

Kotlin Language Documentation 2.0.0

Table of Contents

Kotlin Docs	65
Get started with Kotlin	65
Install Kotlin	65
Choose your Kotlin use case	65
Is anything missing?	67
Welcome to our tour of Kotlin!	67
Hello world	67
Variables	68
String templates	68
Practice	68
Next step	69
Basic types	69
Practice	70
Next step	70
Collections	70
List	71
Set	72
Map	73
Practice	75
Next step	76
Control flow	76
Conditional expressions	76
Ranges	78
Loops	78
Practice	79
Next step	81

Functions	81
Named arguments	82
Default parameter values	82
Functions without return	82
Single-expression functions	83
Functions practice	83
Lambda expressions	84
Lambda expressions practice	87
Next step	87
Classes	87
Properties	88
Create instance	88
Access properties	89
Member functions	89
Data classes	89
Practice	91
Next step	92
Null safety	92
Nullable types	92
Check for null values	93
Use safe calls	93
Use Elvis operator	93
Practice	94
What's next?	94
Kotlin Multiplatform	94
Kotlin Multiplatform use cases	94
Code sharing between platforms	95
Get started	95
Kotlin for server side	96

Frameworks for server-side development with Kotlin	96
Deploying Kotlin server-side applications	96
Products that use Kotlin on the server side	97
Next steps	97
Kotlin for Android	97
Kotlin Wasm	97
Kotlin/Wasm performance	98
Browser API support	99
Leave feedback	99
Learn more	99
Kotlin Native	100
Why Kotlin/Native?	100
Target platforms	100
Interoperability	100
Sharing code between platforms	100
How to get started	101
Kotlin for JavaScript	101
Kotlin/JS IR compiler	101
Kotlin/JS frameworks	101
Join the Kotlin/JS community	102
Kotlin for data analysis	102
Notebooks	103
Kotlin DataFrame	104
Kandy	105
What's next	106
Kotlin for competitive programming	106
Simple example: Reachable Numbers problem	107

Functional operators example: Long Number problem	108
More tips and tricks	109
Learning Kotlin	109
What's new in Kotlin 2.0.0	110
IDE support	110
Kotlin K2 compiler	110
Kotlin/JVM	118
Kotlin/Native	119
Kotlin/Wasm	120
Kotlin/JS	121
Gradle improvements	125
Standard library	132
Install Kotlin 2.0.0	134
What's new in Kotlin 2.0.0-RC3	134
IDE support	134
Kotlin K2 compiler	134
Kotlin/JVM	141
Kotlin/Native	142
Kotlin/Wasm	142
Kotlin/JS	144
Gradle improvements	147
Standard library	153
What to expect from upcoming Kotlin EAP releases	155
How to update to Kotlin 2.0.0-RC3	155
What's new in Kotlin 1.9.20	155
IDE support	156
New Kotlin K2 compiler updates	156
Kotlin/JVM	157
Kotlin/Native	157

Kotlin Multiplatform	161
Kotlin/Wasm	169
Gradle	170
Standard library	171
Documentation updates	172
Install Kotlin 1.9.20	173
 What's new in Kotlin 1.9.0	173
IDE support	174
New Kotlin K2 compiler updates	174
Language	176
Kotlin/JVM	177
Kotlin/Native	178
Kotlin Multiplatform	180
Kotlin/Wasm	181
Kotlin/JS	183
Gradle	184
Standard library	187
Documentation updates	192
Install Kotlin 1.9.0	192
Compatibility guide for Kotlin 1.9.0	192
 What's new in Kotlin 1.8.20	193
IDE support	193
New Kotlin K2 compiler updates	193
Language	194
New Kotlin/Wasm target	198
Kotlin/JVM	199
Kotlin/Native	200
Kotlin Multiplatform	202
Kotlin/JavaScript	205

Gradle	206
Standard library	209
Serialization updates	211
Documentation updates	212
Install Kotlin 1.8.20	212
What's new in Kotlin 1.8.0	213
IDE support	213
Kotlin/JVM	213
Kotlin/Native	214
Kotlin Multiplatform: A new Android source set layout	215
Kotlin/JS	217
Gradle	219
Standard library	221
Documentation updates	224
Install Kotlin 1.8.0	224
Compatibility guide for Kotlin 1.8.0	225
What's new in Kotlin 1.7.20	225
Support for Kotlin K2 compiler plugins	225
Language	226
Kotlin/JVM	230
Kotlin/Native	232
Kotlin/JS	233
Gradle	233
Standard library	234
Documentation updates	236
Install Kotlin 1.7.20	236
What's new in Kotlin 1.7.0	237
New Kotlin K2 compiler for the JVM in Alpha	237

Language	238
Kotlin/JVM	240
Kotlin/Native	240
Kotlin/JS	242
Standard library	243
Gradle	247
Migrating to Kotlin 1.7.0	251
What's new in Kotlin 1.6.20	251
Language	252
Kotlin/JVM	253
Kotlin/Native	255
Kotlin Multiplatform	258
Kotlin/JS	260
Security	261
Gradle	263
What's new in Kotlin 1.6.0	264
Language	264
Supporting previous API versions for a longer period	267
Kotlin/JVM	267
Kotlin/Native	268
Kotlin/JS	270
Kotlin Gradle plugin	271
Standard library	271
Tools	274
Coroutines 1.6.0-RC	275
Migrating to Kotlin 1.6.0	275
What's new in Kotlin 1.5.30	275
Language features	276
Kotlin/JVM	279

Kotlin/Native	280
Kotlin Multiplatform	282
Kotlin/JS	284
Gradle	284
Standard library	287
Serialization 1.3.0-RC	289
What's new in Kotlin 1.5.20	290
Kotlin/JVM	290
Kotlin/Native	291
Kotlin/JS	292
Gradle	293
Standard library	293
What's new in Kotlin 1.5.0	294
Language features	294
Kotlin/JVM	296
Kotlin/Native	298
Kotlin/JS	299
Kotlin Multiplatform	299
Standard library	299
kotlin-test library	303
kotlinx libraries	306
Migrating to Kotlin 1.5.0	307
What's new in Kotlin 1.4.30	307
Language features	307
Kotlin/JVM	310
Kotlin/Native	310
Kotlin/JS	311
Gradle project improvements	311

Standard library	311
Serialization updates	313
What's new in Kotlin 1.4.20	313
Kotlin/JVM	313
Kotlin/JS	314
Kotlin/Native	316
Kotlin Multiplatform	317
Standard library	318
Kotlin Android Extensions	318
What's new in Kotlin 1.4.0	319
Language features and improvements	319
New tools in the IDE	322
New compiler	325
Kotlin/JVM	327
Kotlin/JS	329
Kotlin/Native	329
Kotlin Multiplatform	331
Gradle project improvements	334
Standard library	335
Stable JSON serialization	341
Scripting and REPL	341
Migrating to Kotlin 1.4.0	342
What's new in Kotlin 1.3	342
Coroutines release	342
Kotlin/Native	343
Multiplatform projects	343
Contracts	343
Capturing when subject in a variable	344
@JvmStatic and @JvmField in companions of interfaces	344

Nested declarations in annotation classes	345
Parameterless main	345
Functions with big arity	345
Progressive mode	346
Inline classes	346
Unsigned integers	346
@JvmDefault	347
Standard library	347
Tooling	349
What's new in Kotlin 1.2	349
Table of contents	349
Multiplatform projects (experimental)	350
Other language features	350
Standard library	353
JVM backend	355
JavaScript backend	355
Tools	355
What's new in Kotlin 1.1	356
Table of contents	356
JavaScript	356
Coroutines (experimental)	356
Other language features	357
Standard library	361
JVM Backend	364
JavaScript backend	364
Kotlin releases	365
Update to a new release	365
IDE support	366

Kotlin release compatibility	366
Release details	366
Kotlin roadmap	373
Key priorities	374
Kotlin roadmap by subsystem	374
What's changed since July 2023	375
Basic syntax	376
Package definition and imports	376
Program entry point	377
Print to the standard output	377
Functions	377
Variables	378
Creating classes and instances	379
Comments	379
String templates	380
Conditional expressions	380
for loop	380
while loop	381
when expression	381
Ranges	381
Collections	382
Nullable values and null checks	383
Type checks and automatic casts	384
Idioms	385
Create DTOs (POJOs/POCOs)	385
Default values for function parameters	385
Filter a list	385
Check the presence of an element in a collection	385
String interpolation	385

Instance checks	386
Read-only list	386
Read-only map	386
Access a map entry	386
Traverse a map or a list of pairs	386
Iterate over a range	386
Lazy property	386
Extension functions	386
Create a singleton	387
Use inline value classes for type-safe values	387
Instantiate an abstract class	387
If-not-null shorthand	387
If-not-null-else shorthand	387
Execute a statement if null	388
Get first item of a possibly empty collection	388
Execute if not null	388
Map nullable value if not null	388
Return on when statement	388
try-catch expression	388
if expression	388
Builder-style usage of methods that return Unit	389
Single-expression functions	389
Call multiple methods on an object instance (with)	389
Configure properties of an object (apply)	389
Java 7's try-with-resources	390
Generic function that requires the generic type information	390
Swap two variables	390
Mark code as incomplete (TODO)	390
What's next?	390

Coding conventions	390
Configure style in IDE	390
Source code organization	391
Naming rules	392
Formatting	394
Documentation comments	401
Avoid redundant constructs	402
Idiomatic use of language features	402
Coding conventions for libraries	405
Basic types	406
Numbers	406
Integer types	406
Floating-point types	406
Literal constants for numbers	407
Numbers representation on the JVM	408
Explicit number conversions	408
Operations on numbers	409
Unsigned integer types	410
Unsigned arrays and ranges	411
Unsigned integers literals	411
Use cases	412
Booleans	412
Characters	413
Strings	413
String literals	414
String templates	415
String formatting	415

Arrays	416
When to use arrays	416
Create arrays	417
Access and modify elements	418
Work with arrays	418
Primitive-type arrays	420
What's next?	421
Type checks and casts	421
is and !is operators	422
Smart casts	422
"Unsafe" cast operator	424
"Safe" (nullable) cast operator	425
Conditions and loops	425
If expression	425
When expression	425
For loops	427
While loops	428
Break and continue in loops	428
Returns and jumps	428
Break and continue labels	428
Return to labels	429
Exceptions	430
Exception classes	430
Checked exceptions	431
The Nothing type	431
Java interoperability	432
Packages and imports	432
Default imports	432

Imports	433
Visibility of top-level declarations	433
Classes	433
Constructors	433
Creating instances of classes	435
Class members	435
Inheritance	436
Abstract classes	436
Companion objects	436
Inheritance	436
Overriding methods	437
Overriding properties	437
Derived class initialization order	438
Calling the superclass implementation	438
Overriding rules	439
Properties	439
Declaring properties	439
Getters and setters	439
Compile-time constants	441
Late-initialized properties and variables	441
Overriding properties	442
Delegated properties	442
Interfaces	442
Implementing interfaces	442
Properties in interfaces	442
Interfaces Inheritance	443
Resolving overriding conflicts	443
Functional (SAM) interfaces	444

SAM conversions	444
Migration from an interface with constructor function to a functional interface	444
Functional interfaces vs. type aliases	445
Visibility modifiers	445
Packages	445
Class members	446
Modules	447
Extensions	447
Extension functions	447
Extensions are resolved statically	448
Nullable receiver	449
Extension properties	449
Companion object extensions	449
Scope of extensions	449
Declaring extensions as members	450
Note on visibility	451
Data classes	451
Properties declared in the class body	451
Copying	452
Data classes and destructuring declarations	452
Standard data classes	452
Sealed classes and interfaces	452
Declare a sealed class or interface	453
Inheritance	455
Use sealed classes with when expression	455
Use case scenarios	456
Generics: in, out, where	458
Variance	458

Type projections	460
Generic functions	461
Generic constraints	461
Definitely non-nullable types	462
Type erasure	462
Underscore operator for type arguments	464
Nested and inner classes	464
Inner classes	465
Anonymous inner classes	465
Enum classes	465
Anonymous classes	465
Implementing interfaces in enum classes	466
Working with enum constants	466
Inline value classes	467
Members	467
Inheritance	468
Representation	468
Inline classes vs type aliases	469
Inline classes and delegation	470
Object expressions and declarations	470
Object expressions	470
Object declarations	472
Delegation	475
Overriding a member of an interface implemented by delegation	475
Delegated properties	476
Standard delegates	477
Delegating to another property	477
Storing properties in a map	478

Local delegated properties	479
Property delegate requirements	479
Translation rules for delegated properties	480
Providing a delegate	481
Type aliases	482
Functions	483
Function usage	483
Function scope	487
Generic functions	487
Tail recursive functions	488
Higher-order functions and lambdas	488
Higher-order functions	488
Function types	489
Lambda expressions and anonymous functions	491
Inline functions	493
<code>noinline</code>	494
Non-local returns	494
Reified type parameters	495
Inline properties	495
Restrictions for public API inline functions	496
Operator overloading	496
Unary operations	496
Binary operations	498
Infix calls for named functions	501
Type-safe builders	501
How it works	501
Scope control: <code>@DslMarker</code>	503
Full definition of the <code>com.example.html</code> package	504

Using builders with builder type inference	505
Writing your own builders	505
How builder inference works	507
Null safety	509
Nullable types and non-nullable types	509
Checking for null in conditions	510
Safe calls	510
Nullable receiver	510
Elvis operator	511
The !! operator	511
Safe casts	511
Collections of a nullable type	511
What's next?	512
Equality	512
Structural equality	512
Referential equality	513
Floating-point numbers equality	513
Array equality	513
This expressions	513
Qualified this	514
Implicit this	514
Asynchronous programming techniques	514
Threading	515
Callbacks	515
Futures, promises, and others	515
Reactive extensions	516
Coroutines	516
Coroutines	517

How to start	517
Sample projects	518
Annotations	518
Usage	518
Constructors	518
Instantiation	519
Lambdas	519
Annotation use-site targets	519
Java annotations	520
Repeatable annotations	522
Destructuring declarations	522
Example: returning two values from a function	523
Example: destructuring declarations and maps	523
Underscore for unused variables	523
Destructuring in lambdas	523
Reflection	524
JVM dependency	524
Class references	525
Callable references	525
Get started with Kotlin Multiplatform	528
Start from scratch	528
Dive deep into Kotlin Multiplatform	529
Get help	529
The basics of Kotlin Multiplatform project structure	529
Common code	529
Targets	530
Source sets	532
Integration with tests	536

What's next?	537
Advanced concepts of the multiplatform project structure	537
Dependencies and dependsOn	537
Declaring custom source sets	541
Compilations	543
Set up targets for Kotlin Multiplatform	544
Distinguish several targets for one platform	544
Share code on platforms	545
Share code on all platforms	545
Share code on similar platforms	545
Share code in libraries	546
Connect platform-specific libraries	546
What's next?	546
Expected and actual declarations	546
Rules for expected and actual declarations	547
Different approaches for using expected and actual declarations	547
Advanced use cases	552
What's next?	553
Hierarchical project structure	554
Default hierarchy template	554
Manual configuration	558
Adding dependencies on multiplatform libraries	560
Dependency on a Kotlin library	560
Dependency on Kotlin Multiplatform libraries	562
Dependency on another multiplatform project	563
What's next?	563
Adding Android dependencies	564

What's next?	564
Adding iOS dependencies	565
With CocoaPods	565
Without CocoaPods	566
What's next?	568
Configure compilations	568
Configure all compilations	569
Configure compilations for one target	569
Configure one compilation	570
Create a custom compilation	570
Use Java sources in JVM compilations	571
Configure interop with native languages	572
Compilation for Android	573
Compilation of the source set hierarchy	574
Build final native binaries (Experimental DSL)	574
Declare binaries	575
Configure binaries	576
Build final native binaries	579
Declare binaries	580
Access binaries	581
Export dependencies to binaries	582
Build universal frameworks	584
Build XCFrameworks	584
Customize the Info.plist file	585
Publishing multiplatform libraries	586
Structure of publications	586
Host requirements	587
Publish an Android library	588

Disable sources publication	588
Disable JVM environment attribute publication	589
Multiplatform Gradle DSL reference	589
Id and version	589
Top-level blocks	589
Targets	590
Source sets	596
Compilations	598
Dependencies	601
Language settings	602
Android source set layout	603
Check the compatibility	603
Rename Kotlin source sets	603
Move source files	603
Move the AndroidManifest.xml file	604
Check the relationship between Android and common tests	604
Adjust the implementation of Android flavors	605
Compatibility guide for Kotlin Multiplatform	605
Version compatibility	605
New approach to auto-generated targets	605
Changes in Gradle input and output compile tasks	606
New configuration names for dependencies on the compilation	606
Deprecated Gradle properties for hierarchical structure support	607
Deprecated support of multiplatform libraries published in the legacy mode	608
Deprecated API for adding Kotlin source sets directly to the Kotlin compilation	608
Migration from kotlin-js Gradle plugin to kotlin-multiplatform Gradle plugin	609
Rename of android target to androidTarget	611
Declaring several similar targets	611
Deprecated jvmWithJava preset	613

