that have publicly declared usage of Kotlin, be this via blog posts, GitHub repositories or talks include Square, Pinterest, and Basecamp.

Who develops Kotlin?

Kotlin is developed by a team of engineers at JetBrains (current team size is 100+). The lead language designer is Michail Zarečenskij. In addition to the core team, there are also over 250 external contributors on GitHub.

Where can I learn more about Kotlin?

The best place to start is our website. To start with Kotlin, you can install one of the official IDEs or try it online.

Are there any books on Kotlin?

There are a number of books available for Kotlin. Some of them we have reviewed and can recommend to start with. They are listed on the Books page. For more books, see the community-maintained list at kotlin.link.

Are any online courses available for Kotlin?

You can learn all the Kotlin essentials while creating working applications with the Kotlin Core track by JetBrains Academy.

A few other courses you can take:

- Pluralsight Course: Getting Started with Kotlin by Kevin Jones
- O'Reilly Course: Introduction to Kotlin Programming by Hadi Hariri

You can also check out the other tutorials and content on our YouTube channel.

Does Kotlin have a community?

Yes! Kotlin has a very vibrant community. Kotlin developers hang out on the Kotlin forums, StackOverflow, and more actively on the Kotlin Slack (with close to 30000 members as of April 2020).

Are there Kotlin events?

Yes! There are many User Groups and Meetups now focused exclusively around Kotlin. You can find a list on the website. In addition, there are community-organized Kotlin Nights events around the world.

Is there a Kotlin conference?

Yes! KotlinConf is an annual conference hosted by JetBrains, which brings together developers, enthusiasts, and experts from around the world to share their knowledge and experience with Kotlin.

In addition to technical talks and workshops, KotlinConf also offers networking opportunities, community interactions, and social events where attendees can connect with fellow Kotliners and exchange ideas. It serves as a platform for fostering collaboration and community building within the Kotlin ecosystem.

Kotlin is also being covered in different conferences worldwide. You can find a list of upcoming talks on the website.

Is Kotlin on social media?

Yes. Subscribe to the Kotlin YouTube channel and follow Kotlin on Twitter or on Bluesky.

Any other online Kotlin resources?

The website has a bunch of online resources, including Kotlin Digests by community members, a newsletter, a podcast, and more.

Where can I get an HD Kotlin logo?

Logos can be downloaded here. When using the logos, please follow simple rules in the `guidelines.pdf` inside the archive and Kotlin brand usage guidelines.

For more information, check out the page about Kotlin brand assets.

Google Summer of Code with Kotlin

The Kotlin Foundation provides mentorship for Google Summer of Code (GSoC), a global online program that introduces new contributors to open-source development.

Learn more about GSoC on the official Google Summer of Code website.

GSoC 2026: project ideas

The Kotlin Foundation has published a list of project ideas for Google Summer of Code 2026. If you're interested in contributing, check out the available projects and guidelines: Google Summer of Code with Kotlin 2026.

Past GSoC projects with Kotlin

You can explore past Google Summer of Code projects and ideas with Kotlin:

- Google Summer of Code with Kotlin 2025
- Google Summer of Code with Kotlin 2024
- Google Summer of Code with Kotlin 2023

Google Summer of Code with Kotlin 2026

This article contains the list of project ideas for Google Summer of Code with Kotlin 2026, and contributor guidelines

Kotlin resources:

- Kotlin GitHub repository
- Kotlin Slack and the #gsoc Slack channel

If you got any questions, contact us via gsoc@kotlinfoundation.org

Kotlin contributor guidelines for Google Summer of Code (GSoC)

Getting started

1. Check out the GSoC FAQ and the program announcement.
2. Familiarize yourself with the Kotlin language:
   - The official Kotlin website is a great place to start.
   - Read the official documentation to get a better understanding of the language.
   - Take a look at the Kotlin courses on JetBrains Academy or the Android team's Training options.
   - Follow the Kotlin X or Kotlin Bluesky accounts to stay up to date on the latest news and developments.
   - Check out the Kotlin YouTube channel for tutorials, tips, and the latest updates.
3. Get to know the Kotlin open source community:
   - Explore the general Kotlin contribution guidelines.
   - Join the Kotlin Slack channel to connect with other developers and get help with any questions you may have.

- Join the #gsoc channel to ask questions and get support from the GSoC team.

How to apply

1. Check out the project ideas and select the one you would like to work on.
2. If you are not familiar with Kotlin, read the introductory info on the Kotlin website.
3. Refer to the GSoC contributor guidelines.
4. Apply via the GSoC website.
    - We suggest that you write a working code sample relevant to the proposed project. You can also show us any code sample that you are particularly proud of.
    - Describe why you are interested in Kotlin and your experience with it.
    - If you participate in open source projects, please reference your contribution history.
    - If you have a GitHub, Twitter account, blog, or portfolio of technical or scientific publications, please reference them as well.
    - Disclose any conflicts with the GSoC timeline due to other commitments, such as exams and vacations.

Thank you! We look forward to reading your applications!

Project ideas

Kotlin Compiler Fuzzer (Kai) [Hard, 350 hrs]

In recent years, fuzzing has become a widely used technique for finding complex bugs in software. The Kotlin compiler is no exception: previous fuzzing efforts resulted in more than 200 deduplicated bugs across different compiler subsystems.

However, the existing fuzzer implementation is now obsolete and cannot be reasonably evolved further. The goal of this project is to build a new Kotlin compiler fuzzer, **Kai**, from scratch, based on previous experience and modern tools and techniques.

The main goal of this internship is to establish a solid foundation for future development of the fuzzer. The focus areas include:

- Designing a fuzzer architecture that supports pluggability
- Selecting tools for generating, mutating, and processing Kotlin code
- Defining reliable ways to detect compiler failures
- Designing proper workflows for collecting, classifying, and handling discovered issues

As a deliverable, we aim to create a prototype Kotlin compiler fuzzer that is modular and easy to evolve, unlike a monolithic implementation. Finding real compiler bugs would be a great bonus, but it is not the primary goal of this internship.

If you have preliminary questions about the project, contact the mentor at: marat.akhin [at] jetbrains.com

Expected outcomes

A prototype Kotlin compiler fuzzer with a pluggable architecture that supports future evolution.

Skills required (must-have)

- Proficiency in Kotlin or another JVM-based language
- Technical English sufficient for reading relevant papers and documentation
- Basic understanding of compilers

Skills required (nice-to-have)

- Familiarity with fuzzing or other forms of program analysis
- Experience with Kotlin compiler plugins, IDE plugins, or other pluggable systems
- Experience with greenfield developer tooling projects

What you will learn

- Hands-on experience with compiler fuzzing
- How internal developer tooling is designed and built
- How to design and implement pluggable systems

Possible mentor

Marat Akhin, JetBrains

This project includes a test task. Please email education@kotlinlang.org to receive the details.

Swift-to-Kotlin interop (PoC) [Hard, 350 hrs]

Modern software projects rarely live in a single language ecosystem. On Apple platforms, Swift is the primary language, while Kotlin is widely used for shared and cross-platform business logic. However, there is currently no straightforward way to import Swift APIs directly into Kotlin.

In this project, you will build on an existing open-source Swift–Java bridge and add Kotlin/Native as a target runtime. This includes designing how Swift APIs are exposed to Kotlin/Native, how calls cross the Swift/Kotlin boundary, and how object lifetimes are managed across runtimes.

The goal is to create a proof of concept for Swift-to-Kotlin/Native interop, document design decisions and trade-offs, and evaluate limitations and future directions.

Expected outcomes

A proof of concept for Swift-to-Kotlin/Native interop, with documented design decisions, trade-offs, limitations, and future directions.

Skills required (must-have)

- Currently pursuing or recently completed a degree in Computer Science or a related field
- Familiarity with Swift
- Interest in programming languages and interoperability

Skills required (nice-to-have)

- Familiarity with Kotlin

Possible mentor

Artem Olkov, JetBrains

This project includes a test task. Please email education@kotlinlang.org to receive the details.

Tail call support in the Kotlin/Wasm backend [Medium, 90 hrs]

This project focuses on integrating the tail call proposal into the Kotlin/Wasm backend. The intern will design and implement tail call support and evaluate its impact through benchmarking.

Expected outcomes

Design and implementation of tail call support for Kotlin/Wasm, with benchmarks and evaluation.

Skills required (preferred)

General familiarity with interpreters and compilers, and interest in optimization and benchmarking.

Possible mentor

Charlie Zhang, JetBrains

Kotlin Education landscape report [Medium, 175 hrs]

Kotlin is taught and used in educational settings worldwide. This project aims to create a structured, up-to-date overview of where and how Kotlin is taught.

Expected outcomes

- A “Kotlin in Education” report highlighting trends and gaps
- Reusable datasets for internal and public use
- Input for future strategy

Skills required (preferred)

- Research and data analysis skills
- Interest in programming education and developer ecosystems

Possible mentor

Ksenia Shneyveys, JetBrains

Google Summer of Code with Kotlin 2025

This article contains the list of project ideas for Google Summer of Code with Kotlin 2025, and contributor guidelines

Kotlin resources:

- Kotlin GitHub repository
- Kotlin Slack and the #gsoc Slack channel

If you got any questions, contact us via gsoc@kotlinfoundation.org

Kotlin contributor guidelines for Google Summer of Code (GSoC)

Getting started

1. Check out the GSoC FAQ and the program announcement.

2. Familiarize yourself with the Kotlin language:
    - The official Kotlin website is a great place to start.
    - Read the official documentation to get a better understanding of the language.
    - Take a look at the Kotlin courses on JetBrains Academy or the Android team's Training options.
    - Follow the Kotlin X or Kotlin Bluesky accounts to stay up to date on the latest news and developments.
    - Check out the Kotlin YouTube channel for tutorials, tips, and the latest updates.
3. Get to know the Kotlin open source community:
    - Explore the general Kotlin contribution guidelines.
    - Join the Kotlin Slack channel to connect with other developers and get help with any questions you may have.
    - Join the #gsoc channel to ask questions and get support from the GSoC team.