Deprecated legacy Android source set layout	613
Deprecated commonMain and commonTest with custom dependsOn	614
Deprecated target presets API	614
New approach to forward declarations	615
Kotlin Multiplatform Mobile plugin releases	616
Update to the new release	616
Release details	616
Get started with Kotlin Notebook	621
Next step	621
Set up an environment	621
Set up the environment	621
Next step	622
Create your first Kotlin Notebook	622
Create an empty project	622
Create a Kotlin Notebook	623
Next step	625
Add dependencies to your Kotlin Notebook	625
Add Kotlin DataFrame and Kandy libraries to your Kotlin Notebook	626
Next step	628
Share your Kotlin Notebook	628
Share a Kotlin Notebook	629
What's next	630
Output formats supported by Kotlin Notebook	630
Texts	631
HTML	633
Images	633
Math formulas and equations	635

Data frames	635
Charts	636
What's next	637
Retrieve data from files	637
Before you start	637
Retrieve data from a file	638
Display data	638
Refine data	639
Save DataFrame	640
What's next	641
Retrieve data from web sources and APIs	641
Before you start	641
Fetch data from an API	641
Clean and refine data	642
Analyze data in Kotlin Notebook	643
What's next	645
Connect and retrieve data from databases	645
Before you start	645
Connect to database	646
Retrieve and manipulate data	646
Analyze data in Kotlin Notebook	647
What's next	647
Data visualization in Kotlin Notebook with Kandy	648
Before you start	648
Create the DataFrame	648
Create a line chart	649
Create a points chart	650
Create a bar chart	651

What's next	652
Kotlin and Java libraries for data analysis	652
Kotlin libraries	652
Java libraries	653
Get started with Kotlin/JVM	655
Create a project	655
Create an application	657
Run the application	658
What's next?	659
Comparison to Java	659
Some Java issues addressed in Kotlin	659
What Java has that Kotlin does not	659
What Kotlin has that Java does not	660
What's next?	660
Calling Java from Kotlin	660
Getters and setters	661
Java synthetic property references	661
Methods returning void	662
Escaping for Java identifiers that are keywords in Kotlin	662
Null-safety and platform types	662
Mapped types	667
Java generics in Kotlin	669
Java arrays	669
Java varargs	670
Operators	670
Checked exceptions	671
Object methods	671
Inheritance from Java classes	672

Accessing static members	672
Java reflection	672
SAM conversions	672
Using JNI with Kotlin	672
Using Lombok-generated declarations in Kotlin	673
Calling Kotlin from Java	673
Properties	673
Package-level functions	673
Instance fields	674
Static fields	675
Static methods	675
Default methods in interfaces	676
Visibility	678
KClass	678
Handling signature clashes with @JvmName	678
Overloads generation	679
Checked exceptions	679
Null-safety	680
Variant generics	680
Get started with Spring Boot and Kotlin	681
Next step	681
Join the community	681
Create a Spring Boot project with Kotlin	681
Before you start	681
Create a Spring Boot project	682
Explore the project Gradle build file	685
Explore the generated Spring Boot application	686
Create a controller	687
Run the application	688

Next step	689
Add a data class to Spring Boot project	689
Update your application	689
Run the application	691
Next step	691
Add database support for Spring Boot project	691
Add database support	691
Update the MessageController class	692
Update the MessageService class	693
Configure the database	693
Add messages to database via HTTP request	694
Retrieve messages by id	696
Run the application	697
Next step	698
Use Spring Data CrudRepository for database access	698
Update your application	699
Run the application	700
What's next	700
Test code using JUnit in JVM – tutorial	700
Add dependencies	700
Add the code to test it	701
Create a test	701
Run a test	702
What's next	703
Mixing Java and Kotlin in one project – tutorial	704
Adding Java source code to an existing Kotlin project	704
Adding Kotlin source code to an existing Java project	705
Converting an existing Java file to Kotlin with J2K	706

Using Java records in Kotlin	706
Using Java records from Kotlin code	707
Declare records in Kotlin	707
Further discussion	707
Strings in Java and Kotlin	708
Concatenate strings	708
Build a string	708
Create a string from collection items	708
Set default value if the string is blank	709
Replace characters at the beginning and end of a string	709
Replace occurrences	710
Split a string	710
Take a substring	710
Use multiline strings	711
What's next?	712
Collections in Java and Kotlin	712
Operations that are the same in Java and Kotlin	712
Operations that differ a bit	714
Operations that don't exist in Java's standard library	715
Mutability	716
Covariance	717
Ranges and progressions	717
Comparison by several criteria	718
Sequences	719
Removal of elements from a list	719
Traverse a map	720
Get the first and the last items of a possibly empty collection	720
Create a set from a list	720
Group elements	721

Filter elements	721
Collection transformation operations	722
What's next?	723
Nullability in Java and Kotlin	723
Support for nullable types	724
Platform types	725
Support for definitely non-nullable types	725
Checking the result of a function call	726
Default values instead of null	726
Functions returning a value or null	727
Aggregate operations	727
Casting types safely	727
What's next?	728
Get started with Kotlin/Native in IntelliJ IDEA	728
Before you start	728
Build and run the application	729
Update the application	730
What's next?	732
Get started with Kotlin/Native using Gradle	732
Create project files	733
Build and run the application	734
Open the project in an IDE	734
What's next?	734
Get started with Kotlin/Native using the command-line compiler	734
Obtain the compiler	734
Write "Hello Kotlin/Native" program	734
Compile the code from the console	735
Interoperability with C	735

Platform libraries	735
Simple example	735
Create bindings for a new library	736
Bindings	738
Mapping primitive data types from C – tutorial	742
Types in C language	742
Example C library	742
Inspect generated Kotlin APIs for a C library	743
Primitive types in kotlin	744
Fix the code	745
Next steps	745
Mapping struct and union types from C – tutorial	746
Mapping struct and union C types	746
Inspect Generated Kotlin APIs for a C library	746
Struct and union types in Kotlin	748
Use struct and union types from Kotlin	748
Run the code	750
Next steps	750
Mapping function pointers from C – tutorial	751
Mapping function pointer types from C	751
Inspect generated Kotlin APIs for a C library	751
C function pointers in Kotlin	753
Pass Kotlin function as C function pointer	753
Use the C function pointer from Kotlin	753
Fix the code	753
Next Steps	754
Mapping Strings from C – tutorial	754
Working with C strings	754

Inspect generated Kotlin APIs for a C library	755
Strings in Kotlin	756
Pass Kotlin string to C	756
Read C Strings in Kotlin	757
Receive C string bytes from Kotlin	757
Fix the Code	757
Next steps	758
Create an app using C Interop and libcurl – tutorial	758
Before you start	758
Create a definition file	759
Add interoperability to the build process	760
Write the application code	761
Compile and run the application	761
Interoperability with Swift/Objective-C	762
Usage	762
Mappings	763
Casting between mapped types	770
Subclassing	770
C features	770
Export of KDoc comments to generated Objective-C headers	770
Unsupported	771
Kotlin/Native as an Apple framework – tutorial	772
Create a Kotlin library	772
Generated framework headers	774
Garbage collection and reference counting	777
Use the code from Objective-C	777
Use the code from Swift	777
Xcode and framework dependencies	778
Next steps	778

CocoaPods overview and setup	778
Set up an environment to work with CocoaPods	779
Add and configure Kotlin CocoaPods Gradle plugin	780
Update Podfile for Xcode	781
Possible issues and solutions	781
Add dependencies on a Pod library	782
From the CocoaPods repository	783
On a locally stored library	783
From a custom Git repository	784
From a custom Podspec repository	785
With custom cinterop options	785
Use a Kotlin Gradle project as a CocoaPods dependency	787
Xcode project with one target	787
Xcode project with several targets	788
CocoaPods Gradle plugin DSL reference	788
Enable the plugin	789
cocoapods block	789
pod() function	791
Swift package export setup	792
Prepare file locations	792
Create the XCFramework and the Swift package manifest	793
Exporting multiple modules as an XCFramework	794
Kotlin/Native libraries	795
Kotlin compiler specifics	795
cinterop tool specifics	795
klib utility	795
Several examples	796

Advanced topics	797
Platform libraries	798
POSIX bindings	798
Popular native libraries	798
Availability by default	798
Kotlin/Native as a dynamic library – tutorial	798
Create a Kotlin library	799
Generated headers file	800
Use generated headers from C	803
Compile and run the example on Linux and macOS	804
Compile and run the example on Windows	804
Next steps	804
Kotlin/Native memory management	805
Garbage collector	805
Memory consumption	806
Unit tests in the background	807
What's next	808
iOS integration	808
Threads	808
Garbage collection and lifecycle	809
Support for background state and App Extensions	811
Migrate to the new memory manager	811
Update Kotlin	812
Update dependencies	812
Update your code	812
Support both new and legacy memory managers	813
What's next	813
Debugging Kotlin/Native	813

Produce binaries with debug info with Kotlin/Native compiler	814
Breakpoints	814
Stepping	815
Variable inspection	815
Known issues	816
Symbolicating iOS crash reports	816
Producing .dSYM for release Kotlin binaries	817
Make frameworks static when using rebuild from bitcode	817
Kotlin/Native target support	817
Tier 1	818
Tier 2	818
Tier 3	819
For library authors	820
Privacy manifest for iOS apps	820
What's the issue	820
How to resolve	820
Tips for improving Kotlin/Native compilation times	821
General recommendations	821
Gradle configuration	821
Windows OS configuration	822
License files for the Kotlin/Native binaries	822
Kotlin/Native FAQ	823
How do I run my program?	823
What is Kotlin/Native memory management model?	823
How do I create a shared library?	823
How do I create a static library or an object file?	824
How do I run Kotlin/Native behind a corporate proxy?	824

How do I specify a custom Objective-C prefix/name for my Kotlin framework?	824
How do I rename the iOS framework?	824
How do I enable bitcode for my Kotlin framework?	825
Why do I see InvalidMutabilityException?	825
How do I make a singleton object mutable?	825
How can I compile my project with unreleased versions of Kotlin/Native?	825
Get started with Kotlin/Wasm in IntelliJ IDEA	825
Before you start	826
Open the project in IntelliJ IDEA	827
Run the application	827
Generate artifacts	829
Publish on GitHub pages	831
What's next?	832
Debug Kotlin/Wasm code	832
Before you start	832
Open the project in IntelliJ IDEA	834
Run the application	834
Debug in your browser	836
Leave feedback	839
What's next?	839
Interoperability with JavaScript	840
Use JavaScript code in Kotlin	840
Use Kotlin code in JavaScript	843
Type correspondence	843
Exception handling	845
Kotlin/Wasm and Kotlin/JS interoperability differences	845
Troubleshooting	846
Browser versions	847

Wasm proposals support	847
Set up a Kotlin/JS project	848
Execution environments	849
Support for ES2015 features	849
Dependencies	849
run task	851
test task	852
webpack bundling	853
CSS	854
Node.js	855
Yarn	856
Distribution target directory	858
Module name	858
package.json customization	859
Run Kotlin/JS	859
Run the Node.js target	859
Run the browser target	860
Development server and continuous compilation	861
Debug Kotlin/JS code	863
Debug in browser	863
Debug in the IDE	865
Debug in Node.js	868
What's next?	868
If you run into any problems	868
Run tests in Kotlin/JS	868
Kotlin/JS dead code elimination	872
Exclude declarations from DCE	872
Disable DCE	873

Kotlin/JS IR compiler	873
Lazy initialization of top-level properties	874
Incremental compilation for development binaries	874
Output mode	874
Ignoring compilation errors	875
Minification of member names in production	875
Preview: generation of TypeScript declaration files (d.ts)	875
Current limitations of the IR compiler	876
Migrating existing projects to the IR compiler	876
Authoring libraries for the IR compiler with backwards compatibility	876
Migrating Kotlin/JS projects to the IR compiler	877
Convert JS- and React-related classes and interfaces to external interfaces	877
Convert properties of external interfaces to var	877
Convert functions with receivers in external interfaces to regular functions	878
Create plain JS objects for interoperability	878
Replace <code>toString()</code> calls on function references with <code>.name</code>	878
Explicitly specify <code>binaries.executable()</code> in the build script	879
Additional troubleshooting tips when working with the Kotlin/JS IR compiler	879
Browser and DOM API	879
Interaction with the DOM	879
Use JavaScript code from Kotlin	880
Inline JavaScript	880
external modifier	880
Equality	883
Dynamic type	883
Use dependencies from npm	884
Use Kotlin code from JavaScript	885

Isolating declarations in a separate JavaScript object in plain mode	885
Package structure	885
Kotlin types in JavaScript	886
JavaScript modules	888
Browser targets	888
JavaScript libraries and Node.js files	889
@JsModule annotation	889
Kotlin/JS reflection	891
Class references	891
KType and typeOf()	891
KClass and createInstance()	891
Example	891
Typesafe HTML DSL	892
Build a web application with React and Kotlin/JS – tutorial	893
Before you start	893
Create a web app draft	896
Design app components	901
Compose components	905
Add more components	906
Use packages from npm	908
Use an external REST API	911
Deploy to production and the cloud	914
What's next	915
Get started with Kotlin custom scripting – tutorial	916
Project structure	916
Before you start	916
Create a project	916
Add scripting modules	917

Create a script definition	919
Create a scripting host	921
Run scripts	922
What's next?	923
Collections overview	923
Collection types	923
Constructing collections	927
Construct from elements	927
Create with collection builder functions	928
Empty collections	928
Initializer functions for lists	928
Concrete type constructors	928
Copy	929
Invoke functions on other collections	929
Iterators	930
List iterators	931
Mutable iterators	931
Ranges and progressions	932
Progression	932
Sequences	933
Construct	934
Sequence operations	934
Sequence processing example	935
Collection operations overview	936
Extension and member functions	936
Common operations	936
Write operations	937

Collection transformation operations	938
Map	938
Zip	938
Associate	939
Flatten	940
String representation	940
Filtering collections	941
Filter by predicate	941
Partition	942
Test predicates	942
Plus and minus operators	943
Grouping	943
Retrieve collection parts	944
Slice	944
Take and drop	944
Chunked	945
Windowed	945
Retrieve single elements	946
Retrieve by position	946
Retrieve by condition	947
Retrieve with selector	947
Random element	948
Check element existence	948
Ordering	948
Natural order	949
Custom orders	949
Reverse order	950

Random order	950
Aggregate operations	951
Fold and reduce	951
Collection write operations	953
Adding elements	953
Removing elements	953
Updating elements	954
List-specific operations	954
Retrieve elements by index	954
Retrieve list parts	955
Find element positions	955
List write operations	956
Set-specific operations	958
Map-specific operations	959
Retrieve keys and values	959
Filter	959
Plus and minus operators	960
Map write operations	960
Opt-in requirements	962
Opt in to using API	962
Require opt-in for API	964
Opt-in requirements for pre-stable APIs	965
Scope functions	966
Function selection	966
Distinctions	967
Functions	970
takelf and takeUnless	972

Time measurement	974
Calculate duration	974
Measure time	977
Time sources	978
Coroutines guide	979
Table of contents	979
Additional references	980
Coroutines basics	980
Your first coroutine	980
Extract function refactoring	981
Scope builder	981
Scope builder and concurrency	982
An explicit job	982
Coroutines are light-weight	983
Coroutines and channels – tutorial	983
Before you start	984
Blocking requests	985
Callbacks	988
Suspending functions	992
Coroutines	993
Concurrency	995
Structured concurrency	998
Showing progress	1001
Channels	1003
Testing coroutines	1006
What's next	1009
Cancellation and timeouts	1009
Cancelling coroutine execution	1009

Cancellation is cooperative	1010
Making computation code cancellable	1011
Closing resources with finally	1011
Run non-cancellable block	1012
Timeout	1012
Asynchronous timeout and resources	1013
Composing suspending functions	1014
Sequential by default	1015
Concurrent using <code>async</code>	1015
Lazily started <code>async</code>	1016
Async-style functions	1017
Structured concurrency with <code>async</code>	1018
Coroutine context and dispatchers	1019
Dispatchers and threads	1019
Unconfined vs confined dispatcher	1020
Debugging coroutines and threads	1021
Jumping between threads	1022
Job in the context	1023
Children of a coroutine	1023
Parental responsibilities	1024
Naming coroutines for debugging	1024
Combining context elements	1025
Coroutine scope	1025
Asynchronous Flow	1027
Representing multiple values	1027
Flows are cold	1029
Flow cancellation basics	1030
Flow builders	1030
Intermediate flow operators	1031