How to apply

1. Check out the project ideas and select the one you would like to work on.
2. If you are not familiar with Kotlin, read the introductory info on the Kotlin website.
3. Refer to the GSoC contributor guidelines.
4. Apply via the GSoC website.
    - We suggest that you write a working code sample relevant to the proposed project. You can also show us any code sample that you are particularly proud of.
    - Describe why you are interested in Kotlin and your experience with it.
    - If you participate in open source projects, please reference your contribution history.
    - If you have a GitHub, Twitter account, blog, or portfolio of technical or scientific publications, please reference them as well.
    - Disclose any conflicts with the GSoC timeline due to other commitments, such as exams and vacations.

Thank you! We look forward to reading your applications!

Project ideas

Build Server Protocol: add Kotlin support [Hard, 350 hrs]

The Kotlin team wants to expand official Kotlin support not only for Gradle and Maven build systems, but any other build system as well and support them natively in JetBrains IDEs with minimal effort. On the other hand, we also want to provide basic Kotlin support in non-JetBrains IDEs – one part of such support is being able to get Kotlin-specific information from any build system supporting Kotlin.

The solution to these requirements could be a Build Server Protocol (BSP) which provides an abstraction layer between the build system and IDE.

The goal of this project would be implementing a prototype which uses the BSP protocol to get all the required information for IntelliJ IDEA from a user project so that it will be possible to work with Kotlin code in the project. To limit the scope of this prototype, the user project will use Gradle to build itself automatically.

Preferred skills

- Knowledge of Kotlin
- Understanding of how to write Gradle plugins
- *Bonus*: understanding of how to write plugins for IntelliJ IDEA

Possible mentors

Yahor Berdnikau, Bálint Hegyi, and Reinhold Degenfellner

Tasks for applicants

- **Task #1**. Why are you interested in this project?
- **Task #2**. Practice assignment: Create a Gradle plugin which exposes a specific task. This task should, in the presence of the Kotlin Gradle Plugin, retrieve all the Kotlin sources' structures and output them. Including tests would be a bonus.

Support Android and iOS targets in Kotlin Multiplatform for Gemini using Vertex AI in Firebase [Medium, 175 hrs]

This project aims to create an open-source Kotlin Multiplatform (KMP) library that supports Gemini using Vertex AI in Firebase on at least Android and iOS. It will showcase best practices in creating KMP libraries for existing services, with a focus on appropriate production implementation (for example, proper API key management, user-managed API keys support, and client throttling).

Expected outcomes

- A new Kotlin Multiplatform library with support for an existing Google service
- Sample code and documentation

Preferred skills

- Kotlin
- Kotlin Multiplatform
- Mobile development (Android and iOS)

Possible mentors

Matt Dyor, and the Google team

Add Kotlin Multiplatform support in Bazel [Hard, 350 hrs]

Bazel's support for Kotlin is evolving, but proper Kotlin Multiplatform (KMP) integration remains a challenge. This project aims to improve Bazel's KMP support by addressing dependency resolution issues, enhancing `rules_kotlin` and `rules_jvm_external` compatibility, and enabling cross-platform builds.

Key improvements will focus on handling platform-specific dependencies (expect/actual mechanisms), improving Gradle metadata support, and ensuring a smoother developer experience for KMP in Bazel.

Expected outcomes

- Enhanced dependency resolution for Kotlin Multiplatform in Bazel
- Improved integration with `rules_kotlin` and `rules_jvm_external`
- A working KMP build setup in Bazel for seamless multiplatform development

Preferred skills

- Kotlin Multiplatform and Gradle
- Bazel build system
- Dependency resolution strategies

Possible mentors

Shauvik Roy Choudhary, and the Uber team

Kotlin Language Server (LSP) [Hard, 350 hrs]

The Language Server Protocol (LSP) is a widely adopted standard that enables code intelligence features such as autocompletion, go-to definition, and refactoring across different editors and IDEs. While there is currently no official Kotlin LSP server, there is significant demand for one in the community. A publicly maintained, community-driven implementation can support broad use cases, including code migration, AI-powered code assistance, and seamless integration into various development environments.

This project aims to develop a Kotlin LSP implementation, ensuring compatibility with key LSP features and broadening Kotlin's accessibility across development environments.

Expected outcomes

Develop a Kotlin LSP implementation

Preferred skills

- Kotlin
- Language Server Protocol (LSP)
- Plugin or extension development for IDEs

Possible mentors

Shauvik Roy Choudhary, and the Uber team

Maven Central publishing plugin for Gradle with new APIs [Medium, 175 hrs]

Maven Central is one of the most popular Maven repositories for publishing JVM-focused libraries and projects. It is actively used by Apache Maven or Gradle-based open-source projects, and based on Sonatype Nexus v2, pending migration to a newer version. There is ongoing migration of open source projects to a new Maven Central Instance, which has a very different API implementation and needs special support in the build tool plugins. Developing a Gradle plugin that is compatible with the new Maven Central publication APIs would help the library authors building with Gradle to have a smooth experience with the new process.

Currently, there are multiple implementations of Maven Central publishing plugins in Gradle, for example, the Maven Publish Plugin or the New Maven Central Publishing, which already tries to adopt the new APIs. During the application or the community bonding phase, a potential contributor would need to review the implementations and either suggest an existing plugin to update or decide to build a new plugin or fork. The deliverables would include either a new version of an existing plugin for Maven Central publishing or a new plugin for Gradle. We anticipate the implementation to be in Kotlin or Java and to have proper test coverage and documentation. Additional deliverables may include Kotlin DSL extensions to simplify the use of the plugins and Declarative Gradle extensions.

Expected outcomes

- Updated Maven Central publishing plugin or a new plugin

Preferred skills

- Kotlin
- Gradle
- Maven Repositories

Possible mentors

Oleg Nenashev, and the Gradle team

Improving Configuration Cache and lock contention in key Gradle plugins [Easy to Hard, 90 hrs to 350 hrs]

Gradle is working on Isolated Projects – a new feature that greatly extends the configuration cache to further improve performance, particularly the performance of Android Studio and IntelliJ IDEA sync. From the developer experience standpoint, it is one of the most expected features in Gradle.

One of the problems for Isolated projects is the lock contention in the Gradle core, with plugins sometimes getting in the way of parallel execution. We would like to reduce the lock contention, especially in the key Gradle Build Tool plugins for Java, Kotlin, Android, and Kotlin Multiplatform ecosystems. Contributors are welcome to choose the deliverables based on their interests and the desired project size.

Potential deliverables include but are not limited to:

- Embed the Configuration Cache Report tool into the Gradle Profiler (or "implement a GitHub Action for it")
- Profile Gradle and a few popular Gradle plugins in various projects, with automation of the test suite on GHA
- Determine potential areas and plugins where lock contention can be reduced, with or without Configuration Cache
- While around, contribute to other areas of Configuration Cache compatibility in the target plugins
- Implement some of the discovered improvements

Expected outcomes

Implementing extensibility features in the Kotlin DSL for Gradle and improving support for common project integrations

Preferred skills

- Kotlin
- Gradle
- Java
- Performance analysis
- Profiling

Possible mentors

Oleg Nenashev, Laura Kassovic

Gradle convention plugin for developing Jenkins plugins [Easy to Hard, 90 hrs to 350 hrs]

There are more than 50 Jenkins plugins that are implemented with Gradle. There is a Gradle JPI plugin, but it is not fully compliant with Jenkins hosting requirements and needs an update. In this project idea, the aim would be to recover the Gradle developer flow for Jenkins, reach feature parity with the Apache Maven flow (Parent POM, Plugin Compatibility Tester, Jenkins Bill of Materials, and others), and to improve the developer experience for those who develop Jenkins plugins with Gradle.

Contributors are welcome to choose the deliverables based on their interest and the desired project size.

Potential deliverables include but are not limited to:

- Refreshing the Gradle JPI plugin and making it compliant with hosting best practices
- Migrating the Gradle JPI plugin codebase from Groovy to Kotlin
- Implementing a new convention plugin for Jenkins Plugins that would cover the main features of Jenkins plugin Parent POM, with Kotlin and Kotlin DSL. This would include not just building the plugin, but also testing and static analysis according to Jenkins' best practices
- Adopting the refreshed plugin and/or the convention plugin in the most popular Gradle plugin, including the Gradle plugin itself
- Integrating the Gradle plugin into the Plugin Compatibility Tester and Bill of Materials
- Documenting the updated Gradle development flow for Jenkins plugins

Expected outcomes

Updated Gradle JPI plugin and/or new convention plugin for Jenkins, published on Jenkins Update Center and the Gradle Plugin Portal

Preferred skills

- Kotlin DSL
- Kotlin
- Gradle
- Jenkins
- Java

Possible mentors

Oleg Nenashev, Stefan Wolf

Kotlin DSL and Declarative Gradle documentation samples test framework [Easy to Medium, 90 hrs to 175 hrs]

Many projects, including Gradle, have a lot of Kotlin DSL samples and code snippets (see the Gradle Docs for examples). Testing them against multiple versions poses certain challenges because the snippets often represent incomplete code for the sake of brevity. We would like to build a test framework that simplifies the verification of those samples within a unit test framework (Kotest or JUnit 5) on GitHub Actions or TeamCity. Later we would be interested in doing the same for Declarative Gradle samples.