Terminal flow operators	1032
Flows are sequential	1033
Flow context	1034
Buffering	1035
Composing multiple flows	1038
Flattening flows	1039
Flow exceptions	1041
Exception transparency	1042
Flow completion	1044
Imperative versus declarative	1046
Launching flow	1046
Flow and Reactive Streams	1049
Channels	1049
Channel basics	1049
Closing and iteration over channels	1049
Building channel producers	1050
Pipelines	1050
Prime numbers with pipeline	1051
Fan-out	1052
Fan-in	1053
Buffered channels	1054
Channels are fair	1054
Ticker channels	1055
Coroutine exceptions handling	1056
Exception propagation	1056
CoroutineExceptionHandler	1057
Cancellation and exceptions	1057
Exceptions aggregation	1059
Supervision	1060

Shared mutable state and concurrency	1062
The problem	1062
Volatile are of no help	1063
Thread-safe data structures	1063
Thread confinement fine-grained	1064
Thread confinement coarse-grained	1065
Mutual exclusion	1065
Select expression (experimental)	1066
Selecting from channels	1066
Selecting on close	1067
Selecting to send	1069
Selecting deferred values	1070
Switch over a channel of deferred values	1070
Debug coroutines using IntelliJ IDEA – tutorial	1072
Create coroutines	1072
Debug coroutines	1073
Debug Kotlin Flow using IntelliJ IDEA – tutorial	1076
Create a Kotlin flow	1076
Debug the coroutine	1077
Add a concurrently running coroutine	1080
Debug a Kotlin flow with two coroutines	1080
Serialization	1081
Libraries	1081
Formats	1082
Example: JSON serialization	1082
What's next	1083
Lincheck guide	1084
Add Lincheck to your project	1084

Explore Lincheck	1084
Additional references	1085
Write your first test with Lincheck	1085
Create a project	1085
Add required dependencies	1085
Write a concurrent counter and run the test	1085
Trace the invalid execution	1086
Test the Java standard library	1087
Next step	1088
See also	1088
Stress testing and model checking	1088
Stress testing	1089
Model checking	1090
Which testing strategy is better?	1091
Configure the testing strategy	1091
Scenario minimization	1092
Logging data structure states	1092
Next step	1093
Operation arguments	1093
Next step	1095
Data structure constraints	1095
Next step	1096
Progress guarantees	1096
Next step	1098
Sequential specification	1098
Keywords and operators	1099
Hard keywords	1099

Soft keywords	1100
Modifier keywords	1101
Special identifiers	1102
Operators and special symbols	1102
Gradle	1103
What's next?	1103
Get started with Gradle and Kotlin/JVM	1103
Create a project	1103
Explore the build script	1105
Run the application	1106
What's next?	1108
Configure a Gradle project	1108
Apply the plugin	1109
Targeting the JVM	1110
Targeting multiple platforms	1116
Targeting Android	1116
Targeting JavaScript	1116
Triggering configuration actions with the KotlinBasePlugin interface	1117
Configure dependencies	1117
Declare repositories	1123
What's next?	1123
Compiler options in the Kotlin Gradle plugin	1124
How to define options	1124
All compiler options	1126
What's next?	1129
Compilation and caches in the Kotlin Gradle plugin	1129
Incremental compilation	1130
Gradle build cache support	1131

Gradle configuration cache support	1132
The Kotlin daemon and how to use it with Gradle	1132
Rolling back to the previous compiler	1133
Defining Kotlin compiler execution strategy	1134
Kotlin compiler fallback strategy	1135
Trying the latest language version	1135
Build reports	1136
What's next?	1137
Support for Gradle plugin variants	1137
Troubleshooting	1138
What's next?	1140
Maven	1140
Configure and enable the plugin	1140
Declare repositories	1140
Set dependencies	1141
Compile Kotlin-only source code	1141
Compile Kotlin and Java sources	1142
Enable incremental compilation	1143
Configure annotation processing	1143
Create JAR file	1143
Create a self-contained JAR file	1143
Specify compiler options	1144
Use BOM	1145
Generate documentation	1145
Enable OSGi support	1145
Ant	1145
Getting the Ant tasks	1145
Targeting JVM with Kotlin-only source	1146
Targeting JVM with Kotlin-only source and multiple roots	1146

Targeting JVM with Kotlin and Java source	1146
Targeting JavaScript with single source folder	1146
Targeting JavaScript with Prefix, PostFix and sourcemap options	1147
Targeting JavaScript with single source folder and metaInfo option	1147
References	1147
Introduction	1148
Community	1149
Get started with Dokka	1149
Gradle	1150
Apply Dokka	1150
Generate documentation	1151
Build javadoc.jar	1154
Configuration examples	1155
Configuration options	1159
Maven	1169
Apply Dokka	1170
Generate documentation	1170
Build javadoc.jar	1171
Configuration example	1171
Configuration options	1172
CLI	1177
Get started	1177
Generate documentation	1178
Command line options	1179
JSON configuration	1182
HTML	1189
Generate HTML documentation	1189

Configuration	1190
Customization	1192
Markdown	1194
GFM	1194
Jekyll	1195
Javadoc	1196
Generate Javadoc documentation	1197
Dokka plugins	1198
Apply Dokka plugins	1199
Configure Dokka plugins	1200
Notable plugins	1201
Module documentation	1202
File format	1202
Pass files to Dokka	1203
IDEs for Kotlin development	1203
IntelliJ IDEA	1203
Fleet	1203
Android Studio	1203
Eclipse	1204
Compatibility with the Kotlin language versions	1204
Other IDEs support	1204
What's next?	1204
Migrate to Kotlin code style	1204
Kotlin coding conventions and IntelliJ IDEA formatter	1204
Differences between "Kotlin coding conventions" and "IntelliJ IDEA default code style"	1205
Migration to a new code style discussion	1205
Migration to a new code style	1205
Store old code style in project	1206

Kotlin Notebook	1207
Data analytics and visualization	1208
Prototyping	1208
Backend development	1209
Code documentation	1210
Sharing code and outputs	1211
What's next	1212
Run code snippets	1212
IDE: scratches and worksheets	1212
Browser: Kotlin Playground	1214
Command line: ki shell	1216
Kotlin and continuous integration with TeamCity	1218
Gradle, Maven, and Ant	1219
IntelliJ IDEA Build System	1219
Other CI servers	1221
Document Kotlin code: KDoc	1221
KDoc syntax	1221
Inline markup	1222
What's next?	1223
Kotlin and OSGi	1223
Maven	1223
Gradle	1223
FAQ	1224
K2 compiler migration guide	1224
Language feature improvements	1225
How to enable the Kotlin K2 compiler	1230
Try the Kotlin K2 compiler in the Kotlin Playground	1231

Support in IntelliJ IDEA	1231
How to roll back to the previous compiler	1231
Changes	1231
Compatibility with Kotlin releases	1243
Compiler plugins support	1243
Share your feedback on the new K2 compiler	1244
Kotlin command-line compiler	1244
Install the compiler	1244
Create and run an application	1245
Compile a library	1245
Run the REPL	1245
Run scripts	1246
Kotlin compiler options	1246
Compiler options	1246
Common options	1247
Kotlin/JVM compiler options	1248
Kotlin/JS compiler options	1249
Kotlin/Native compiler options	1250
All-open compiler plugin	1252
Gradle	1252
Maven	1253
Spring support	1254
Command-line compiler	1254
No-arg compiler plugin	1255
In your Kotlin file	1255
Gradle	1255
Maven	1256
JPA support	1256

Command-line compiler	1256
SAM-with-receiver compiler plugin	1257
Gradle	1257
Maven	1257
Command-line compiler	1258
kapt compiler plugin	1258
Use in Gradle	1258
Try Kotlin K2 compiler	1259
Annotation processor arguments	1259
Gradle build cache support	1259
Improve the speed of builds that use kapt	1259
Compile avoidance for kapt	1261
Incremental annotation processing	1261
Java compiler options	1261
Non-existent type correction	1262
Use in Maven	1262
Use in IntelliJ build system	1262
Use in CLI	1263
Generate Kotlin sources	1263
AP/Javac options encoding	1263
Keep Java compiler's annotation processors	1264
Lombok compiler plugin	1264
Supported annotations	1264
Gradle	1265
Maven	1265
Using with kapt	1266
Command-line compiler	1266
Power-assert compiler plugin	1266

Apply the plugin	1267
Configure the plugin	1267
Use the plugin	1268
What's next	1271
Kotlin Symbol Processing API	1271
Overview	1271
How KSP looks at source files	1272
SymbolProcessorProvider: the entry point	1272
Resources	1273
Supported libraries	1273
KSP quickstart	1274
Add a processor	1275
Create a processor of your own	1276
Use your own processor in a project	1277
Pass options to processors	1278
Make IDE aware of generated code	1278
Why KSP	1280
KSP makes creating lightweight compiler plugins easier	1280
Comparison to kotlinc compiler plugins	1280
Comparison to reflection	1280
Comparison to kapt	1280
Limitations	1281
KSP examples	1281
Get all member functions	1281
Check whether a class or function is local	1281
Find the actual class or interface declaration that the type alias points to	1281
Collect suppressed names in a file annotation	1281
How KSP models Kotlin code	1281

Type and resolution	1282
Java annotation processing to KSP reference	1283
Program elements	1283
Types	1283
Misc	1284
Details	1285
Incremental processing	1291
Aggregating vs Isolating	1292
Example 1	1293
Example 2	1293
How file dirtiness is determined	1293
Reporting bugs	1293
Multiple round processing	1294
Changes to your processor	1294
Multiple round behavior	1294
Advanced	1295
KSP with Kotlin Multiplatform	1295
Compilation and processing	1296
Avoid the ksp(...) configuration on KSP 1.0.1+	1296
Running KSP from command line	1296
KSP FAQ	1297
Why KSP?	1297
Why is KSP faster than kapt?	1297
Is KSP Kotlin-specific?	1297
How to upgrade KSP?	1297
Can I use a newer KSP implementation with an older Kotlin compiler?	1298
How often do you update KSP?	1298
Besides Kotlin, are there other version requirements to libraries?	1298

What is KSP's future roadmap?	1298
Learning materials overview	1298
Kotlin Koans	1299
Kotlin hands-on	1299
Building Reactive Spring Boot applications with Kotlin coroutines and RSocket	1299
Building web applications with React and Kotlin/JS	1299
Building web applications with Spring Boot and Kotlin	1299
Creating HTTP APIs with Ktor	1300
Creating a WebSocket chat with Ktor	1300
Creating an interactive website with Ktor	1300
Introduction to Kotlin coroutines and channels	1300
Introduction to Kotlin/Native	1300
Kotlin Multiplatform: networking and data storage	1300
Targeting iOS and Android with Kotlin Multiplatform	1300
Kotlin tips	1300
null + null in Kotlin	1300
Deduplicating collection items	1301
The suspend and inline mystery	1301
Unshadowing declarations with their fully qualified name	1302
Return and throw with the Elvis operator	1302
Destructuring declarations	1303
Operator functions with nullable values	1303
Timing code	1304
Improving loops	1304
Strings	1305
Doing more with the Elvis operator	1305
Kotlin collections	1306
What's next?	1306

Kotlin books	1306
Advent of Code puzzles in idiomatic Kotlin	1309
Get ready for Advent of Code	1310
Advent of Code 2022	1310
Advent of Code 2021	1312
Advent of Code 2020	1313
What's next?	1314
Learning Kotlin with JetBrains Academy plugin	1314
Teaching Kotlin with JetBrains Academy plugin	1315
Introduction	1315
What's next	1315
Minimizing mental complexity	1315
Simplicity	1316
Readability	1317
Consistency	1319
Predictability	1320
Debuggability	1322
Testability	1325
What's next	1325
Backward compatibility	1325
Compatibility types	1325
Use the Binary compatibility validator	1326
Specify return types explicitly	1326
Avoid adding arguments to existing API functions	1326
Avoid widening or narrowing return types	1327
Avoid using data classes in your API	1328
Considerations for using the PublishedApi annotation	1329
Evolve APIs pragmatically	1329

Use the RequiresOptIn mechanism	1329
What's next	1329
Informative documentation	1330
Provide comprehensive documentation	1330
Create personas for your users	1330
Document by example whenever possible	1331
Thoroughly document your API	1331
Document lambda parameters	1331
Use explicit links in documentation	1332
Be self-contained where possible	1332
Use simple English	1332
What's next	1332
Participate in the Kotlin Early Access Preview	1332
How the EAP can help you be more productive with Kotlin	1333
Build details	1333
Install the EAP Plugin for IntelliJ IDEA or Android Studio	1333
If you run into any problems	1335
Configure your build for EAP	1335
Configure in Gradle	1335
Configure in Maven	1336
FAQ	1337
What is Kotlin?	1337
What is the current version of Kotlin?	1337
Is Kotlin free?	1337
Is Kotlin an object-oriented language or a functional one?	1337
What advantages does Kotlin give me over the Java programming language?	1337
Is Kotlin compatible with the Java programming language?	1338
What can I use Kotlin for?	1338

Can I use Kotlin for Android development?	1338
Can I use Kotlin for server-side development?	1338
Can I use Kotlin for web development?	1338
Can I use Kotlin for desktop development?	1338
Can I use Kotlin for native development?	1338
What IDEs support Kotlin?	1338
What build tools support Kotlin?	1338
What does Kotlin compile down to?	1338
Which versions of JVM does Kotlin target?	1339
Is Kotlin hard?	1339
What companies are using Kotlin?	1339
Who develops Kotlin?	1339
Where can I learn more about Kotlin?	1339
Are there any books on Kotlin?	1339
Are any online courses available for Kotlin?	1339
Does Kotlin have a community?	1339
Are there Kotlin events?	1340
Is there a Kotlin conference?	1340
Is Kotlin on social media?	1340
Any other online Kotlin resources?	1340
Where can I get an HD Kotlin logo?	1340
Kotlin Evolution	1340
Principles of Pragmatic Evolution	1340
Incompatible changes	1341
Decision making	1341
Language and tooling releases	1342
Libraries	1342
Compiler options	1343
Compatibility tools	1343

Stability of Kotlin components	1343
Stability levels explained	1343
GitHub badges for Kotlin components	1344
Stability of subcomponents	1344
Current stability of Kotlin components	1344
Stability of Kotlin components (pre 1.4)	1346
Compatibility guide for Kotlin 2.0	1347
Basic terms	1348
Language	1348
Tools	1350
Compatibility guide for Kotlin 1.9	1353
Basic terms	1353
Language	1353
Standard library	1360
Tools	1362
Compatibility guide for Kotlin 1.8	1363
Basic terms	1363
Language	1363
Standard library	1372
Tools	1373
Compatibility guide for Kotlin 1.7.20	1374
Basic terms	1375
Language	1375
Compatibility guide for Kotlin 1.7	1375
Basic terms	1376
Language	1376
Standard library	1380
Tools	1382

Compatibility guide for Kotlin 1.6	1385
Basic terms	1385
Language	1385
Standard library	1390
Tools	1393
Compatibility guide for Kotlin 1.5	1396
Basic terms	1396
Language and stdlib	1396
Tools	1403
Compatibility guide for Kotlin 1.4	1404
Basic terms	1404
Language and stdlib	1404
Tools	1418
Compatibility guide for Kotlin 1.3	1419
Basic terms	1419
Incompatible changes	1419
Compatibility modes	1427
Google Summer of Code with Kotlin 2024	1427
Kotlin contributor guidelines for Google Summer of Code (GSoC)	1428
Project ideas	1428
Google Summer of Code with Kotlin 2023	1431
Project ideas	1431
Security	1434
Kotlin documentation as PDF	1435
Contribution	1435
Participate in Early Access Preview	1435
Contribute to the compiler and standard library	1435

Contribute to the Kotlin IDE plugin	1435
Contribute to other Kotlin libraries and tools	1435
Contribute to the documentation	1436
Translate documentation to other languages	1436
Hold events and presentations	1436
KUG guidelines	1436
How to run a KUG?	1436
Support for KUGs from JetBrains	1437
Support from JetBrains for other tech communities	1437
Kotlin Night guidelines	1437
Event guidelines	1437
Event requirements	1437
JetBrains support	1438
Kotlin brand assets	1438
Kotlin Logo	1438
Kotlin mascot	1439
Kotlin User Group brand assets	1440
Kotlin Night brand assets	1443
Testing page	1446
Synchronized tabs	1446
Sections	1446
Codeblocks	1447
Tables	1447
Lists	1449
Text elements	1450
Variables	1450
Embedded elements	1450
Notes	1453

Kotlin Docs

Get started with Kotlin

Kotlin is a modern but already mature programming language designed to make developers happier. It's concise, safe, interoperable with Java and other languages, and provides many ways to reuse code between multiple platforms for productive programming.

To start, why not take our tour of Kotlin? This tour covers the fundamentals of the Kotlin programming language.

[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

Backend

Here is how you can take the first steps in developing Kotlin server-side applications.

1. Create your first backend application:

- To start from scratch, [create a basic JVM application with the IntelliJ IDEA project wizard](#).
- If you prefer more robust examples, choose one of the frameworks below and create a project:

Spring

Ktor

A mature family of frameworks with an established ecosystem that is used by millions of developers worldwide.

- [Create a RESTful web service with Spring Boot](#)
- [Build web applications with Spring Boot and Kotlin](#)
- [Use Spring Boot with Kotlin and RSocket](#)

A lightweight framework for those who value freedom in making architectural decisions.

- [Create HTTP APIs with Ktor](#).
- [Create a WebSocket chat with Ktor](#).
- [Create an interactive website with Ktor](#).
- [Publish server-side Kotlin applications: Ktor on Heroku](#)

2. Use Kotlin and third-party libraries in your application Learn more about [adding library and tool dependencies to your project](#)

- The [Kotlin standard library](#) offers a lot of useful things such as [collections](#) or [coroutines](#).
- Take a look at the following [third-party frameworks, libs and tools for Kotlin](#)

3. Learn more about Kotlin for server-side:

- [How to write your first unit test](#)
- [How to mix Kotlin and Java code in your application](#)

4. Join the Kotlin server-side community:

-  Slack: [get an invite](#) and join the #getting-started, #server, #spring, or #ktor channels.
-  StackOverflow: subscribe to the "kotlin", "spring-kotlin", or "ktor" tags.

5. Follow Kotlin on  Twitter,  Reddit, and  Youtube, and don't miss any important ecosystem updates.

If you've encountered any difficulties or problems, report an issue to our[issue tracker](#).

Cross-platform

Here you'll learn how to develop and improve your cross-platform application using[Kotlin Multiplatform](#).

1. [Set up your environment for cross-platform development](#)
2. Create your first application for iOS and Android:

- To start from scratch, [create a basic cross-platform application with the project wizard](#).
 - If you have an existing Android application and want to make it cross-platform, complete the [Make your Android application work on iOS](#) tutorial.
 - If you prefer real-life examples, clone and play with an existing project, for example the networking and data storage project from the [Create a multiplatform app using Ktor and SQLDelight](#) tutorial or any [sample project](#).
3. Use a wide set of multiplatform libraries to implement the required business logic only once in the shared module. Learn more about [adding dependencies](#).

Library	Details
Ktor	Docs
Serialization	Docs and sample
Coroutines	Docs and sample
DateTime	Docs
SQLDelight	Third-party library. Docs

You can also find a multiplatform library in the [community-driven list](#).

4. Learn more about Kotlin Multiplatform:
- Learn more about [Kotlin Multiplatform](#).
 - Look through [samples projects](#).
 - [Publish a multiplatform library](#).
 - Learn how Kotlin Multiplatform is used at [Netflix](#), [VMware](#), [Yandex](#), and [many other companies](#).

5. Join the Kotlin Multiplatform community:

-  Slack: [get an invite](#) and join the #getting-started and #multiplatform channels.
-  StackOverflow: Subscribe to the "kotlin-multiplatform" tag.

6. Follow Kotlin on  Twitter,  Reddit, and  Youtube, and don't miss any important ecosystem updates.

If you've encountered any difficulties or problems, report an issue to our [issue tracker](#).

Android

- If you want to start using Kotlin for Android development, read [Google's recommendation for getting started with Kotlin on Android](#)
- If you're new to Android and want to learn to create applications with Kotlin, check out [this Udacity course](#).

Follow Kotlin on  Twitter,  Reddit, and  Youtube, and don't miss any important ecosystem updates.

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. Create and edit notebooks seamlessly within the IDE:
 - [Get started with Kotlin Notebook](#)
2. Explore and experiment with your data:
 - [DataFrame](#) – a library for data analysis and manipulation.
 - [Kandy](#) – a plotting tool for data visualization.
3. Get the latest updates about Kotlin for Data Analysis:
 -  Slack: [get an invite](#) and join the #datascience channel.

-  Twitter: follow [KotlinForData](#).
4. Follow Kotlin on  [Twitter](#),  [Reddit](#), and  [Youtube](#), and don't miss any important ecosystem updates.

Is anything missing?

If anything is missing or seems confusing on this page, please [share your feedback](#).

Welcome to our tour of Kotlin!

This tour covers the fundamentals of the Kotlin programming language and can be completed entirely within your browser. There is no installation required.

Each chapter in this tour contains:

- Theory to introduce key concepts of the language with examples.
- Practice with exercises to test your understanding of what you have learned.
- Solutions for your reference.

In this tour you will learn about:

- [Variables](#)
- [Basic types](#)
- [Collections](#)
- [Control flow](#)
- [Functions](#)
- [Classes](#)
- [Null safety](#)

To have the best experience, we recommend that you read through these chapters in order. But if you want, you can choose which chapters you want to read.

Ready to go?

[Start the Kotlin tour](#)

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

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

To assign a value, use the assignment operator =.

For example:

```
fun main() {  
    //sampleStart  
    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  
    //sampleEnd  
}
```

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 that you declare all variables as read-only (val) by default. Declare mutable variables (var) only if necessary.

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() {  
    //sampleStart  
    val customers = 10  
    println("There are $customers customers")  
    // There are 10 customers  
  
    println("There are ${customers + 1} customers")  
    // There are 11 customers  
    //sampleEnd  
}
```

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](#)

Basic types

Every variable and data structure in Kotlin has a data type. Data 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 data type is called type inference. `customers` is assigned an integer value. From this, Kotlin infers that `customers` has numerical data type: `Int`. As a result, the compiler knows that you can perform arithmetic operations with `customers`:

```
fun main() {  
    //sampleStart  
    var customers = 10  
  
    // Some customers leave the queue  
    customers = 8  
  
    customers += 3 // Example of addition: 11  
    customers -= 7 // Example of subtraction: 18  
    customers *= 2 // Example of multiplication: 30  
    customers /= 3 // Example of division: 10  
  
    println(customers) // 10  
    //sampleEnd  
}
```

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

In total, Kotlin has the following basic types:

Category Basic types

Integers Byte, Short, Int, Long

Unsigned integers UByte, UShort, UInt, ULong

Floating-point numbers Float, Double

Booleans Boolean

Category	Basic types
Characters	Char
Strings	String

For more information on basic types and their properties, see [Basic types](#).

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() {
    //sampleStart
    // 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
    //sampleEnd
}
```

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 = 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 val e: Boolean = false val f: Char = '\n' }
```

Next step

[Collections](#)

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() {
//sampleStart
    // 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]
//sampleEnd
}
```

To prevent unwanted modifications, you can obtain read-only views of mutable lists by assigning them 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() {
//sampleStart
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("The first item in the list is: ${readOnlyShapes[0]}")
    // The first item in the list is: triangle
//sampleEnd
}
```

To get the first or last item in a list, use [.first\(\)](#) and [.last\(\)](#) functions respectively:

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

`.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 .

For more information about extension functions, see [Extension functions](#). For the purposes of this tour, you only need to know how to call them.

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

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

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

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

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

```
fun main() {  
    //sampleStart  
    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]  
    //sampleEnd  
}
```

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() {  
    //sampleStart  
    // 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]  
    //sampleEnd  
}
```

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

To prevent unwanted modifications, obtain read-only views of mutable sets by casting them to 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() {
//sampleStart
    val readOnlyFruit = setOf("apple", "banana", "cherry", "cherry")
    println("This set has ${readOnlyFruit.count()} items")
    // This set has 3 items
//sampleEnd
}
```

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

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

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

```
fun main() {
//sampleStart
    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]
//sampleEnd
}
```

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() {
```

```

//sampleStart
    // 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}
//sampleEnd
}

```

To prevent unwanted modifications, obtain read-only views of mutable maps by casting them to 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() {
//sampleStart
    // 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
//sampleEnd
}

```

To get the number of items in a map, use the [.count\(\)](#) function:

```

fun main() {
//sampleStart
    // 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
//sampleEnd
}

```

To add or remove items from a mutable map, use [.put\(\)](#) and [.remove\(\)](#) functions respectively:

```

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

    juiceMenu.remove("orange") // Remove key "orange" from the map
    println(juiceMenu)
    // {apple=100, kiwi=190, coconut=150}
//sampleEnd
}

```

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

```

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

```

To obtain a collection of the keys or values of a map, use the [keys](#) and [values](#) properties respectively:

```

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

```

`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() {
//sampleStart
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
    println("orange" in readOnlyJuiceMenu.keys)
    // true
    println(200 in readOnlyJuiceMenu.values)
    // false
//sampleEnd
}

```

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 spelt 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 spelt as ${number2word[n]}") }
```

Next step

[Control flow](#)

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 as it leads to more robust and safer programs.

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() {
//sampleStart
    val d: Int
    val check = true

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

    println(d)
    // 1
//sampleEnd
}
```

There is no ternary operator condition ? then : else in Kotlin. Instead, if can be used as an expression. When using if as an expression, there are no curly braces {}:

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

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

When

Use when when you have a conditional expression with multiple branches. when can be used either as a statement or as an expression.

Here is an example of using when as a statement:

- Place the conditional expression within parentheses () and the actions to take within curly braces {}.
- Use -> in each branch to separate each condition from each action.

```
fun main() {  
    //sampleStart  
    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  
    //sampleEnd  
}
```

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

Here is an example of using when as an expression. The when syntax is assigned immediately to a variable:

```
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  
    //sampleEnd  
}
```

If when is used as an expression, the else branch is mandatory, unless the compiler can detect that all possible cases are covered by the branch conditions.

The previous example showed that when is useful for matching a variable. when is also useful when you need to check a chain of Boolean expressions:

```
fun main() {  
    //sampleStart  
    val temp = 18  
  
    val description = when {  
        // If temp < 0 is true, sets description to "very cold"  
        temp < 0 -> "very cold"  
        // If temp < 10 is true, sets description to "a bit cold"  
        temp < 10 -> "a bit cold"  
        // If temp < 20 is true, sets description to "warm"  
        temp < 20 -> "warm"  
        // Sets description to "hot" if no previous condition is satisfied  
        else -> "hot"  
    }  
    println(description)  
    // warm  
    //sampleEnd  
}
```

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

Collections can also be iterated over by loops:

```
fun main() {  
    //sampleStart  
    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!  
    //sampleEnd  
}
```

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

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() {  
    //sampleStart  
    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  
    //sampleEnd  
}
```

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](#).

Practice

Exercise 1

Using a when expression, update the following program so that when you input the names of GameBoy buttons, the actions are printed to output.

Button Action

A Yes

B No

X Menu

Y Nothing

Button Action

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" }))
```

Exercise 2

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++
    // 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 3

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

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

```
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.toString() }) } }
```

Exercise 4

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) } }
```

Next step

[Functions](#)

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 =. And due to Kotlin's type inference, 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
}
```

Omitting the return type is only possible when your function has no body ({}). Unless your function's return type is `Unit`.

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 pi via `PI`. For more information about importing packages, see [Packages and imports](#).

```
import kotlin.math.PI

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

Can also be written as a lambda expression:

```
fun main() {  
    println({ string: String -> string.uppercase() }("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:

- `string` is a function parameter.
- `string` has type `String`.
- the function returns the result of the `.uppercase()` function called on `string`.

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:

- assign a lambda to a variable that you can then invoke later

- [pass a lambda expression as a parameter to another function](#)
- [return a lambda expression from a function](#)
- [invoke a lambda expression on its own](#)

Assign to variable

To assign a lambda expression to a variable, use the assignment operator `=`:

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

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() {
    //sampleStart
    val numbers = listOf(1, -2, 3, -4, 5, -6)
    val positives = numbers.filter { x -> x > 0 }
    val negatives = numbers.filter { x -> x < 0 }
    println(positives)
    // [1, 3, 5]
    println(negatives)
    // [-2, -4, -6]
    //sampleEnd
}
```

The `.filter()` function accepts a lambda expression as a predicate:

- `{ x -> x > 0 }` takes each element of the list and returns only those that are positive.
- `{ x -> x < 0 }` takes each element of the list and returns only those that are negative.

If a lambda expression is the only function parameter, you can drop the function parentheses `()`. 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() {
    //sampleStart
    val numbers = listOf(1, -2, 3, -4, 5, -6)
    val doubled = numbers.map { x -> x * 2 }
    val tripled = numbers.map { x -> x * 3 }
    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 = { string -> string.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() {
    //sampleStart
    println({ string: String -> string.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() {
    //sampleStart
    // 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") } }
```

Next step

[Classes](#)

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 most useful predefined member functions of data classes are:

Function	Description
----------	-------------

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)

    //sampleStart
    // 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() {
    //sampleStart
    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() {
    //sampleStart
    val user = User("Alex", 1)
    val secondUser = User("Alex", 1)
    val thirdUser = User("Max", 2)