Expected outcomes

Implementing extensibility features in the Kotlin DSL for Gradle and improving support for common project integrations

Preferred skills

- Kotlin
- Gradle
- Java
- Static analysis

Possible mentors

Oleg Nenashev, Laura Kassovic

IntelliJ Platform Gradle Plugin – Gradle Reporting and Parallel Verifications [Medium, 175 hrs]

The IntelliJ Platform Gradle Plugin, a plugin for the Gradle build system, simplifies configuring your environment for building, testing, verifying, and publishing plugins for IntelliJ-based IDEs. The plugin manages the build, test, and verification steps while keeping up with the constant changes introduced in the IntelliJ Platform. The IntelliJ Platform Gradle Plugin is used by JetBrains, third-party developers, and external companies to integrate their workflows with JetBrains tools.

Expected outcomes

- Introduce Gradle Reporting to provide detailed, configurable verification task reports.
- Utilize Gradle Worker API to enable parallel execution of the `verifyPlugin` task against multiple IntelliJ Platform versions, reducing the task execution time.
- Explore additional Gradle enhancements to further improve plugin development workflows.

Preferred skills

- Kotlin
- Gradle
- IntelliJ Platform

Possible mentors

Jakub Chrzanowski, JetBrains

Add More Kotlin OpenRewrite Recipes [Medium, 175 hrs]

OpenRewrite is a powerful framework for automating code migrations and refactorings in a structured manner. While OpenRewrite has strong support for Java, the Kotlin ecosystem would benefit from a more comprehensive set of OpenRewrite recipes to help developers seamlessly migrate their codebases.

This project aims to expand the Kotlin OpenRewrite recipe collection by adding more automated transformations, such as migrating Java-based AutoValue classes to idiomatic Kotlin data classes, modernizing Kotlin code to follow best practices, and enabling more seamless migrations across Kotlin versions. These recipes will help Kotlin developers maintain clean, up-to-date, and idiomatic codebases with minimal manual effort.

Expected outcomes

- Development of new OpenRewrite recipes for Kotlin code migrations

Preferred skills

- Kotlin

- OpenRewrite framework
- Java-to-Kotlin migration strategies

Possible mentors

Shauvik Roy Choudhary, and the Uber team

Add BOM Support to Bazel `rules_jvm_external` [Hard, 350 hrs]

Bazel's `rules_jvm_external` provides a structured way to declare external Java dependencies, but it currently lacks proper support for Bill of Materials (BOM) files. BOM files are widely used in Maven and Gradle to manage dependencies in a consistent manner without requiring developers to specify individual versions. This project aims to enhance `rules_jvm_external` by adding BOM support, allowing developers to use BOM-based dependency resolution within Bazel. The project may involve contributing to an existing open-source effort or implementing BOM support directly in `rules_jvm_external`, ensuring compatibility with widely used dependency management approaches.

Expected outcomes

- Implementation of BOM support in Bazel `rules_jvm_external`
- Improved dependency resolution and usability for Bazel users
- Documentation and examples for using BOM support in Bazel

Preferred skills

- Starlark (Bazel's scripting language)
- Bazel build system
- Dependency resolution strategies

Possible mentors

Shauvik Roy Choudhary, and the Uber team

Clean and actionable reporting for Gradle code quality plugins for Kotlin [Easy to Medium, 90 hrs to 175 hrs]

Gradle recently introduced a new Problems API that allows Gradle and third-party plugins to propagate issues and warnings in a unified way. This API provides clean and actionable error reporting and more insights into the console output, dedicated HTML reports, and connected observability tools. IDEs such as IntelliJ IDEA or Android Studio can also access the details via Gradle's API integration tool, and can show warnings right in the code editor. Several core features and plugins have already adopted the Problems API: Java compilation, dependency resolution errors, deprecation warnings, etc. We want the code quality plugins for Kotlin to adopt this API, too; it would significantly improve the developer experience for 100,000+ Kotlin developers using Gradle.

In this project, we invite contributors to choose a number of Kotlin code quality plugins, such as Ktlint, Detekt, Diktat, ArchUnit, or Checkstyle for Kotlin, and integrate them with Problems API. You can also work on integrating a similar analysis for Gradle builds defined with KotlinDSL.

Expected outcomes

- Implement Problems API integration in the mentioned plugins

Preferred skills

- Kotlin

- Gradle

Possible mentors

Oleg Nenashev, Balint Hegyi, Reinhold Degenfellner

Google Summer of Code with Kotlin 2024

This article contains the list of project ideas for Google Summer of Code with Kotlin 2024, and contributor guidelines

Kotlin resources:

- Kotlin GitHub repository
- Kotlin Slack and the #gsoc Slack channel

If you got any questions, contact us via gsoc@kotlinfoundation.org

Kotlin contributor guidelines for Google Summer of Code (GSoC)

Getting started

1. Check out the GSoC FAQ and the program announcement.
2. Familiarize yourself with the Kotlin language:
   - The official Kotlin website is a great place to start.
   - Read the official documentation to get a better understanding of the language.
   - Take a look at the Kotlin courses on JetBrains Academy or the Android team's Training options.
   - Follow the Kotlin X or Kotlin Bluesky accounts to stay up to date on the latest news and developments.
   - Check out the Kotlin YouTube channel for tutorials, tips, and the latest updates.
3. Get to know the Kotlin open source community:
   - Explore the general Kotlin contribution guidelines.
   - Join the Kotlin Slack channel to connect with other developers and get help with any questions you may have.
   - Join the #gsoc channel to ask questions and get support from the GSoC team.

How to apply

1. Check out the project ideas and select the one you would like to work on.
2. If you are not familiar with Kotlin, read the introductory info on the Kotlin website.
3. Refer to the GSoC contributor guidelines.
4. Apply via the GSoC website.
   - We suggest that you write a working code sample relevant to the proposed project. You can also show us any code sample that you are particularly proud of.

- Describe why you are interested in Kotlin and your experience with it.
- If you participate in open source projects, please reference your contribution history.
- If you have a GitHub, Twitter account, blog, or portfolio of technical or scientific publications, please reference them as well.
- Disclose any conflicts with the GSoC timeline due to other commitments, such as exams and vacations.

Thank you! We look forward to reading your applications!

Project ideas

Incremental compilation for the Kotlin-to-WebAssembly compiler [Hard, 350 hrs]

Incremental compilation is a technique that helps increase compilation speed by recompiling only changed files instead of your whole program (also known as performing a clean build). The Kotlin-to-Wasm compiler currently supports only clean builds, but during this project, we will enhance it to support incremental compilation, too.

Expected outcomes

Implementation of the incremental compilation feature in the Kotlin-to-WebAssembly compiler, contributing to faster development workflows.

Skills required (preferred)

Kotlin

Possible mentor

Artem Kobzar, JetBrains

Compose Multiplatform with Node.js native bindings for Skia [Hard, 350 hrs]

Compose Multiplatform is a declarative framework for sharing UIs built with Kotlin across multiple platforms. The current approach to rendering desktop applications uses the JVM as a platform, but what if we use Compose Multiplatform for Web and try to run an application outside the browser with Skia's native bindings? Will this improve desktop applications performance and memory consumption? Or will it do the opposite? We'll find out in this project!

Expected outcomes

Integration of Skia bindings with Compose Multiplatform and evaluation of performance impact on desktop applications.

Skills required (preferred)

Kotlin, Node.js, C++, or Rust

Possible mentor

Artem Kobzar, JetBrains

Compose Multiplatform component gallery generator [Medium, 350 hrs]

Compose Multiplatform is a declarative framework for sharing UIs built with Kotlin across multiple platforms. At the beginning of the React era of web development, Storybook was created, and Storybook's proposed approach of describing component states and generating the whole UI library gallery is still one of the essential approaches

to documentation in web development. Can we do the same with Compose Multiplatform, using it to generate a gallery of web UI elements, as well as galleries for mobile and desktop? Let's give it a try in this project.

Expected outcomes

Creation of a tool to generate UI component galleries for web, mobile, and desktop platforms using Compose Multiplatform.

Skills required (preferred)

Kotlin, Jetpack Compose, UI/UX Design

Possible mentor

Artem Kobzar, JetBrains

Kotlin DSL improvements for declarative Gradle [Medium, 175 hrs]

Last November, the Gradle team announced the new Declarative Gradle project, introducing a higher level Kotlin DSL in the project. We invite GSoC contributors to join us and work on improving the developer experience of the new DSL, in particular by implementing extensibility in Gradle plugins to support the most common Kotlin and Java project integrations: static analysis, test frameworks like Kotest and others.

Expected outcomes

Implementing extensibility features in the Kotlin DSL for Gradle and improving support for common project integrations.

Skills required (preferred)

Kotlin, Gradle, Java, Static Analysis

Possible mentor

Oleg Nenashev, Gradle

Gradle guidelines

Kotlin DSL documentation samples test framework [Easy or Medium, 90 hrs or 175 hrs]

Many projects, including Gradle, have a lot of Kotlin DSL samples and code snippets (see the Gradle Docs for examples). Testing them against multiple versions poses certain challenges because the snippets often represent incomplete code for the sake of brevity. We would like to build a test framework that simplifies the verification of those samples within a unit test framework (Kotest or JUnit 5) on GitHub Actions and Teamcity.