    // 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

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

Hint

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

Hint

`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())
}

```

Next step

[Null safety](#)

Null safety

In Kotlin, it's possible to have a null value. 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 inferredNotNull = "The compiler assumes non-nullable"

    // Throws a compiler error
    inferredNotNull = 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() {
    var 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() {
    var 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() {
    var 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() {
    var 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 Kotlin tour, check out our tutorials for popular Kotlin applications:

- [Create a backend application](#)
- [Create a cross-platform application for Android and iOS](#)

Kotlin Multiplatform

The Kotlin Multiplatform technology is designed to simplify the development of cross-platform projects. It reduces time spent writing and maintaining the same code for [different platforms](#) while retaining the flexibility and benefits of native programming.

[Kotlin Multiplatform](#)

Kotlin Multiplatform use cases

Android and iOS applications

Sharing code between mobile platforms is a major Kotlin Multiplatform use case. With Kotlin Multiplatform, you can build cross-platform mobile applications that share code between Android and iOS projects to implement networking, data storage and data validation, analytics, computations, and other application logic.

Check out the [Get started with Kotlin Multiplatform](#) and [Create a multiplatform app using Ktor and SQLDelight](#) tutorials, where you will create applications for

Android and iOS that include a module with shared code for both platforms.

Thanks to [Compose Multiplatform](#), a Kotlin-based declarative UI framework developed by JetBrains, you can also share UIs across Android and iOS to create fully cross-platform apps:

Sharing different levels and UI

Check out the [Get started with Compose Multiplatform](#) tutorial to create your own mobile application with UIs shared between both platforms.

Multiplatform libraries

Kotlin Multiplatform is also helpful for library authors. You can create a multiplatform library with common code and its platform-specific implementations for JVM, web, and native platforms. Once published, a multiplatform library can be used as a dependency in other cross-platform projects.

See the [Publish a multiplatform library](#) for more details.

Desktop applications

Compose Multiplatform helps share UIs across desktop platforms like Windows, macOS, and Linux. Many applications, including the [JetBrains Toolbox app](#), have already adopted this approach.

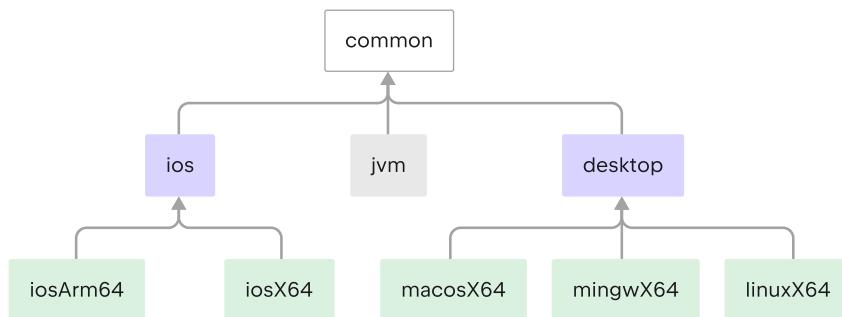
Try this [Compose Multiplatform desktop application template](#) to create your own project with UIs shared among desktop platforms.

Code sharing between platforms

Kotlin Multiplatform allows you to maintain a single codebase of the application logic for [different platforms](#). You also get advantages of native programming, including great performance and full access to platform SDKs.

Kotlin provides the following code sharing mechanisms:

- Share common code among [all platforms](#) used in your project.
- Share code among [some platforms](#) included in your project to reuse much of the code in similar platforms:



Code shared across different platforms

- If you need to access platform-specific APIs from the shared code, use the Kotlin mechanism of [expected and actual declarations](#).

Get started

- Begin with the [Get started with Kotlin Multiplatform](#) if you want to create iOS and Android applications with shared code
- Explore [sharing code principles and examples](#) if you want to create applications or libraries targeting other platforms

New to Kotlin? Take a look at [Getting started with Kotlin](#).

Sample projects

Look through [cross-platform application samples](#) to understand how Kotlin Multiplatform works.

Kotlin for server side

Kotlin is a great fit for developing server-side applications. It allows you to write concise and expressive code while maintaining full compatibility with existing Java-based technology stacks, all with a smooth learning curve:

- Expressiveness: Kotlin's innovative language features, such as its support for [type-safe builders](#) and [delegated properties](#), help build powerful and easy-to-use abstractions.
- Scalability: Kotlin's support for [coroutines](#) helps build server-side applications that scale to massive numbers of clients with modest hardware requirements.
- Interoperability: Kotlin is fully compatible with all Java-based frameworks, so you can use your familiar technology stack while reaping the benefits of a more modern language.
- Migration: Kotlin supports gradual migration of large codebases from Java to Kotlin. You can start writing new code in Kotlin while keeping older parts of your system in Java.
- Tooling: In addition to great IDE support in general, Kotlin offers framework-specific tooling (for example, for Spring) in the plugin for IntelliJ IDEA Ultimate.
- Learning Curve: For a Java developer, getting started with Kotlin is very easy. The automated Java-to-Kotlin converter included in the Kotlin plugin helps with the first steps. [Kotlin Koans](#) can guide you through the key features of the language with a series of interactive exercises.

Frameworks for server-side development with Kotlin

Here are some examples of the server-side frameworks for Kotlin:

- [Spring](#) makes use of Kotlin's language features to offer [more concise APIs](#), starting with version 5.0. The [online project generator](#) allows you to quickly generate a new project in Kotlin.
- [Ktor](#) is a framework built by JetBrains for creating Web applications in Kotlin, making use of coroutines for high scalability and offering an easy-to-use and idiomatic API.
- [Quarkus](#) provides first class support for using Kotlin. The framework is open source and maintained by Red Hat. 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, offers [dedicated support](#) for Kotlin, including [full documentation](#).
- [kotlinx.html](#) is 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](#) is a modern JVM-based full-stack framework for building modular, easily testable microservices and serverless applications. It comes with a lot of useful built-in features.
- [http4k](#) is the functional toolkit with a tiny footprint for Kotlin HTTP applications, written in pure Kotlin. The library is based on the "Your Server as a Function" paper from Twitter and represents modeling both HTTP servers and clients as simple Kotlin functions that can be composed together.
- [Javalin](#) is a very lightweight web framework for Kotlin and Java which supports WebSockets, HTTP2, and async requests.
- The available options for persistence include direct JDBC access, JPA, and using NoSQL databases through their Java drivers. For JPA, the [kotlin-jpa compiler plugin](#) adapts Kotlin-compiled classes to the requirements of the framework.

You can find more frameworks at <https://kotlin.link/>.

Deploying Kotlin server-side applications

Kotlin applications can be deployed into any host that supports Java Web applications, including Amazon Web Services, Google Cloud Platform, and more.

To deploy Kotlin applications on [Heroku](#), you can follow the [official Heroku tutorial](#).

AWS Labs provides a [sample project](#) showing the use of Kotlin for writing [AWS Lambda](#) functions.

Google Cloud Platform offers a series of tutorials for deploying Kotlin applications to GCP, both for [Ktor](#) and [App Engine](#) and [Spring and App engine](#). In addition, there is an [interactive code lab](#) for deploying a Kotlin Spring application.

Products that use Kotlin on the server side

[Corda](#) is an open-source distributed ledger platform that is supported by major banks and built entirely in Kotlin.

[JetBrains Account](#), the system responsible for the entire license sales and validation process at JetBrains, is written in 100% Kotlin and has been running in production since 2015 with no major issues.

Next steps

- For a more in-depth introduction to the language, check out the Kotlin documentation on this site and [Kotlin Koans](#).
- 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.
- http4k provides the [CLI](#) to generate fully formed projects, and a [starter](#) repo to generate an entire CD pipeline using GitHub, Travis, and Heroku with a single bash command.
- Want to migrate from Java to Kotlin? Learn how to perform [typical tasks with strings in Java and Kotlin](#).

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 [Get started with Kotlin Multiplatform for Android and iOS](#).

Kotlin Wasm

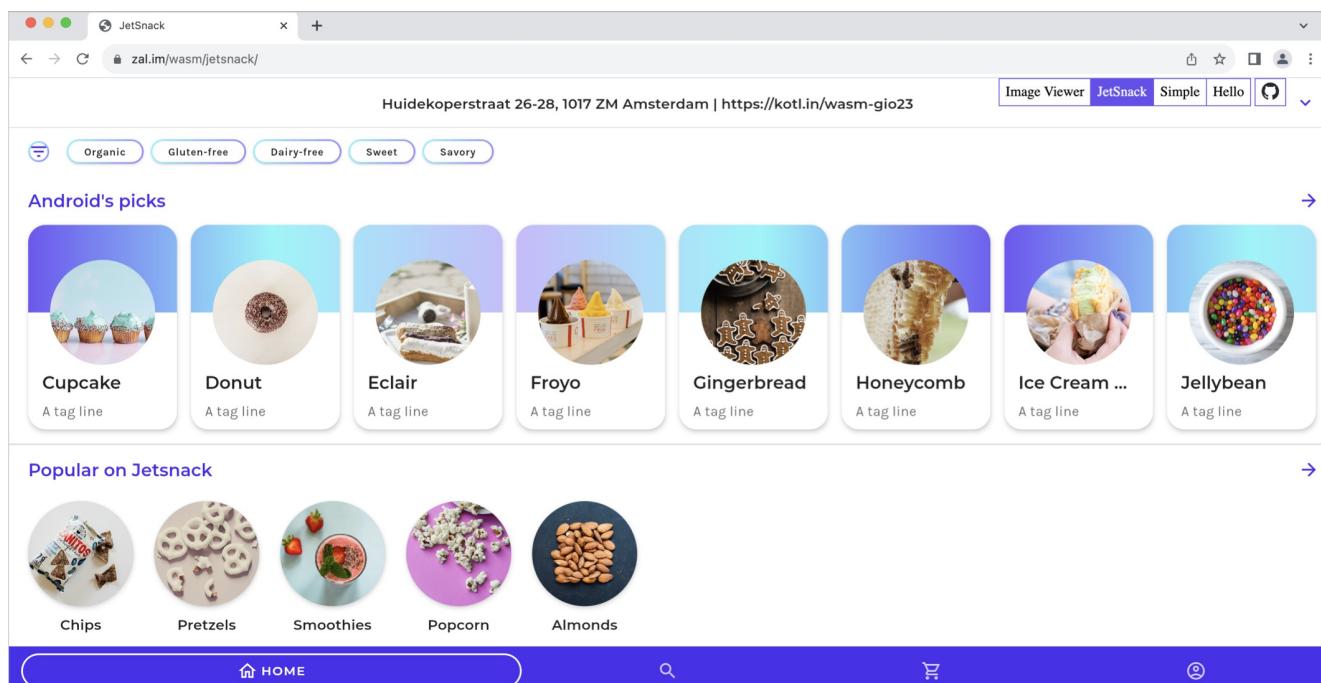
Kotlin Wasm is [Alpha](#). It may be changed at any time. You can use it in scenarios before production. We would appreciate your feedback in [YouTrack](#).

[Join the Kotlin/Wasm community.](#)

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. Specifically for web platforms, Compose Multiplatform uses [Kotlin/Wasm](#) as its compilation target.

[Explore our online demo of an application built with Compose Multiplatform and Kotlin/Wasm](#)



Kotlin/Wasm demo

To run applications built with [Kotlin/Wasm](#) in a browser, you need a browser version that supports the new garbage collection and exception handling proposals. To check the browser support status, see the [WebAssembly roadmap](#).

[WebAssembly \(Wasm\)](#) 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 to run on the web.

Kotlin/Wasm compiles your [Kotlin](#) code into Wasm format. Using [Kotlin/Wasm](#), you can create applications that run on different environments and devices, which support Wasm and meet [Kotlin](#)'s requirements.

Additionally, you can use the most popular [Kotlin](#) libraries in [Kotlin/Wasm](#) out of the box. Like 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?

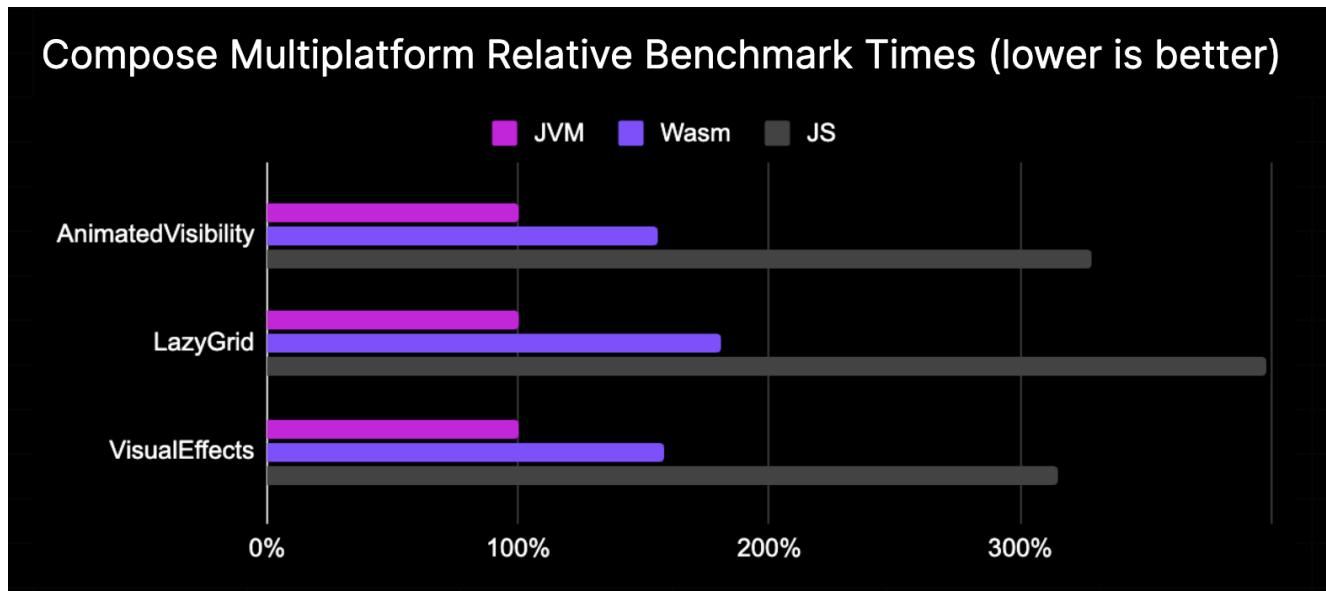
Get started with Kotlin/Wasm →

Get started with [Kotlin/Wasm](#)

Kotlin/Wasm performance

Although [Kotlin/Wasm](#) is still in Alpha, 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.

Kotlin Native

Kotlin/Native is a technology for compiling Kotlin code to native binaries which 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 is ideal for situations when a developer needs to produce a self-contained program that does not require an additional runtime or virtual machine.

Target platforms

Kotlin/Native supports the following platforms:

- macOS
- iOS, tvOS, watchOS
- Linux
- Windows (MinGW)
- Android NDK

To compile Apple targets, macOS, iOS, tvOS, and watchOS, you need [Xcode](#) and its command-line tools installed.

[See the full list of supported targets.](#)

Interoperability

Kotlin/Native supports two-way interoperability with native programming languages for different operating systems. The compiler creates:

- an executable for many [platforms](#)
- a static library or [dynamic](#) library with C headers for C/C++ projects
- an [Apple framework](#) for Swift and Objective-C projects

Kotlin/Native supports interoperability to use existing libraries directly from Kotlin/Native:

- static or dynamic [C libraries](#)
- C, [Swift](#), and [Objective-C](#) frameworks

It is easy to include compiled Kotlin code in existing projects written in C, C++, Swift, Objective-C, and other languages. It is also easy to use existing native code, static or dynamic [C libraries](#), Swift/Objective-C [frameworks](#), graphical engines, and anything else directly from Kotlin/Native.

Kotlin/Native [libraries](#) 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.

Sharing code between platforms

[Kotlin Multiplatform](#) 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.

You can use the [Get started with Kotlin Multiplatform](#) tutorial to create applications and share business logic between iOS and Android. To share UIs among iOS, Android, desktop, and web, try [Compose Multiplatform](#), JetBrains' declarative UI framework based on Kotlin and [Jetpack Compose](#).

How to get started

New to Kotlin? Take a look at [Getting started with Kotlin](#).

Recommended documentation:

- [Get started with Kotlin Multiplatform](#)
- [Interoperability with C](#)
- [Interoperability with Swift/Objective-C](#)

Recommended tutorials:

- [Get started with Kotlin/Native](#)
- [Get started with Kotlin Multiplatform](#)
- [Mapping primitive data types from C](#)
- [Kotlin/Native as a dynamic Library](#)
- [Kotlin/Native as an Apple framework](#)

Kotlin for JavaScript

Kotlin/JS provides the ability to transpile your Kotlin code, the Kotlin standard library, and any compatible dependencies to JavaScript. The current implementation of Kotlin/JS targets [ES5](#).

The recommended way to use Kotlin/JS is via the `kotlin.multiplatform` Gradle plugin. It lets you easily set up and control Kotlin projects targeting JavaScript in one place. This includes essential functionality such as controlling the bundling of your application, adding JavaScript dependencies directly from npm, and more. To get an overview of the available options, check out [Set up a Kotlin/JS project](#).

Kotlin/JS IR compiler

The [Kotlin/JS IR compiler](#) comes with a number of improvements over the old default compiler. For example, it reduces the size of generated executables via dead code elimination and provides smoother interoperability with the JavaScript ecosystem and its tooling.

The old compiler has been deprecated since the Kotlin 1.8.0 release.

By generating TypeScript declaration files (`d.ts`) from Kotlin code, the IR compiler makes it easier to create "hybrid" applications that mix TypeScript and Kotlin code and to leverage code-sharing functionality using Kotlin Multiplatform.

To learn more about the available features in the Kotlin/JS IR compiler and how to try it for your project, visit the [Kotlin/JS IR compiler documentation page](#) and the [migration guide](#).

Kotlin/JS frameworks

Modern web development benefits significantly from frameworks that simplify building web applications. Here are a few examples of popular web frameworks for Kotlin/JS written by different authors:

KVision

KVision is an object-oriented web framework that makes it possible to write applications in Kotlin/JS with ready-to-use components that can be used as building blocks for your application's user interface. You can use both reactive and imperative programming models to build your frontend, use connectors for Ktor, Spring Boot, and other frameworks to integrate it with your server-side applications, and share code using [Kotlin Multiplatform](#).

[Visit KVision site](#) for documentation, tutorials, and examples.

For updates and discussions about the framework, join the `#kvision` and `#javascript` channels in the [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 it makes use of Kotlin's coroutines and flows to express components and their data bindings. It provides state management, validation, routing, and more out of the box, and integrates with Kotlin Multiplatform projects.

[Visit fritz2 site](#) for documentation, tutorials, and examples.

For updates and discussions about the framework, join the [#fritz2](#) and [#javascript](#) channels in the [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. By using this approach, Doodle gives you precise control over the rendering of arbitrary UI elements, vector shapes, gradients, and custom visualizations.

[Visit Doodle site](#) for documentation, tutorials, and examples.

For updates and discussions about the framework, join the [#doodle](#) and [#javascript](#) channels in the [Kotlin Slack](#).

Join the Kotlin/JS community

You can join the [#javascript](#) channel in the official [Kotlin Slack](#) to chat with the community and the team.

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 notebooks](#), [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 file formats, including CSV, JSON, and TXT.
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.

Would you like to try our Kotlin tools for data analysis?

Get started with Kotlin Notebook

[Get started with Kotlin Notebook](#)

Our Kotlin data analysis tools let you smoothly handle your data from start to finish. Effortlessly retrieve your data with simple drag-and-drop functionality in our Kotlin Notebook. Clean, transform, and visualize it with just a few lines of code. Additionally, export your output charts in a matter of clicks.

The screenshot shows the Kotlin Notebook interface within an IDE. The left sidebar displays the project structure of 'kotlin-notebook' with files like .gradle, .idea, gradle, src/main/kotlin/Main.kt, resources/movies.csv, movies-dataset.c, test, .gitignore, build.gradle.kts, gradle.properties, gradlew, gradlew.bat, kotlin-notebook.ipynb, kotlin-notebook-gif.ipynb, kotlin-notebook-test.ipynb, and settings.gradle.kts. The main area shows a code cell for 'Import libraries' with two commands: %use kandy and %use dataframe. Below it, a section titled 'Read data from a CSV file' contains a note 'Drag and drop' and a code block for 'Clean and transform data' with instructions to filter movies from 2015 onwards, split genres, and remove movie ID. A code cell shows the implementation of this logic using Kotlin's DataFrames. Further sections include 'Group and count movies by genre' with a corresponding code cell.

Kotlin Notebook

Notebooks

Notebooks are interactive editors that integrate code, graphics, and text in a single environment. When using a notebook, you can run code cells and immediately see the output.

Kotlin offers different notebook solutions, such as [Kotlin Notebook](#), [Datalore](#), and [Kotlin-Jupyter Notebook](#), providing convenient features for data retrieving, transformation, exploration, modeling, and more. These Kotlin notebook solutions are based on our [Kotlin Kernel](#).

You can seamlessly share your code among Kotlin Notebook, Datalore, and Kotlin-Jupyter Notebook. Create a project in one of our Kotlin notebooks and continue working in another notebook without compatibility issues.

Benefit from the features of our powerful Kotlin notebooks and the perks of coding with Kotlin. Kotlin integrates with these notebooks to help you manage data and share your findings with colleagues while building up your data science and machine learning skills.

Discover the features of our different Kotlin notebook solutions and choose the one that best aligns with your project requirements.

The screenshot shows the IntelliJ IDEA interface with the following details:

- Title Bar:** Shows "dataframe" and "master".
- Toolbar:** Includes icons for file operations, search, and navigation.
- Code Editor:** An "In" cell (In 4) contains Java code for plotting a dataset. A code completion tooltip is open over the variable "x(t)", listing methods like `String.toString()`, `Object.toString()`, and `timeMs` (highlighted in blue).
- Output Area:** An "Out" cell (Out 4) displays a scatter plot with points at coordinates (time, relativeHumidity). The legend indicates "flowOn" with green dots for "true" and orange dots for "false".
- Sidebar:** Shows icons for navigation, search, and other notebook-related functions.
- Status Bar:** Shows "dataframe > kandy.ipynb".

Kotlin Notebook

Kotlin Notebook

The [Kotlin Notebook](#) is a plugin for IntelliJ IDEA that allows you to create notebooks in Kotlin. It provides our IDE experience with all common IDE features, offering real-time code insights and project integration.

Kotlin notebooks in Datalore

With [Datalore](#), you can use Kotlin in the browser straight out of the box without additional installation. You can also share your notebooks and run them remotely, collaborate with other Kotlin notebooks in real-time, receive smart coding assistance as you write code, and export results through interactive or static reports.

Jupyter Notebook with Kotlin Kernel

[Jupyter Notebook](#) is an open-source web application that allows you to create and share documents containing code, visualizations, and Markdown text. [Kotlin-Jupyter](#) is an open-source project that brings Kotlin support to Jupyter Notebook to harness Kotlin's power within the Jupyter environment.

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.

A screenshot of a Jupyter Notebook cell titled "weather.ipynb". The cell contains the following code:

```
[16] 1 val lastYear = amsWeather.filter { datetime.year == 2023 }
2 lastYear.head(3)
```

The code is executed at 2024.04.04 14:24:58 in 262ms. The output shows a table with 3 rows and 24 columns, representing weather data for Amsterdam in January 2023. The columns include name, datetime, temp, feelslike, dew, humidity, precip, and others. The first three rows are:

name	datetime	temp	feelslike	dew	humidity	precip
Amsterdam	2023-01-01T00:00	14	14	85	6919	0
Amsterdam	2023-01-01T01:00	14	14	87	7017	14

Below the code cell, there are two bullet points:

- Add a column for the date and move it to the left
- Remove the name column

Another code cell [15] shows the modified code:

```
[15] 1 val withDate = lastYear|
2     .add("date") { datetime.date }
3     .moveToLeft { takeLast(1) }
4     .remove { name }
5 withDate.head(3)
```

The output shows the same table, but the "name" column has been removed and a new "date" column has been added to the left of the "datetime" column. The first three rows are:

date	datetime	temp	feelslike	dew	humidity	precip
2023-01-01	2023-01-01T00:00	14	14	85	6919	0
2023-01-01	2023-01-01T01:00	14	14	87	7017	14

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.

Kandy has seamless integration with Kotlin Notebook, Datalore, and Kotlin-Jupyter Notebook. You can also easily combine the Kandy and Kotlin DataFrame libraries to complete different data-related tasks.



Kandy

What's next

- [Get started with Kotlin Notebook.](#)
- [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.](#)

Kotlin for competitive programming

This tutorial is designed both for competitive programmers that did not use Kotlin before and for Kotlin developers that did not participate in any competitive programming events before. It assumes the corresponding programming skills.

[Competitive programming](#) is a mind sport where contestants write programs to solve precisely specified algorithmic problems within strict constraints. Problems can range from simple ones that can be solved by any software developer and require little code to get a correct solution, to complex ones that require knowledge of special algorithms, data structures, and a lot of practice. While not being specifically designed for competitive programming, Kotlin incidentally fits well in this domain, reducing the typical amount of boilerplate that a programmer needs to write and read while working with the code almost to the level offered by dynamically-typed scripting languages, while having tooling and performance of a statically-typed language.

See [Get started with Kotlin/JVM](#) on how to set up development environment for Kotlin. In competitive programming, a single project is usually created and each

problem's solution is written in a single source file.

Simple example: Reachable Numbers problem

Let's take a look at a concrete example.

[Codeforces](#) Round 555 was held on April 26th for 3rd Division, which means it had problems fit for any developer to try. You can use [this link](#) to read the problems. The simplest problem in the set is the [Problem A: Reachable Numbers](#). It asks to implement a straightforward algorithm described in the problem statement.

We'd start solving it by creating a Kotlin source file with an arbitrary name. A.kt will do well. First, you need to implement a function specified in the problem statement as:

Let's denote a function $f(x)$ in such a way: we add 1 to x , then, while there is at least one trailing zero in the resulting number, we remove that zero.

Kotlin is a pragmatic and unopinionated language, supporting both imperative and function programming styles without pushing the developer towards either one. You can implement the function f in functional style, using such Kotlin features as [tail recursion](#):

```
tailrec fun removeZeroes(x: Int): Int =  
    if (x % 10 == 0) removeZeroes(x / 10) else x  
  
fun f(x: Int) = removeZeroes(x + 1)
```

Alternatively, you can write an imperative implementation of the function f using the traditional [while loop](#) and mutable variables that are denoted in Kotlin with [var](#):

```
fun f(x: Int): Int {  
    var cur = x + 1  
    while (cur % 10 == 0) cur /= 10  
    return cur  
}
```

Types in Kotlin are optional in many places due to pervasive use of type-inference, but every declaration still has a well-defined static type that is known at compilation.

Now, all is left is to write the main function that reads the input and implements the rest of the algorithm that the problem statement asks for — to compute the number of different integers that are produced while repeatedly applying function f to the initial number n that is given in the standard input.

By default, Kotlin runs on JVM and gives direct access to a rich and efficient collections library with general-purpose collections and data-structures like dynamically-sized arrays (`ArrayList`), hash-based maps and sets (`HashMap`/`HashSet`), tree-based ordered maps and sets (`TreeMap`/`TreeSet`). Using a hash-set of integers to track values that were already reached while applying function f , the straightforward imperative version of a solution to the problem can be written as shown below:

Kotlin 1.6.0 and later

```
fun main() {  
    var n = readln().toInt() // read integer from the input  
    val reached = HashSet<Int>() // a mutable hash set  
    while (reached.add(n)) n = f(n) // iterate function f  
    println(reached.size) // print answer to the output  
}
```

There is no need to handle the case of misformatted input in competitive programming. An input format is always precisely specified in competitive programming, and the actual input cannot deviate from the input specification in the problem statement. That's why you can use Kotlin's `readln()` function. It asserts that the input string is present and throws an exception otherwise. Likewise, the `String.toInt()` function throws an exception if the input string is not an integer.

Earlier versions

```
fun main() {  
    var n = readLine()!!.toInt() // read integer from the input  
    val reached = HashSet<Int>() // a mutable hash set  
    while (reached.add(n)) n = f(n) // iterate function f  
    println(reached.size) // print answer to the output  
}
```

Note the use of Kotlin's [null-assertion operator](#) `!!` after the `readLine()` function call. Kotlin's `readLine()` function is defined to return a [nullable type](#) `String?` and returns null on the end of the input, which explicitly forces the developer to handle the case of missing input.

There is no need to handle the case of misformatted input in competitive programming. In competitive programming, an input format is always precisely specified and the actual input cannot deviate from the input specification in the problem statement. That's what the null-assertion operator `!!` essentially does — it asserts that the input string is present and throws an exception otherwise. Likewise, the [String.toInt\(\)](#).

All online competitive programming events allow the use of pre-written code, so you can define your own library of utility functions that are geared towards competitive programming to make your actual solution code somewhat easier to read and write. You would then use this code as a template for your solutions. For example, you can define the following helper functions for reading inputs in competitive programming:

Kotlin 1.6.0 and later

```
private fun readStr() = readLn() // string line
private fun readInt() = readStr().toInt() // single int
// similar for other types you'd use in your solutions
```

Earlier versions

```
private fun readStr() = readLine()!! // string line
private fun readInt() = readStr().toInt() // single int
// similar for other types you'd use in your solutions
```

Note the use of private [visibility modifier](#) here. While the concept of visibility modifier is not relevant for competitive programming at all, it allows you to place multiple solution files based on the same template without getting an error for conflicting public declarations in the same package.

Functional operators example: Long Number problem

For more complicated problems, Kotlin's extensive library of functional operations on collections comes in handy to minimize the boilerplate and turn the code into a linear top-to-bottom and left-to-right fluent data transformation pipeline. For example, the [Problem B: Long Number](#) problem takes a simple greedy algorithm to implement and it can be written using this style without a single mutable variable:

Kotlin 1.6.0 and later

```
fun main() {
    // read input
    val n = readLn().toInt()
    val s = readLn()
    val fl = readLn().split(" ").map { it.toInt() }
    // define local function f
    fun f(c: Char) = '0' + fl[c - '1']
    // greedily find first and last indices
    val i = s.indexOfFirst { c -> f(c) > c }
        .takeIf { it >= 0 } ?: s.length
    val j = s.withIndex().indexOfFirst { (j, c) -> j > i && f(c) < c }
        .takeIf { it >= 0 } ?: s.length
    // compose and write the answer
    val ans =
        s.substring(0, i) +
        s.substring(i, j).map { c -> f(c) }.joinToString("") +
        s.substring(j)
    println(ans)
}
```

Earlier versions

```
fun main() {
    // read input
    val n = readLine()!!.toInt()
    val s = readLine()!!
    val fl = readLine()!!.split(" ").map { it.toInt() }
    // define local function f
    fun f(c: Char) = '0' + fl[c - '1']
    // greedily find first and last indices
    val i = s.indexOfFirst { c -> f(c) > c }
        .takeIf { it >= 0 } ?: s.length
    val j = s.withIndex().indexOfFirst { (j, c) -> j > i && f(c) < c }
        .takeIf { it >= 0 } ?: s.length
    // compose and write the answer
}
```

```

    val ans =
        s.substring(0, i) +
        s.substring(i, j).map { c -> f(c) }.joinToString("") +
        s.substring(j)
    println(ans)
}

```

In this dense code, in addition to collection transformations, you can see such handy Kotlin features as local functions and the [elvis operator](#) ?: that allow to express [idioms](#) like "take the value if it is positive or else use length" with a concise and readable expressions like .takeIf { it >= 0 } ?: s.length, yet it is perfectly fine with Kotlin to create additional mutable variables and express the same code in imperative style, too.

To make reading the input in competitive programming tasks like this more concise, you can have the following list of helper input-reading functions:

Kotlin 1.6.0 and later

```

private fun readStr() = readln() // string line
private fun readInt() = readStr().toInt() // single int
private fun readStrings() = readStr().split(" ") // list of strings
private fun readInts() = readStrings().map { it.toInt() } // list of ints

```

Earlier versions

```

private fun readStr() = readLine()!! // string line
private fun readInt() = readStr().toInt() // single int
private fun readStrings() = readStr().split(" ") // list of strings
private fun readInts() = readStrings().map { it.toInt() } // list of ints

```

With these helpers, the part of code for reading input becomes simpler, closely following the input specification in the problem statement line by line:

```

// read input
val n = readInt()
val s = readStr()
val fl = readInts()

```

Note that in competitive programming it is customary to give variables shorter names than it is typical in industrial programming practice, since the code is to be written just once and not supported thereafter. However, these names are usually still mnemonic — a for arrays, i, j, and others for indices, r, and c for row and column numbers in tables, x and y for coordinates, and so on. It is easier to keep the same names for input data as it is given in the problem statement. However, more complex problems require more code which leads to using longer self-explanatory variable and function names.

More tips and tricks

Competitive programming problems often have input like this:

The first line of the input contains two integers n and k

In Kotlin this line can be concisely parsed with the following statement using [destructuring declaration](#) from a list of integers:

```
val (n, k) = readInts()
```

It might be tempting to use JVM's java.util.Scanner class to parse less structured input formats. Kotlin is designed to interoperate well with JVM libraries, so that their use feels quite natural in Kotlin. However, beware that java.util.Scanner is extremely slow. So slow, in fact, that parsing 105 or more integers with it might not fit into a typical 2 second time-limit, which a simple Kotlin's split(" ").map { it.toInt() } would handle.

Writing output in Kotlin is usually straightforward with `println(...)` calls and using Kotlin's [string templates](#). However, care must be taken when output contains on order of 105 lines or more. Issuing so many `println` calls is too slow, since the output in Kotlin is automatically flushed after each line. A faster way to write many lines from an array or a list is using `joinToString()` function with "\n" as the separator, like this:

```
println(a.joinToString("\n")) // each element of array/list on a separate line
```

Learning Kotlin

Kotlin is easy to learn, especially for those who already know Java. A short introduction to the basic syntax of Kotlin for software developers can be found directly in the reference section of the website starting from [basic syntax](#).

IDEA has built-in [Java-to-Kotlin converter](#). It can be used by people familiar with Java to learn the corresponding Kotlin syntactic constructions, but it is not perfect, and it is still worth familiarizing yourself with Kotlin and learning the [Kotlin idioms](#).

A great resource to study Kotlin syntax and API of the Kotlin standard library are [Kotlin Koans](#).

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 kotlinc-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](#)

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 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 and 80,000 projects were involved in the stabilization process, trying the new K2 compiler in their 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.

We explored the performance of the K2 compiler in different projects in a [blog post](#). 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.

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:

```

//sampleStart
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.
    }
}

//sampleEnd
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
    if (unknownObject is Tau) {

        // Uses the overloaded inc() operator from interface Rho,
        // which smart casts the type of unknownObject 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 thinks unknownObject has type
        // Tau, so calling the sigma() function is not allowed.

        // In Kotlin 2.0.0, the compiler knows unknownObject has type
        // Sigma, so calling the tau() function is not allowed.
        unknownObject.tau()
        // Unresolved reference 'tau'

        // In Kotlin 1.9.20, the compiler mistakenly thinks that
        // unknownObject has type Tau, the tau() function can be
        // called successfully.
    }
}

```

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 Platform code

```
fun foo(x: Any) = println("common foo")  
  
fun exampleFunction() {  
    foo(42)  
}  
  
// 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 Platform 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.  
}  
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 the K2 Kotlin mode.

In your IDE, go to **Settings | Languages & Frameworks | Kotlin** and select the **Enable the K2-based Kotlin plugin** option. The IDE will analyze your code with its K2 Kotlin mode.

The K2 Kotlin mode is in Alpha and is available starting from 2024.1. The performance and stability of code highlighting and code completion have been significantly improved, but not all IDE features are supported yet.

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 the K2 Kotlin mode in [our blog](#).
- We are actively collecting feedback about K2 Kotlin mode. 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

This version brings the following changes:

- [Generation of lambda functions using invokedynamic](#)
- [The kotlinc-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 kotlinc-metadata-jvm library is Stable

In Kotlin 2.0.0, the kotlinc-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 kotlinc-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

The Kotlin/Native compiler used to resolve 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 under the option](#)
- [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 the binaries 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 are introducing support for catching JavaScript exceptions, no additional information is provided when a JavaScript exception occurs, like a call stack. However, [we are working on these implementations](#).

New exception handling proposal is now supported under the 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. It is turned off by default.