Expected outcomes

Implementation of a basic test framework for Kotlin DSL samples, integrated with GitHub Actions for continuous testing.

Skills required (preferred)

Kotlin, Testing Frameworks, CI/CD

Possible mentor

Oleg Nenashev, Gradle

Gradle guidelines

Gradle build server – support for Android projects [Medium or Hard, 175 hrs or 350 hrs]

Kotlin and Gradle are the default choices for building Android projects. In November 2023, the Microsoft team announced the Gradle Build Server project, which is a Gradle-specific implementation of the Build Server Protocol (BSP). It would be great to introduce full support for Android builds there. For smaller-scope projects, it is possible to implement auto-discovery and cancellation for Gradle tasks in the Gradle Build Server.

Expected outcomes

Implementation of Android project support in the Gradle Build Server, including auto-discovery and task cancellation.

Skills required (preferred)

Kotlin, Gradle, Android Development, Visual Studio Code

Possible mentor

Oleg Nenashev, Gradle

Gradle guidelines

Implement memory usage profiling for Kotlin/Native benchmarks [Medium, 175 hrs]

The kotlinx-benchmark library, an open-source toolkit, facilitates the benchmarking of Kotlin code across various platforms. It currently features GC profiling for the JVM, detailing each benchmark method's allocation rate. This project aims to extend similar profiling capabilities to Kotlin/Native, advancing toward uniform benchmarking capabilities across platforms.

The contributor will collaborate closely with the Kotlin/Native team to create an API for accessing allocation data from the Kotlin/Native memory manager. The objective is to generate reports that align with the JVM format, ensuring the consistency of data presentation across platforms. Furthermore, the project involves identifying and rectifying any discrepancies in reporting formats for other library features, thereby standardizing the benchmarking output for comprehensive cross-platform analysis.

Expected outcomes

Implementation of memory usage profiling in kotlinx-benchmark for Kotlin/Native and standardized benchmarking output.

Skills required (preferred)

Kotlin, Kotlin/Native, Benchmarking, Memory Profiling

Possible mentor

Abduqodiri Qurbonzoda, JetBrains
Alexander Shabalin, JetBrains

Support Android target in kotlinx-benchmark [Medium, 175 hrs]

The kotlinx-benchmark library is an open-source tool designed for benchmarking Kotlin code across multiple platforms, including the JVM, JS, WasmJs, and Native. Despite its broad compatibility, the library currently does not support benchmarking on Android. This project aims to bridge that gap. The plan is to utilize an existing Android library, such as androidx.benchmark, behind the scenes to integrate this functionality. A key aspect of the project will be ensuring that all features currently available for other platforms are also supported on Android, maintaining the library's multiplatform utility.

Expected outcomes

Integration of benchmarking support for Android platforms in kotlinx-benchmark, ensuring feature parity with other platforms.

Skills required (preferred)

Kotlin, Android Development, Benchmarking

Possible mentor

Abduqodiri Qurbonzoda, JetBrains
Rahul Ravikumar, Google

Enabling click-to-run for kotlinx-benchmark benchmarks in IntelliJ IDEA [Medium, 175 hrs]

kotlinx-benchmark is an open-source library for benchmarking multiplatform code written in Kotlin. It includes a Gradle plugin that, when applied, provides tasks for running benchmarks. However, executing these tasks requires navigating to the IDE's Gradle panel or using the terminal. Additionally, running a specific benchmark necessitates further steps, adding to the complication. To mitigate this inconvenience and streamline the process, this project aims to enable users to run an individual benchmark or an entire suite directly from the IntelliJ IDEA interface, mirroring the convenience offered for unit tests. Achieving this goal may necessitate collaboration with the IntelliJ IDEA team and/or contributions directly to the IntelliJ project.

Expected outcomes

Integration of click-to-run functionality for kotlinx-benchmark benchmarks in IntelliJ IDEA, improving user experience.

Skills required (preferred)

Kotlin, IntelliJ IDEA Plugin Development, Benchmarking

Possible mentor

Abduqodiri Qurbonzoda, JetBrains

Google Summer of Code with Kotlin 2023

This article contains the list of project ideas for Google Summer of Code with Kotlin 2023.

> Google Summer of Code 2023 has already ended. If you want to participate in GSoC 2024, check out this list of project ideas.

Project ideas

Kotlin Multiplatform protobufs [Hard, 350 hrs]

Description

Add support for Kotlin/Common protos to protoc with Kotlin/Native (iOS) runtime and Objective-C interop.

Motivation

While protobufs have many platform implementations, there isn't a way to use them in Kotlin Multiplatform projects.

Expected outcomes

Design and build Kotlin Multiplatform Protobuf support, culminating in contributions to:

- GitHub – protocolbuffers/protobuf: Protocol Buffers – Google's data interchange format
- GitHub – google/protobuf-gradle-plugin
- Kotlin Multiplatform Gradle Plugin

Skills required (preferred)

- Kotlin
- Objective-C
- C++

Kotlin Compiler error messages [Hard, 350 hrs]

Description

Add improved compiler error messages to the K2 Kotlin compiler: more actionable and detailed information (like Rust has).

Motivation

Rust compiler error messages are often regarded as being by far the most helpful of any compiler. The Kotlin K2 compiler provides a great foundation for better compiler errors in Kotlin but this potential is somewhat untapped.

Expected outcomes

Using StackOverflow and other data sources, uncover common compiler errors which would have significant value to users. Make contributions back to the compiler to improve those error messages.

Skills required (preferred)

- Kotlin
- Compiler architecture

Kotlin Multiplatform libraries [Easy or Medium, 175 or 350 hrs]

Description

Create and deliver (to Maven Central) Kotlin Multiplatform libraries that are commonly needed. For instance, compression, crypto.

Motivation

Kotlin Multiplatform is still fairly new and could use some additional libraries which are either platform independent (Kotlin/Common) and/or have platform implementations (`expect` / `actual`).

Expected outcomes

Design and deliver at least one Kotlin Multiplatform library with a greater priority on JVM/Android and Kotlin/Native (iOS) than other targets (Kotlin/JS).

Skills required (preferred)

- Kotlin
- Objective-C

Groovy to Kotlin Gradle DSL Converter [Medium, 350 hrs]

Description

The project aims to create a Groovy-to-Kotlin converter with a primary focus on Gradle scripts. We will start from basic use cases, such as when a user wants to paste Groovy-style dependency declarations to a Kotlin script and the IDE automatically converts them. Later, we will start supporting more complex code constructs and conversions of complete files.

Motivation

The Kotlin Gradle DSL is gaining popularity, so much so that it will soon become the default choice for building projects with Gradle. However, many documents and resources about Gradle still refer to Groovy, and pasting Groovy samples into `build.gradle.kts` requires manual editing. Furthermore, many new features around Gradle will be in Kotlin first, and consequently users will migrate from the Groovy DSL to the Kotlin DSL. The automatic code conversion of a build setup will therefore greatly ease this migration, saving a lot of time.

Expected outcomes

A plugin for IntelliJ IDEA that can convert Groovy code to Kotlin with the main focus on the Gradle DSL.

Skills required (preferred)

- Basic knowledge of Gradle
- Basic knowledge of parsers and how compilers work in general
- Basic knowledge of Kotlin

Eclipse Gradle KTS editing [Medium, 350 hrs]



Description

Improve the experience of editing Gradle Kotlin Scripts (KTS) in Eclipse.

Motivation

IntelliJ IDEA and Android Studio have great support for editing KTS Gradle build scripts, but the Eclipse support is lacking. Ctrl-Click to definition, Code completion, Code error highlighting could all be improved.

Expected outcomes

Make contributions to the Gradle Eclipse plugin that improve the developer experience for editing KTS.

Skills required (preferred)

- Kotlin
- Gradle
- Eclipse platform and plugins

Improve support for parameter forwarding in the Kotlin Plugin for IntelliJ IDEA [Medium, 350 hrs]

Description and motivation

The Kotlin plugin provides Kotlin language support in IntelliJ IDEA and Android Studio. In the scope of this project, you will improve parameter forwarding support for the plugin.

To prefer composition over inheritance is a widely known principle. IntelliJ IDEA provides great support for writing code that uses inheritance (completion and quick-fixes the IDE suggests), but the support for code that uses composition instead of inheritance has yet to be implemented.

The main problem of working with code that heavily uses composition is parameter forwarding. In particular:

- The IDE doesn't suggest completing parameter declarations that can be forwarded as arguments to other functions that currently use default parameter values.
- The IDE doesn't rename the chain of forwarded parameters.
- The IDE doesn't provide any quick-fixes that fill in all the required arguments with parameters that can be forwarded.

One notable example where such support would be greatly appreciated is Jetpack Compose. Android's modern tool kit for building UI, Jetpack Compose heavily uses function composition and parameter forwarding. It quickly becomes tedious to work with `@Composable` functions because they have a lot of parameters. For example, `androidx.compose.material.TextField` has 19 parameters.

Expected outcomes

- Improved parameter and argument completion suggestions in IntelliJ IDEA.
- Implemented IDE quick-fixes that suggest filling in all the required arguments with parameters with the same names and types.
- The **Rename** refactoring renames the chain of forwarded parameters.
- All other IDE improvements around parameter forwarding and functions that have a lot of parameters.