`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 per 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
@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
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()`

Since 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 kotlinc.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 kotlinc.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(method = "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' }))
```

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"
}
```

```
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 implement the following changes:

- The `webpack` task now targets a separate folder.
- The `distributeResources` task is removed completely.
- 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 that 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](#)
- [Bumping minimum supported versions](#)
- [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 to try latest language version](#)
- [New JSON output format for build reports](#)

- [kapt configurations inherit annotation processors from super configurations](#)
- [Kotlin Gradle plugin no longer uses deprecated Gradle conventions](#)

New Gradle DSL for compiler options in multiplatform projects

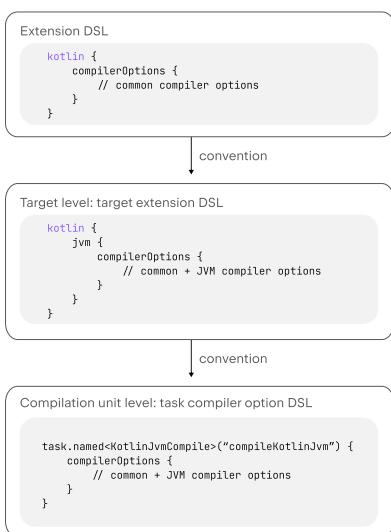
This feature is [Experimental](#). It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in [YouTrack](#).

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

With this change, you may need to add the `.kotlin` directory 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
<code>kotlin.project.persistent.dir</code>	Configures the location where your project-level data is stored. Default: <code><project-root-directory>/.kotlin</code>
<code>kotlin.project.persistent.dir.gradle.disableWrite</code>	A boolean value that controls whether writing Kotlin data to the <code>.gradle</code> directory is disabled. Default: <code>false</code>

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 the version 1.9.20-Beta, the Kotlin/Native distribution is published to [Maven Central](#) along with CDN.

This allowed us to change how Kotlin looks for and downloads the necessary artifacts. Instead of the CDN, it now uses by default 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.nativedistribution.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 KotlinCompiaction 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("-Xerror-tolerance-policy=SYNTAX")  
            }  
        }  
    }  
}
```

- 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 to try 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_numbers>.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 super configurations

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 super configuration. Kapt sees that the commonAnnotationProcessors Gradle configuration has a dependency on the Dagger annotation processor and successfully includes it 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 `syntheticValues()` 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 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 ")
        // Kotlin is awesome!
    }
}

```

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.

What's new in Kotlin 2.0.0-RC3

Released: May 10, 2024

This document doesn't cover all of the features of the Early Access Preview (EAP) release, but it highlights some major improvements.

See the full list of changes in the [GitHub changelog](#).

The Kotlin 2.0.0-RC3 release is out! It mostly covers the stabilization of the [new Kotlin K2 compiler](#), which reached its Beta status for all targets since 1.9.20. In addition, there are new features in [Kotlin/Wasm](#) and [Kotlin/JS](#), as well as [improvements for the Gradle build tool](#).

IDE support

The Kotlin plugins that support 2.0.0-RC3 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 2.0.0-RC3 in your build scripts.

For details about IntelliJ IDEA's support for the Kotlin K2 compiler, see [Support in IntelliJ IDEA](#).

Kotlin K2 compiler

The JetBrains team is still working on the stabilization of the new Kotlin K2 compiler. The new Kotlin K2 compiler will bring major performance improvements, speed up new language feature development, unify all platforms that Kotlin supports, and provide a better architecture for multiplatform projects.

The K2 compiler is in [Beta](#) for all target platforms: JVM, Native, Wasm, and JS. The JetBrains team has ensured the quality of the new compiler by successfully compiling dozens of user and internal projects. A large number of users are also involved in the stabilization process, trying the new K2 compiler in their projects and reporting any problems they find.

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 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-RC3, 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 was smart cast, and information about this variable was shared further within the scope of the if block. However, if you declared the variable outside the if condition, no information about the variable was 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-RC3, 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 is accessible in the condition statement and its 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 makes your code easier to read, and easily 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-RC3, the compiler can access  
            // information about isCat, so it knows that  
            // animal was smart cast to type Cat.  
            // Therefore, the purr() function is successfully 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-RC3, 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-RC3, 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-RC3, 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. So, 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 if 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-RC3, 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, it was a bug that class properties with a function type weren't smart cast. We fixed this behavior in the K2 compiler in Kotlin 2.0.0-RC3. For example:

```

class Holder(val provider: () -> Unit?) {
    fun process() {
        // In Kotlin 2.0.0-RC3, 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-RC3, 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:

```

//sampleStart
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-RC3, 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.
    }
}
//sampleEnd
fun main() {
    testString()
}

```

Increment and decrement operators

Prior to Kotlin 2.0.0-RC3, 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-RC3, this is 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
    if (unknownObject is Tau) {

        // Use the overloaded inc() operator from interface Rho,
        // which smart casts the type of unknownObject to Sigma.
        ++unknownObject

        // In Kotlin 2.0.0-RC3, the compiler knows unknownObject has type
        // Sigma, so the sigma() function is called successfully.
        unknownObject.sigma()

        // In Kotlin 1.9.20, the compiler thinks unknownObject has type
        // Tau, so calling the sigma() function throws an error.

        // In Kotlin 2.0.0-RC3, the compiler knows unknownObject has type
        // Sigma, so calling the tau() function throws an error.
        unknownObject.tau()

        // Unresolved reference 'tau'

        // In Kotlin 1.9.20, the compiler mistakenly thinks that
        // unknownObject has type Tau, so the tau() function is
        // called successfully.
    }
}

```

Kotlin Multiplatform improvements

In Kotlin 2.0.0-RC3, 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, due to its design, the Kotlin compiler couldn't keep common and platform source sets separate at compile time. This means that common code could access platform code, which resulted in different behavior between platforms. In addition, some compiler settings and dependencies from common code were leaked into platform code.

In Kotlin 2.0.0-RC3, we redesigned the compilation scheme as part of the new Kotlin K2 compiler so that there is a strict separation between common and platform source sets. The most noticeable change is 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

Platform code

<pre> fun foo(x: Any) = println("common foo") fun exampleFunction() { foo(42) } </pre>	<pre> // JVM fun foo(x: Int) = println("platform foo") // JavaScript // There is no foo() function overload // on the JavaScript platform </pre>
--	---

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 common code results in the foo() function from platform code being called: platform foo
- On the JavaScript platform, calling the foo() function in common code results in the foo() function from common code being called: common foo, since there is none available in platform code.

In Kotlin 2.0.0-RC3, common code doesn't have access to platform code, so both platforms successfully resolve the foo() function to the foo() function in 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 example, if you used expected and actual classes:

Common code

Platform code

```
expect class Identity {
    fun confirmIdentity(): String
}

fun common() {
    // Before 2.0.0-RC3,
    // it triggers an IDE-only error
    Identity().confirmIdentity()
    // RESOLUTION_TO_CLASSIFIER : Expected class
    // Identity has no default constructor.
}
```

```
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, only an IDE error was reported, 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 are 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 will notice this difference mainly when you use overloads from a multiplatform library in your common code.

For example, if you have this 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 common code is 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 has 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-RC3, if you used [expected and actual declarations](#) in your Kotlin Multiplatform project, they had to have the same [visibility level](#). Kotlin 2.0.0-RC3 supports different visibility levels only if the actual declaration is less strict than the expected declaration. For example:

```
expect internal class Attribute // Visibility is internal
actual class Attribute        // Visibility is public by default,
                             // which is less strict
```

If you are using a [type alias](#) in your actual declaration, the visibility of the type must be less strict. Any visibility modifiers for actual typealias are ignored. For example:

```
expect internal class Attribute           // Visibility is internal
internal actual typealias Attribute = Expanded // The internal visibility
                                                // modifier is ignored
class Expanded                           // Visibility is public by default,
                                         // which is less strict
```

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](#)

In addition, the Kotlin K2 compiler supports:

- the [Jetpack Compose](#) 1.5.0 compiler plugin and later versions.
- 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.

Tasks error in Gradle configuration cache

Since Kotlin 2.0.0-Beta4, 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](#).

How to enable the Kotlin K2 compiler

Starting with Kotlin 2.0.0-Beta1, 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-RC3 release. [Check it out!](#)

Support in IntelliJ IDEA

The K2 Kotlin mode is in Alpha. The performance and stability of code highlighting and code completion have been significantly improved, but not all IDE features are supported yet.

Starting from 2024.1, IntelliJ IDEA can use the new K2 compiler to analyze your code with its K2 Kotlin mode. To try it out, go to Settings | Languages & Frameworks | Kotlin and select the Enable the K2-based Kotlin plugin option.

Learn more about the K2 Kotlin mode in [our blog](#).

We are actively collecting feedback about K2 Kotlin mode. 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!

- Provide your feedback directly to K2 developers on Kotlin Slack – [get an invite](#) and join the `#k2-early-adopters` channel.
- 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

This version brings the following changes:

- [Generate lambda functions using invokedynamic](#)
- [The kotlinx-metadata-jvm library is now Stable](#)

Generate lambda functions using invokedynamic

Kotlin 2.0.0-RC3 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](#), the option for invokedynamic generation was available by using the `-Xlambdas=indy` compiler flag. In Kotlin 2.0.0-RC3, invokedynamic has become the default method for lambda generation. This method produces lighter binaries and aligns Kotlin with JVM optimizations, ensuring that 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.

To retain the legacy behavior of generating lambda functions, you can either:

- Annotate specific lambdas with `@JvmSerializableLambda`.
- Use the compiler argument `-Xlambdas=class` to generate all lambdas in a module using the legacy method.

The kotlinx-metadata-jvm library is now Stable

In 2.0.0-RC3, the `kotlinx-metadata-jvm` library became [Stable](#). Since the library's package and coordinates have changed to `kotlin`, you can now find it under the

name kotlin-metadata-jvm (without the "x").

Before, the kotlinx-metadata-jvm library had its own publishing scheme and version. Now, we 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:

- [Resolving conflicts with Objective-C methods](#)
- [Changed log level for compiler arguments in Kotlin/Native](#)

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-RC3 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. It allows you to use it only 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 tasks, such as `compile`, `link`, and `cinterop`, changed from `info` to `debug`.

With `debug` as its default value, the log level is consistent with other compile tasks and provides detailed debugging information, including all compiler arguments.

Kotlin/Wasm

Kotlin 2.0.0-RC3 improves performance and interoperability with JavaScript:

- [Unsigned primitive types in functions with @JsExport](#)
- [Binaryen available by default in production builds](#)
- [Generation of TypeScript declaration files in Kotlin/Wasm](#)
- [Support for named export](#)
- [withWasm\(\) function is split into JS and WASI variants](#)
- [New functionality to catch JavaScript exceptions](#)
- [The new exception handling proposal is now supported](#)

Unsigned primitive types in functions with @JsExport

Kotlin 2.0.0-RC3 further improves the interoperability between Kotlin/Wasm and JavaScript. Now you can use `unsigned primitive types` inside external declarations and functions with the `@JsExport` annotation that makes Kotlin/Wasm functions available in JavaScript code.

It helps to ease the previous limitation when `generic class types` couldn't be used directly inside exported declarations. Now you can export functions with unsigned primitives as a return or parameter type and consume external declarations that return unsigned primitives.

For more information on Kotlin/Wasm interoperability with JavaScript, see the [documentation](#).

Binaryen available by default in production builds

Kotlin/Wasm now applies WebAssembly's [Binaryen](#) library during production compilation to all the projects as opposed to the previous manual approach.

Binaryen is a great tool for code optimization. We believe it will improve your project performance and enhance your development experience.

This change only affects production compilation. The development compilation process stays the same.

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-RC3, 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` section, add the `generateTypeScriptDefinitions()` function:

```
kotlin {  
    wasmJs {  
        binaries.executable()  
        browser {  
        }  
        generateTypeScriptDefinitions()  
    }  
}
```

Support for named export

Previously, all exported declarations from Kotlin/Wasm were imported into JavaScript using default export. Now, you can import each Kotlin declaration marked with `@JsExport` by name:

```
// Kotlin:  
@JsExport  
fun add(a: Int, b: Int) = a + b  
  
// JS:  
import { add } from "./index.mjs"
```

Named exports make it easier to share code between Kotlin and JavaScript modules. They help improve readability and manage dependencies between modules.

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.

New functionality to catch 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 2.0.0-RC3, 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 are introducing support for catching JavaScript exceptions, no additional information is provided when a JavaScript exception occurs, like a call stack. However, we are working on these implementations.

The new exception handling proposal is now supported

In this release, we introduce support for the new version of WebAssembly's [exception handling proposal](#) within Kotlin/Wasm. With this update, Kotlin/Wasm gains enhanced capabilities for managing exceptions.

The new exception handling proposal is activated using the `-Xwasm-use-new-exception-proposal` compiler option. It is turned off by default.

Additionally, we have ensured compatibility between the new compiler option and existing configurations, such as `-Xwasm-use-traps-instead-of-exceptions`.

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](#)

New compilation target

In Kotlin 2.0.0-RC3, 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 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](#).

We believe that using generators instead of state machines will improve both the final bundle size and your debugging experience.

[Learn more about ES2015 \(ECMAScript 2015, ES6\) in the official documentation](#).

Passing arguments to the main function

Starting with Kotlin 2.0.0-RC3, 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 expression 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 JS 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 passing 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 per 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/JavaScript 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  
@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  
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 the next Kotlin release.

Support for `createInstance()`

Since 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-RC3, we provide a new plugin: `js-plain-objects`, that 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 kotlinc.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 that uses a `fetch()` function to interact with a JavaScript API using external interfaces to describe the shape of the JavaScript objects:

```
import kotlinc.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(method = "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 method is not recognized, the wrong method
// (GET) is used.
fetch("https://google.com", options = js("{ method: '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-RC3"
}

```

Groovy

```

plugins {
    id "org.jetbrains.kotlin.plugin.js-plain-objects" version "2.0.0-RC3"
}

```

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-RC3, 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, in your gradle.properties file, set the following property:

```
kotlin.js.yarn=false
```

Gradle improvements

Kotlin 2.0.0-RC3 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 to try latest language version](#)
- [New JSON output format for build reports](#)

New Gradle DSL for compiler options in multiplatform projects

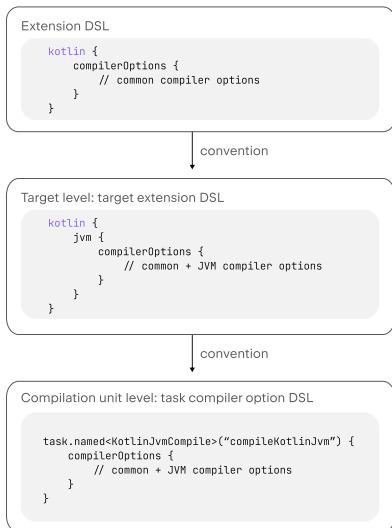
This feature is [Experimental](#). It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in [YouTrack](#).

Prior to Kotlin 2.0.0-RC3, 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-RC3 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, such as `commonMain`, as well as at the 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 (defaults) for a lower level:

- The extension compiler options values are the default for target compiler options, including shared source sets, like `commonMain`, `nativeMain`, and `commonTest`.
- Target compiler options values are used as the default for compilation unit (task) compiler options, for example, `compileKotlinJvm` and `compileTestKotlinJvm` tasks.

In turn, the configuration made at a lower level overrides related settings at a higher level:

- Task-level compiler options override related configuration at the target or the extension level.
- Target-level compiler options override related configuration at the extension level.

When configuring your project, keep in mind that some old ways of setting up compiler options were [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 been merged into the Kotlin repository. This will help with the transition of Compose projects to Kotlin 2.0.0, as the Compose compiler will always be released simultaneously with Kotlin. Additionally, this update also bumps the Compose compiler to version 2.0.0.

To use the new Compose compiler in your projects, apply the Gradle plugin in your build.gradle.kts file and set its version equal to the Kotlin version (2.0.0-RC2 or newer).

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-RC3, 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 publication of this attribute. 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 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 now deprecated.

Visibility changes in Gradle

This change impacts only Kotlin DSL users.

In Kotlin 2.0.0-RC3, 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 to 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 the 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

If your build script is configured incorrectly, you should see compiler warnings with suggestions on how to fix it. 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 [#eap Slack channel](#). [Get a Slack invite](#).

New directory for Kotlin data in Gradle projects

With this change, you may need to add the .kotlin directory to your project's .gitignore file.

In Kotlin 1.8.20, the Kotlin Gradle plugin started to store 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, since Kotlin 2.0.0-Beta2 we store Kotlin data in your <project-root-directory>/.kotlin by default. We will continue to store some data in .gradle/kotlin directory for backward compatibility.

There are new Gradle properties so that you can configure a directory of your choice and more:

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-RC3, 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](#).