Skills required (preferred)

- Knowledge of Kotlin and Java
- Ability to navigate in a large codebase

Enhance the kotlinx-benchmark library API and user experience [Easy, 175 hrs]

Read the blog post about this project

Description

`kotlinx-benchmark` is an open-source library for benchmarking multiplatform code written in Kotlin. It has a barebones skeleton but lacks quality-of-life features, such as fine-grained benchmark configuration (like time units, modes), feature parity between JVM and Kotlin/Native benchmarking, a command-line API, and modern Gradle support. Its documentation, integration tests, and examples are also lagging.

Motivation

The library has already been implemented, but it is sometimes difficult to use correctly and confuses some users. Improving the library's user experience would greatly help the Kotlin community.

Expected outcomes

- The library has clear documentation with usage examples.

- The library API is simple and easy to use.
- Options for benchmarking Kotlin/JVM code are also available for benchmarking code on other platforms.

Skills required (preferred)

- Kotlin
- Gradle internals

Parallel stacks for Kotlin Coroutines in the debugger [Hard, 350 hrs]

Read the blog post about this project

Description

Implement Parallel Stacks view for Kotlin coroutines to improve the coroutine debugging experience.

Motivation

Currently, support for coroutines debugging is very limited in IntelliJ IDEA. The Kotlin debugger has the **Coroutines Panel** that allows a user to view all of the coroutines and their states, but it's not very helpful when debugging an application with lots of coroutines in it. The JetBrains Rider has the Parallel Stacks feature that allows a user to inspect threads and their stack traces in a graph view, which could be a great way of inspecting coroutines.

Expected outcomes

Using the Kotlin coroutines debugger API, develop the IntelliJ IDEA plugin which would add the parallel stacks view for coroutines to the debugger. Find ways to improve the graph representation of coroutines.

Skills required (preferred)

- Kotlin
- Kotlin coroutines
- IntelliJ IDEA plugin development

Contribution

Kotlin is an open-source project under the Apache 2.0 License. The source code, tooling, documentation, and even this website are maintained on GitHub. Kotlin is developed by JetBrains, but we are always on the lookout for more people to help us.

Participate in Early Access Preview

You can help us improve Kotlin by participating in Kotlin Early Access Preview (EAP) and providing us with your valuable feedback.

For every release, Kotlin ships a few preview builds where you can try out the latest features before they go to production. You can report any bugs you find to our issue tracker YouTrack and we will try to fix them before a final release. This way, you can get bug fixes earlier than the standard Kotlin release cycle.

Contribute to the compiler and standard library

If you want to contribute to the Kotlin compiler and standard library, go to JetBrains/Kotlin GitHub, check out the latest Kotlin version, and follow the instructions on how to contribute.

You can help us by completing open tasks. Please keep an open line of communication with us because we may have questions and comments on your changes. Otherwise, we won't be able to incorporate your contributions.

Contribute to the Kotlin IDE plugin

Kotlin IDE plugin is a part of the IntelliJ IDEA repository.

To contribute to the Kotlin IDE plugin, clone the IntelliJ IDEA repository and follow the instructions on how to contribute.

Contribute to other Kotlin libraries and tools

Besides the standard library that provides core capabilities, Kotlin has a number of additional (kotlinx) libraries that extend its functionality. Each kotlinx library is developed in a separate repository, has its own versioning and release cycle.

If you want to contribute to a kotlinx library (such as kotlinx.coroutines or kotlinx.serialization) and tools, go to Kotlin GitHub, choose the repository you are interested in and clone it.

Follow the contribution process described for each library and tool, such as kotlinx.serialization, ktor, and others.

If you have a library that could be useful to other Kotlin developers, let us know via feedback@kotlinlang.org.

Contribute to the documentation

If you've found an issue in the Kotlin documentation, feel free to check out the documentation source code on GitHub and send us a pull request. Follow these guidelines on style and formatting.

Please keep an open line of communication with us because we may have questions and comments on your changes. Otherwise, we won't be able to incorporate your contributions.

Translate documentation to other languages

You are welcome to translate the Kotlin documentation into your own language and publish the translation on your website. However, we won't be able to host your translation in the main repository and publish it on kotlinlang.org.

This site is the official documentation for the language, and we ensure that all the information here is correct and up to date. Unfortunately, we won't be able to review documentation in other languages.

Hold events and presentations

If you've given or just plan to give presentations or hold events on Kotlin, please fill out the form. We'll feature them on the event list.

KUG guidelines

A Kotlin User Group, or KUG, is a community that is dedicated to Kotlin and that offers you a place to share your Kotlin programming experience with like-minded people.

To become a KUG, your community should have some specific features shared by every KUG. It should:

- Provide Kotlin-related content with regular meetups as the main form of activity.
- Host regular events (at least once every 3 months) with open registration and without any restriction for attendance.
- Be driven and organized by the community, and it should not use events to earn money or gain any other business benefits from members and attendees.
- Follow and ensure a code of conduct to provide a welcoming environment for attendees of any background and experience (check out our recommended Code of Conduct).

There are no limits regarding the format for KUG meetups. They can take place in whatever fashion works best for the community, whether that includes presentations, hands-on labs, lectures, hackathons, or informal beer-driven get-togethers.



For Kotlin User Group brand assets, see Kotlin brand assets documentation.

How to run a KUG?

- To promote group cohesion and prevent miscommunication, we recommend keeping to a limit of one KUG per city. Check out the list of KUGs to see if there is already a KUG in your area.
- Use the official KUG logo and branding. Check out the branding guidelines.
- Keep your user group active. Run meetups regularly, at least once every 3 months.
- Announce your KUG meetups at least 2 weeks in advance. The announcement should contain a list of talks and the names of the speakers, as well as the location, timing, and any other crucial info about the event.
- KUG events should be free or, if you need to cover organizing expenses, limit prices to a maximum of 10 USD.
- Your group should have a code of conduct available for all members.

If your community has all the necessary features and follows these guidelines, you are ready to Apply to be a new KUG.

Have a question? Contact us

Support for KUGs from JetBrains

Active KUGs that host at least 1 meetup every 3 months can apply for the community support program, which includes:

- Official KUG branding.
- A special entry on the Kotlin website.
- Free licenses for JetBrains products to raffle off at meetups.
- Priority support for Kotlin events and campaigns.

- Help with recruiting Kotlin speakers for your events.

Support from JetBrains for other tech communities

If you organize any other tech communities, you can apply for support as well. By doing so, you may receive:

- Free licenses for JetBrains products to raffle off at meetups.
- Information about Kotlin official events and campaigns.
- Kotlin stickers.
- Help with recruiting Kotlin speakers for your events.

Disclaimer: Support and/or any assistance provided by JetBrains is subject to its discretion and compliance with regulatory requirements and applicable laws by the recipient.

Kotlin Night guidelines

Kotlin Night is a meetup that includes 3-4 talks on Kotlin or related technologies.

For Kotlin Night brand assets, see Kotlin brand assets documentation.

Event guidelines

- Please use the branding materials we've provided. Having all events and materials in the same style will help keep the Kotlin Night experience consistent.
- Kotlin Night should be a free event. A minimal fee can be charged to cover expenses, but it should remain a non-profit event.
- The event should be announced publicly and open for all people to attend without any kind of discrimination.
- If you publish the contents of the talks online after the event, they must be free and accessible to everyone, without any sign-up or registration procedures.
- Recordings are optional but recommended, and they should also be made available. If you decide to record the talks, we suggest having a plan to ensure the quality is good.
- The talks should primarily be about Kotlin and should not focus on marketing or sales.
- The event can serve food and drinks optionally.

Event requirements

JetBrains is excited to support your Kotlin Night event. Because we want all events to provide the same high-quality experience, we need organizers to ensure that some basic requirements are met for the event to receive JetBrains support. As an organizer, you are responsible for the following aspects of the event:

1. The location and everything required to host the event, including booking a comfortable venue. Please make sure that:

- All the participants are aware of the exact date, place, and starting time of the event, along with the event schedule and program.
- There is enough space as well as food and beverages, if you provide them, for everyone.
- You have a plan with your speakers. This includes a schedule, topics, abstracts for the talks, and any necessary equipment for the presentations.

2. Content and speakers
    - Feel free to invite presenters from your local community, from neighboring countries, or even from all over the globe. You don't have to have any JetBrains representatives or speakers at your event. However, we are always happy to hear about more Kotlin Nights, so feel free to notify us.
3. Announcements and promotion
    - Announce your event at least three weeks before the date of a meetup.
    - Include the schedule, topics, abstracts, and speaker bios in the announcement.
    - Spread the word on social media.
4. Providing event material to JetBrains after the event
    - We would be glad to announce your event at kotlinlang.org, and we would appreciate it if you provided slides and video materials for a follow-up posting.

JetBrains support

JetBrains provides support with:

- Access to Kotlin Night Branding, which includes the name and logos
- Merchandise, such as stickers and t-shirts for speakers and small souvenirs for attendees
- A listing for the event on the Kotlin Talks page
- Help to reach out to speakers to take part in the event, if necessary
- Help to find a location if possible (via contacts, etc.), as well as help to identify possible partnerships with local businesses

Code of conduct and guidelines for Kotlin Slack

Kotlin Slack aims to be an inclusive space committed to providing a friendly, safe and welcoming environment for all, regardless of gender, sexual orientation, ability, ethnicity, socioeconomic status, and religion (or lack thereof).

For this to be the case, it is vital that we all follow a basic set of guidelines and most importantly, adhere to the code of conduct. As such, please make sure you read this Code of Conduct and Basic Usage Guidelines in its entirety. This isn't your regular License Agreement that you should scroll through and agree blindly. It's here for a reason and also contains practical information.

- How to behave
- How not to behave
- How to report issues
- Moderators

- Basic usage guidelines
- Moderators
- Copyright

How to behave

- Participate in an authentic and active way. In doing so, you contribute to the health and longevity of this community.
- Exercise consideration, respect and empathy in your speech and actions. Remember, we have all been through different stages of learning when adopting technologies.
- Refrain from demeaning, discriminatory, or harassing behavior and speech.
- Disagreements on things are fine, argumentative behavior or trolling are not.

How not to behave

- Do *not* perform threats of violence or use violent language directed against another person.
- Do *not* make jokes of sexist, racist, homophobic, transphobic, ableist or otherwise discriminatory nature, or use language of this nature.
- Do *not* post or display sexually explicit or violent material.
- Do *not* post or threaten to post other people's personally identifying information ("doxing").
- Do *not* make personal insults, particularly those related to gender, sexual orientation, race, religion, or disability.
- Do *not* engage in sexual attention. This includes sexualized comments or jokes and sexual advances.
- Do *not* advocate for, or encourage, any of the above behavior.

How to report issues

If someone is acting inappropriately or violating this Code of Conduct in any shape or form, and they are not receptive to your feedback, or you prefer not to confront them, please get in touch with one of the Administrators. The main Administrators are Ilya Ryzhenkov (@orangy) and Hadi Hariri (@hhariri).

Basic usage guidelines

There are over 50 000 users on many different timezones using the Kotlin Slack, and the number is growing rapidly. The influx of messages per day and the numerous channels can lead to a lot of noise and little value in the long run. If we all follow a set of guidelines, it can help make things more useful and bearable for everyone.

- Please remember this is a Slack team for Kotlin. If you have generic questions that are about a technology (which may or may not be used with Kotlin), maybe it's better to use another forum, such as StackOverflow.
- This Slack is not an official support channel. It is a place where the community hangs out along with some members of the JetBrains and Kotlin team (suffix [JetBrains]).
- This Slack is not the best place for reporting bugs. For this, please use YouTrack.
- Find the right channel to ask your question. There are channels for pretty much every topic.

- New channels should be focused around a well-known technology or area of interest. Create them sparingly. And before requesting for a new channel, make sure it doesn't exist in any other shape or form. Ask an administrator to create a new channel on the #meta channel. If you do end up creating a channel, make sure you set the topic.
- Please refrain from cross-posting the same message on multiple channels. It is considered spamming.
- When you ask a question, please be patient. Don't repeat it. Or at least not immediately.
- Please do not ping or mention someone directly to get your questions answered, especially project owners, whether the project is Kotlin or other, unless they specifically indicate that you can.
- Don't split messages into multiple ones. Ask it all in a single message.
- Use code blocks. Don't paste code as plain text. Slack supports Kotlin markup. If the code you are pasting is longer than a few lines, use "Code or text snippet" available from `+` menu next to message input. Only first few lines will be displayed to all users, and people interested in the code can expand your code to look into details.
- While it's polite to say "Hi" or "I have a question" before asking something, it doesn't scale and leads to noise. You can say hello in the same message you ask the question if you like.
- Using threads is not required, but do take into account that it does allow people to more easily follow conversations, especially those jumping in late.
- Use reactions to show gratitude as opposed to a message. It reduces the noise and gets the message across.
- Don't use reactions to tell people they're in the wrong place or asking the wrong question. If they're on the wrong channel, point them to the right one. If their question is badly worded, help them correct it. Have empathy.
- Take into account cultural differences. As they say, what the British say isn't what the British mean. Remember this both as the receiver and producer of messages.
- Don't create integrations with Slack. Given this is running on the free tier, the number is limited, and thus it won't be approved.
- @channel, @here and other forms of notifications are disabled, even if you own a channel (unfortunately, Slack doesn't allow fine-grained permissions).
- Use the #meta channel if you're unsure about how to do something.

Moderators

A moderator is a member of the Kotlin Slack community who volunteers to keep the kotlinlang Slack a safe and welcoming place for other members. Moderators are approved by kotlinlang Slack administrators. Their responsibilities and the guidelines they follow are described below.

Responsibilities

- Monitor Slack channels to ensure compliance with the Code of Conduct.
- Help community members follow the rules.
- If a channel has additional rules, ensure these are visible and up to date.
- Monitor the #reports and #meta channels.
- Participate in the private #moderators channel.

Guidelines

- Moderators communicate in a friendly manner and do their best to help other community members.
- Any changes suggested by a moderator should be discussed by all moderators and administrators before coming into effect.

Process

- Each moderator oversees at least one channel, or more if possible.
- When moderators notice inappropriate behavior, they explain the rules and guidelines to the members and help figure out how to correct the unwanted behavior. If the unwanted behavior persists, moderators report the case to administrators for further actions. In the case of obvious and/or particularly blatant violations, offending members can be reported immediately.
- To set up or update a channel's rules, the moderator initiates a change proposal and discusses it with the appropriate community members.
- Communication regarding moderation takes place in the private #moderators channel.
- If community members notice that a moderator is ignoring their responsibilities (or engaging in any unwanted behavior), they can alert administrators.

Moderator status acquisition and revocation process

- Any member is welcome to apply to be a moderator once they have been in the Slack community for longer than 6 months.
- Moderator status is provided based upon the decision of the administrators.
- Moderator status can be revoked if (this list is not exhaustive and can be updated on a case-by-case basis):
  - A moderator ignores their responsibilities, which leads to issues in a channel.
  - A moderator violates these guidelines.

The list of moderators

At the moment we have 17 confirmed moderators, and they will be responsible for observing a total of 64 channels. For all other channels, if any issues arise, members can ping any of the moderators or admins directly, or they can post a message in #meta.

- Alexander Nozik (@altavir) – #mathematics, #science, #datascience, #education
- Anderson Lameck (@andylamax) – #coroutines, #kotlin-native, #webassembly, #serialization, #random, #javascript, #react
- Andrey Mischenko (@gildor) – #coroutines, #android, #gradle, #multiplatform, #getting-started, #kotlin-asia, #singapore
- Eric Ampire (@Eric Ampire) – #events
- Holger Steinhauer (@Holger Steinhauer) – #fosdem, #berlin, #vkug, #german-lang, #server
- Louis CAD (@louiscad) – #coroutines, #android
- Magda Miu (@Magda Miu) – #android, #100daysofkotlin
- Maryam Alhuthayfi (@Maryam Alhuthayfi) – #compose, #books, #ksp, #eap, #dagger, and #kontributors.

- Nicola Corti (@gammax) – #feed, #detekt, #appintro, #chucker, #fosdem, #london, #berlin, #hamburg, #sweden, #ktlint, #kug-leads, #kotlinconf, #koin, #koin-dev, #spek, #vkug
- Paulien van Alst (@Paulien van Alst) – #detekt, #koin, #mockk
- Qian Jin (@qian) – #kotlin-native, #android-studio, #french
- Raul Raja (@raulraja) - #arrow, #arrow-contributors, #arrow-meta
- Sam Sam (@sam) – #kotest, #kotest-contributors
- Simon Vergauwen (@simon.vergauwen) – #arrow, #arrow-contributors, #arrow-meta
- Youssef Shoaib (@Youssef Shoaib) – #getting-started, #feed, #language-evolution, #language-proposals
- Zach Klippenstein (@Zach Klippenstein) – #coroutines, #compose, #compose-desktop, #compose-web, #squarelibraries, #library-development

Copyright

This Code of Conduct is distributed under a Creative Commons Attribution-ShareAlike license. Portions of a text derived from the Citizen Code of Conduct.

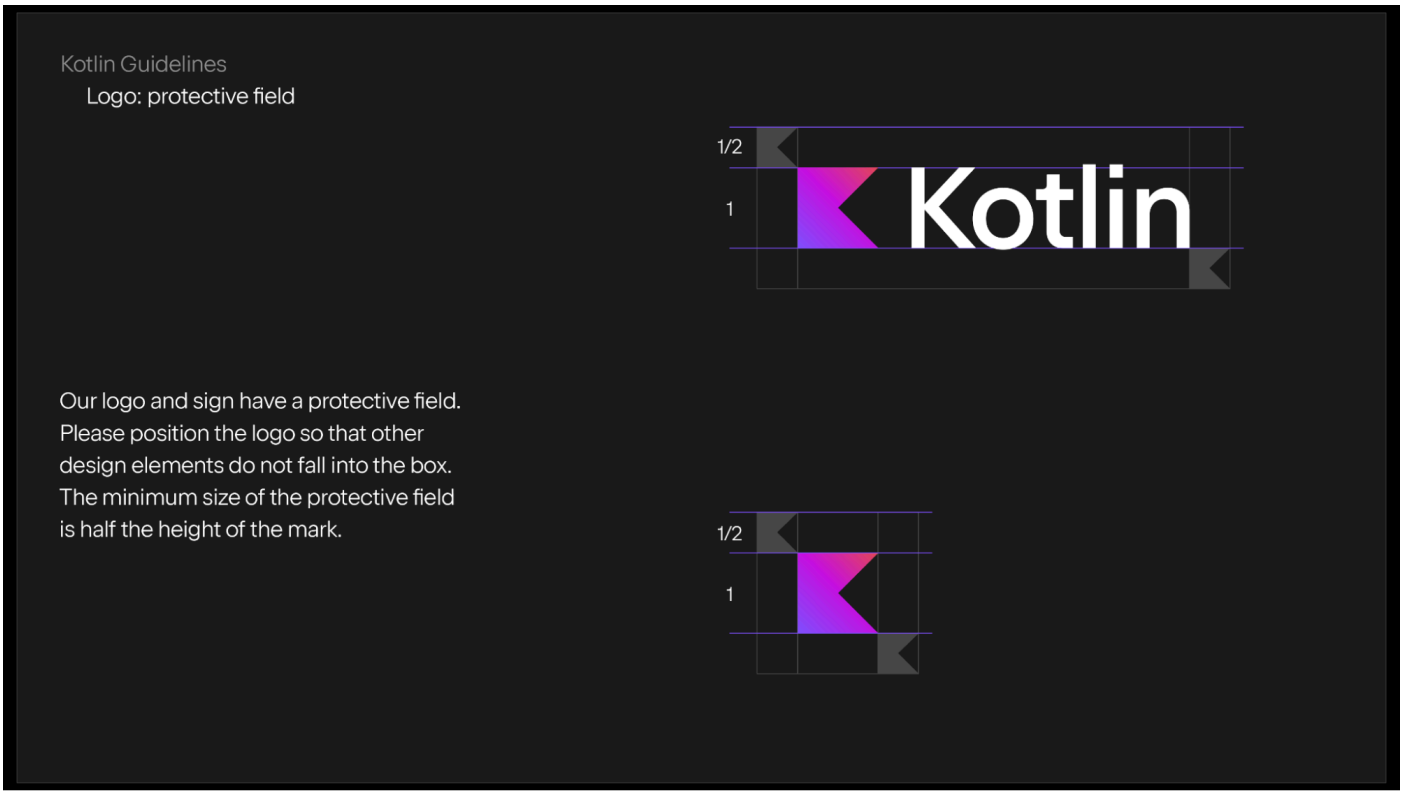
Kotlin brand assets

Kotlin logo

Our logo consists of a mark and a typeface. The full-color version is the main one and should be used in the vast majority of cases.

Download all versions

Our logo and mark have a protective field. Please position the logo so that other design elements do not come into the box. The minimum size of the protective field is half the height of the mark.



Kotlin logo proportions

Pay special attention to the following restrictions concerning the use of the logo:

- Do not separate the mark from the text. Do not swap elements.
- Do not change the transparency of the logo.
- Do not outline the logo.
- Do not repaint the logo in third-party colors.
- Do not change the text.
- Do not set the logo against a complex background. Do not place the logo in front of a bright background.

Read the Kotlin brand usage guidelines.

Kotlin ecosystem logos

The Kotlin brand package also includes logos for several official Kotlin ecosystem projects, libraries, frameworks, and technologies. These assets follow the same visual principles as the Kotlin logo and help ensure consistent representation across the ecosystem.

The package contains logos for:

- Kotlin Foundation
- Kotlin Multiplatform (KMP)
- Talking Kotlin
- Compose Multiplatform (CMP)
- Compose Hot Reload
- Ktor

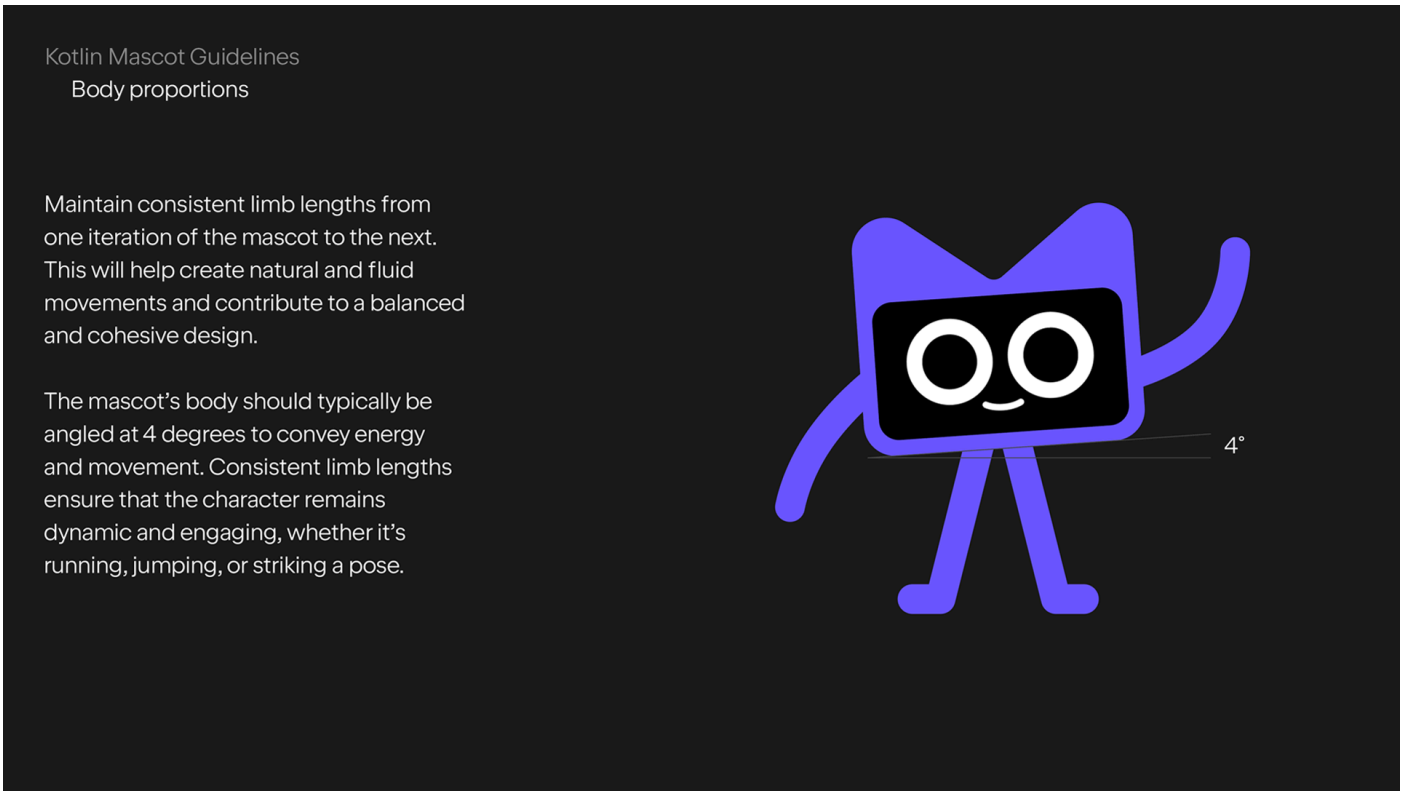
- Exposed

You can find these logos in the same download bundle:

Download all versions

Kotlin mascot

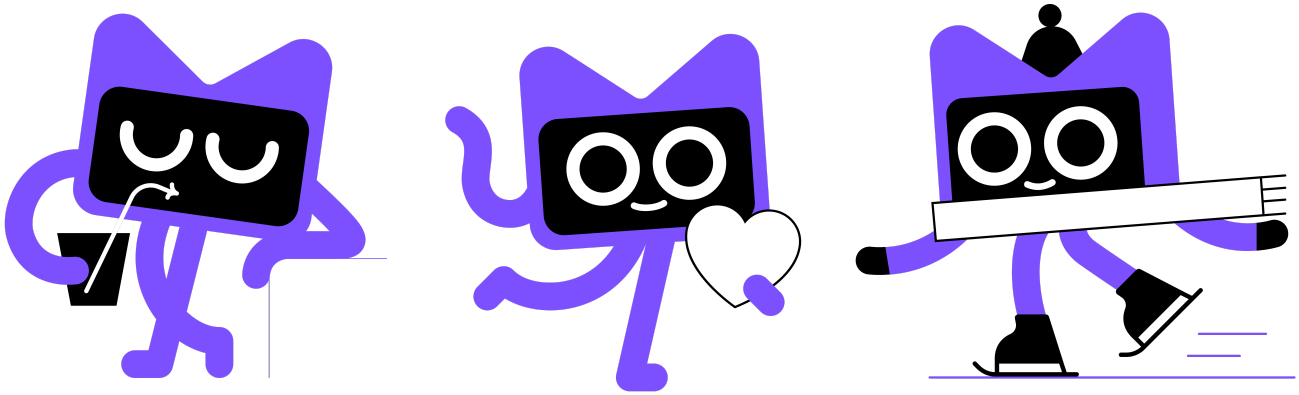
Meet Kodee, the Kotlin mascot and your friendly companion who's always there to encourage and inspire your creativity. When using it, we ask you to follow these simple guidelines.



Kotlin mascot Kodee proportions

You can use Kodee in your digital and print materials. For this purpose, we have prepared a variety of Kotlin mascot assets for you to download and explore.

Download all assets



Kotlin mascot Kodee in action

Kotlin User Group brand assets

We provide Kotlin user groups with a logo that is specifically designed to be recognizable and convey a reference to Kotlin.

- The official Kotlin logo is associated with the language itself. It should not be used otherwise in different scopes, as this could cause confusion. The same applies to its close derivatives.
- User groups logo also means that the opinions and actions of the community are independent of the Kotlin team.
- Your opinions don't have to agree with ours, and we think this is the most beneficial model for a creative and strong community.

Download all assets

Style for user groups

Since the launch of the Kotlin community support program at the beginning of 2017, the number of user groups has multiplied, with around 2–4 new user groups joining us every month. Please check out the complete list of groups in the **Kotlin User Groups** section to find one in your area.

We provide new Kotlin user groups with a user group logo and a profile picture.



Branding image

There are two main reasons why we are doing it:

- Firstly, we received many requests from the community asking for special Kotlin style branded materials to help them be recognized as officially dedicated user groups.
- Secondly, we wanted to provide a distinct style for the user group and community content to make it clear which Kotlin-related materials are from the official team and which are created by the community.

Create the logo of your user group

To create a logo of your users group:

1. Copy the Kotlin user group logo file to your Google Drive (you have to be signed in to your Google account).
2. Replace the **Your City** text with the name of your user group.
3. Download the picture and use it for the user group materials.



Belarusian Kotlin User Group sample

Belarusian Kotlin User Group Profile Picture sample

You can download a set of graphics including vector graphics and samples of cover pictures for social networks.

Create your group's profile picture for different platforms

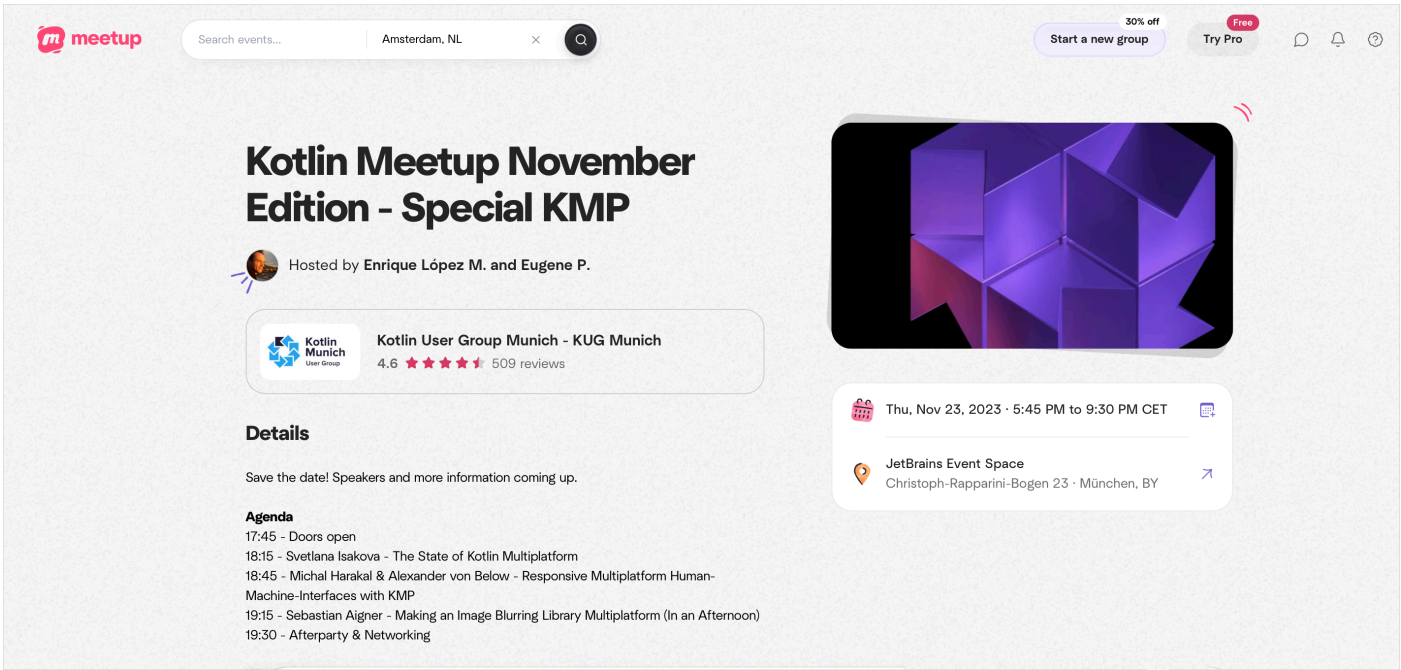
To create your group's profile picture:

1. Make a copy of the Kotlin user group profile picture file to your Google Drive (you have to be signed in to your Google account).
2. Add a shortened name of the user group's location (up to 4 capital symbols according to our default sample).
3. Download the picture and use it for your profiles on Facebook, Twitter, or any other platform.

Create meetup.com cover photo

To create a cover photo with a group's logo for meetup.com:

1. Make a copy of the picture file to your Google Drive (you have to be signed in to your Google account).
2. Add a shortened name of the user group's location to the logo on the right upper corner of the picture. If you want to replace the general pattern with a custom picture, click on the background pattern-picture, choose 'Replace Image', then 'Upload from Computer' or any other source.
3. Download the picture and use it for your profile on meetup.com.



User Group examples

Kotlin Night brand assets

JetBrains provides branding and materials for Kotlin Night events. Our team will prepare digital assets for the event promotion and ship your merchandise pack containing stickers and t-shirts. Check out what we have to make your Kotlin Night fun!

Download all assets

Social media

Stickers can be used to brand any media necessary for a Kotlin Night. Just stick them on anything you can get your hands on. It's fun!



Cover

Cover/Logo



Avatar



Example of usage

Cover Social

Branding stickers

Stickers can be used to brand assets for a Kotlin Night. Just stick them on anything you can get your hands on. It is funny!

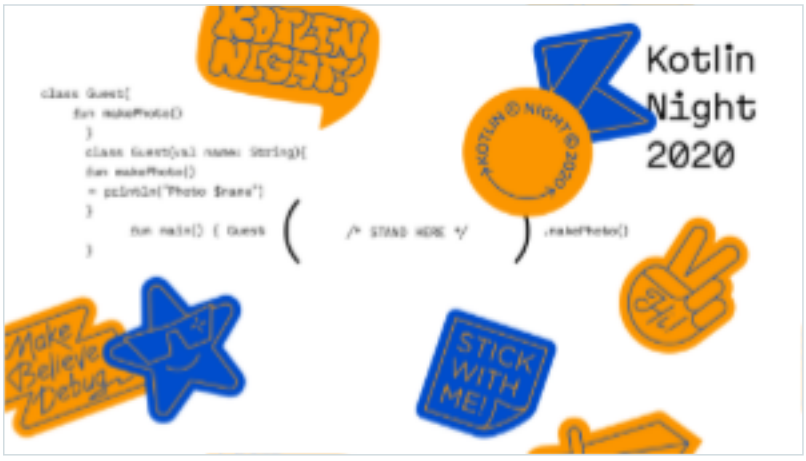


Example of usage

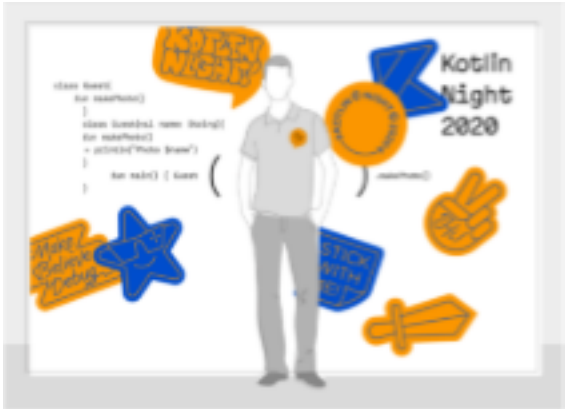
Stickers

Press-wall

You can decorate a press wall with stickers for unforgettable event pictures.



Press-wall



Example of usage

Press Wall

Sticky badges

Use stickers as badges for the attendees and boost networking at the event!

Board for stickers

Or you can provide a board where your guests can paste stickers with their impressions, feedback, and wishes.



board pack

T-shirts

Guests of the event are offered to paste stickers on the board with their impressions of the meeting. What does it mean to you?



Front print



Back print

Sticker pack

Kotlin documentation as PDF

Here you can download a PDF version of Kotlin documentation that includes everything except tutorials and API reference.

Download Kotlin 2.4.0 documentation (PDF)

View the latest Kotlin documentation (online)

Security

We do our best to make sure our products are free of security vulnerabilities. To reduce the risk of introducing a vulnerability, you can follow these best practices:

- Always use the latest Kotlin release. For security purposes, we sign our releases published on Maven Central with these PGP keys:
  - Key ID: **kt-a@jetbrains.com**
  - Fingerprint: **2FBA 29D0 8D2E 25EE 84C1 32C3 0729 A0AF F899 9A87**
  - Key size: **RSA 3072**
- Use the latest versions of your application's dependencies. If you need to use a specific version of a dependency, periodically check if any new security vulnerabilities have been discovered. You can follow the guidelines from GitHub or browse known vulnerabilities in the CVE base.

Starting with Kotlin 2.4.0, the Kotlin standard library for the JVM has an 18-month support window for each release line. Language releases (2. *x*. *0*) and the following tooling releases (2. *x*. *20*) belong to the same release line (2. *x*). To learn more about how we handle security vulnerabilities in the standard library for the JVM, see Standard library security support.

We always want to hear about any security issues you find. To report vulnerabilities that you discover in Kotlin, post a message directly to our issue tracker or send us an email.

For more information on how our responsible disclosure process works, see the JetBrains Coordinated Disclosure Policy.