It happened even if there was no task to compile code for a Kotlin/Native target 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-RC3, 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, 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
```

Please report any problems to our issue tracker [YouTrack](#), as this property will be removed in future releases.

Deprecating old ways of defining compiler options

In this release, we continue refining the ways of setting up compiler options. It should resolve ambiguity between different ways and make the project configuration more straightforward.

Since Kotlin 2.0.0-RC3, the following DSLs for specifying compiler options are deprecated:

- The HasCompilerOptions interface. It was inconsistent with other DSLs and had the same compilerOptions object as the Kotlin 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("-Xerror-tolerance-policy=SYNTAX")  
            }  
        }  
    }  
}
```

- The `KotlinCompile<KotlinOptions>` interface. Use `KotlinCompilationTask<CompilerOptions>` instead.
- 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 to try latest language version

Prior to Kotlin 2.0.0-RC3, 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-RC3, 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-RC3, 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,
        ...
      }
    }
  }
}
```

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-RC3, 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 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-RC3, 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

The release makes the `modCount` protected property of the `AbstractMutableList` interface 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's size or alter the list in a way that iterations in progress may 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

The release makes the `removeRange()` protected function of the `AbstractMutableList` interface 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. This function creates a new CharArray that contains characters from the specified string. The common `String.toCharArray(destination)`, however, moves String characters into an existing destination CharArray, which 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 ")
    }
}
```

What to expect from upcoming Kotlin EAP releases

Starting from Kotlin 2.0.0-RC1, you can use the K2 compiler in production.

The upcoming release candidates will further increase the stability of the K2 compiler. If you are currently using K2 in your project, we encourage you to stay updated on Kotlin releases and experiment with the updated K2 compiler. [Share your feedback on using Kotlin K2](#).

How to update to Kotlin 2.0.0-RC3

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. The bundled plugin supports upcoming Kotlin EAP releases.

To update to the new Kotlin EAP version, [change the Kotlin version](#) to 2.0.0-RC3 in your build scripts.

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.](#)

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){}[]'
```

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)

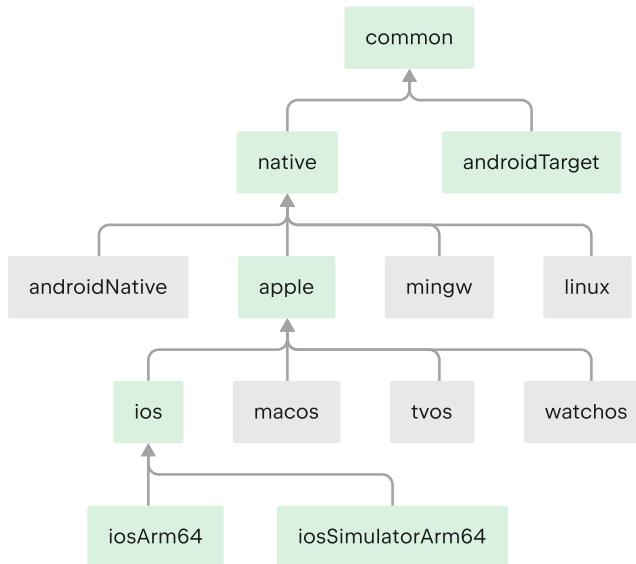
Kotlin 1.9.20

```
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 {  
    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:



A screenshot of an IDE interface showing a code editor with Kotlin build script code. The cursor is at the end of a line under 'sourceSets'. A tooltip labeled 'IDE completion' is displayed, showing suggestions: 'for', 'source', 'set', and 'names'. The code in the editor is:

```
1 plugins { this: PluginDependenciesSpecScope
2     kotlin("multiplatform")
3 }
4
5 group = "org.jetbrains.kotlin"
6 version = "1.0"
7
8 repositories { this: RepositoryHandler
9     mavenCentral()
10    mavenLocal()
11    google()
12 }
13
14 kotlin { this: KotlinMultiplatfo IDE completion
15     androidTarget() for
16     iosX64() source
17     iosSimulatorArm64() set
18     names
19
20 }
21
```

[Watch animation online.](#)

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.

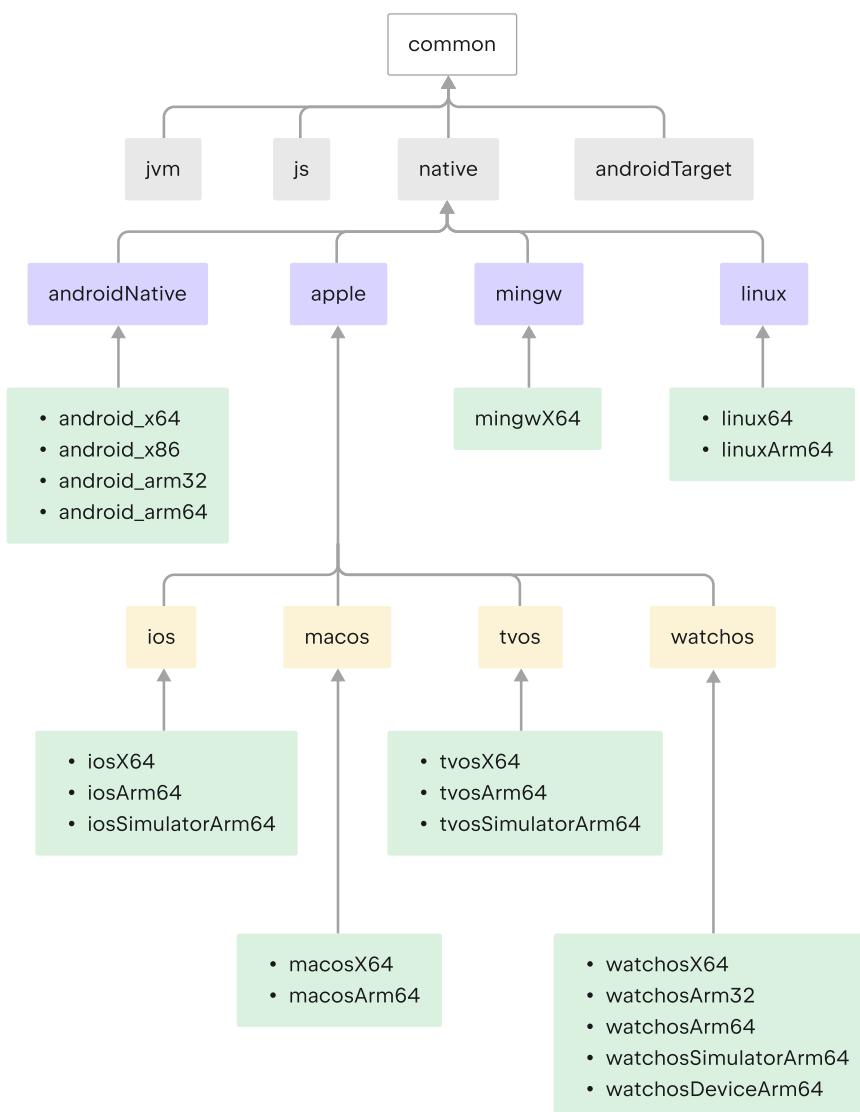
One option you have here is to find different source sets for your purposes, either in the default hierarchy template or ones that have been manually created. Another is to opt out of the template completely.

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.

[New Project](#)

[Template Gallery](#)

Coming Soon

Project Name

KotlinProject

Project ID

kmp.compose.demo



Android



With Compose Multiplatform UI toolkit based on Jetpack Compose



iOS



UI Implementation

Share UI (with Compose Multiplatform UI toolkit) Alpha

Do not share UI (use only SwiftUI)



Desktop



With Compose Multiplatform UI toolkit



Web

Coming Soon



Server



DOWNLOAD

Multiproject 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 Multiproject 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 project →](#)

Create a project

Full support for the Gradle configuration cache in Kotlin Multiproject

Previously, we introduced a [preview](#) of the Gradle configuration cache, which was available for Kotlin multiproject libraries. With 1.9.20, the Kotlin Multiproject 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 multiprojects 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 multiproject, a dependency for the standard library (`stdlib`) is added automatically to each source set. This is the easiest way to get started with your multiprojects.

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

Standard library version 1.9.20

Standard library version 1.9.10 and earlier

Standard library version 1.9.20

```
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")  
            }  
        }  
    }  
}
```

```
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 `kotlin.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-nullables types](#) – Learn about definitely non-nullables 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.](#)

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 KEP](#).

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](#).

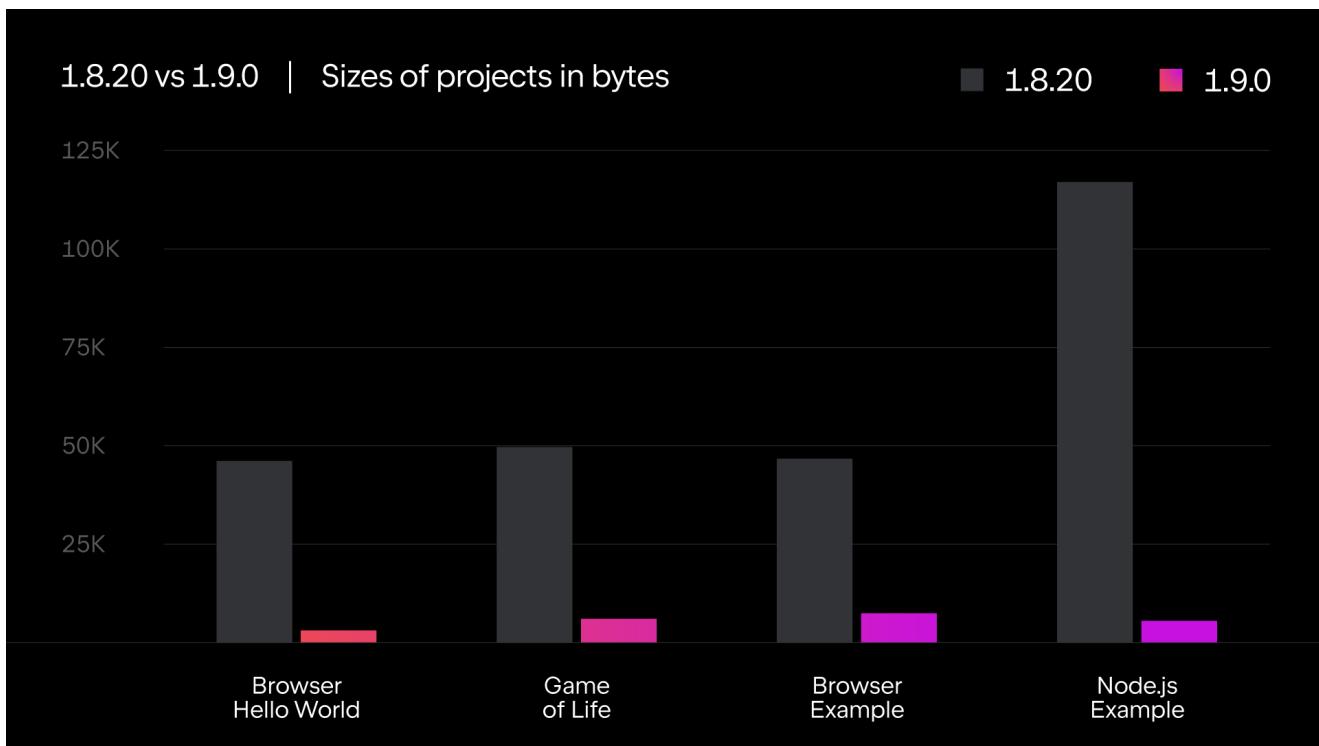
The Kotlin Multiplatform plugin still doesn't support the Gradle configuration cache with Xcode integration tasks or the [Kotlin CocoaPods Gradle plugin](#). We expect to add this feature in future Kotlin releases.

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 by following our [migration guide](#).

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) {
        usesModules() // 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

```
tasks.named<org.jetbrains.kotlin.gradle.tasks.KotlinNativeCompile>("compileKotlinLinuxX64") {
    compilerOptions {
        moduleName.set("my-module-name")
    }
}
```

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 $n: elapsed1=$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. <code>upperCase = false</code> .
-----------	---

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](#).

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.](#)

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
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.getDeclaredConstructors()[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>()
    .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 KEP](#).

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.

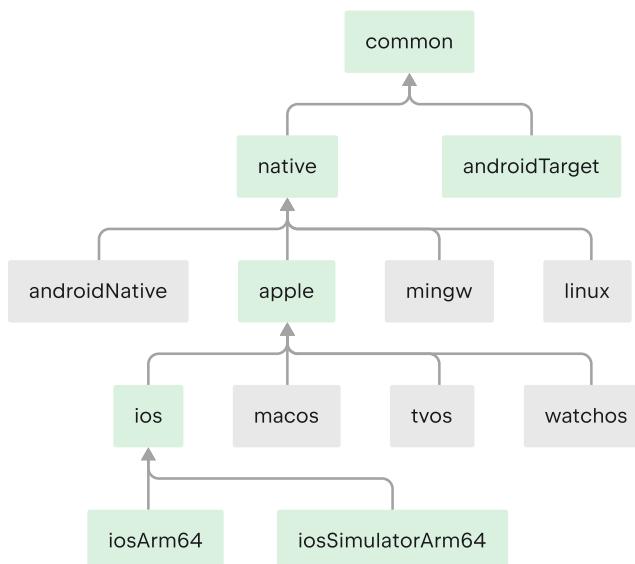
Set up your project

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:

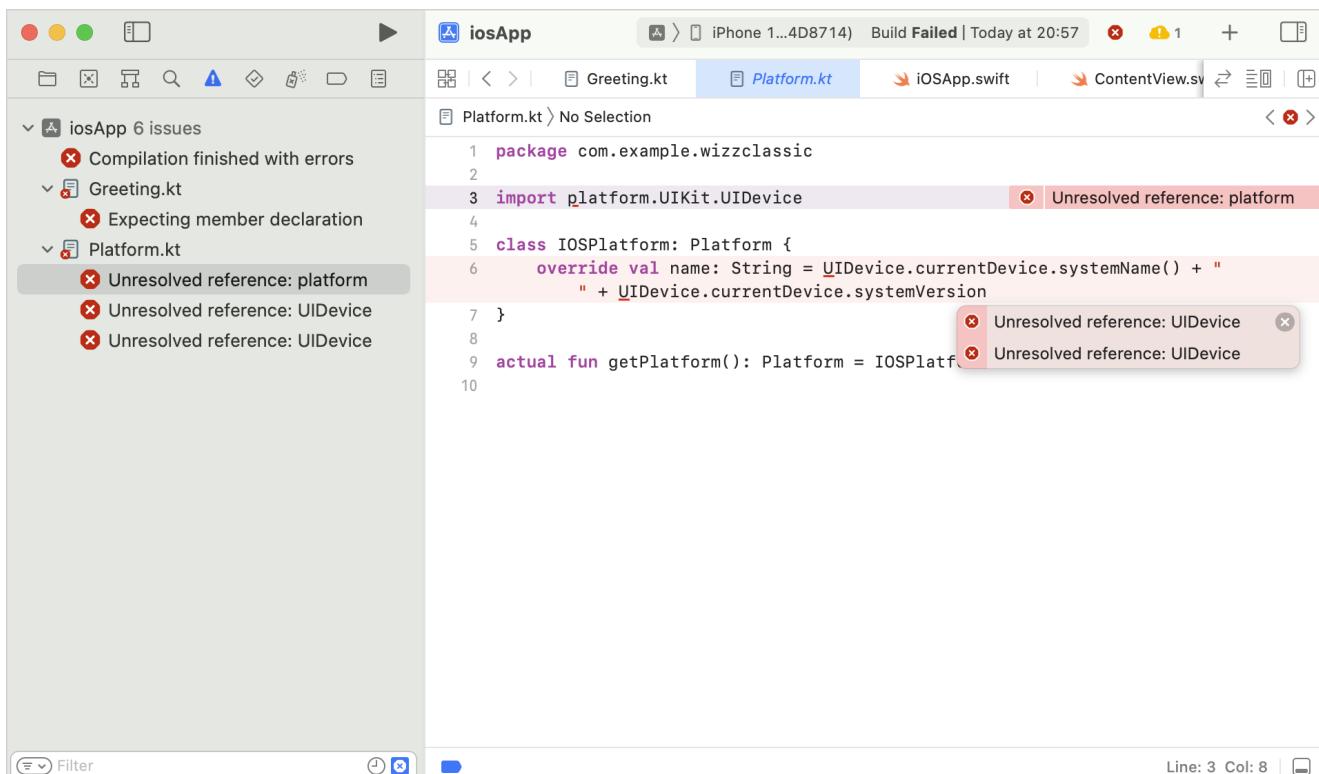
- There's no Kotlin 1.8.20 plugin available for IntelliJ IDEA 2023.1 EAP yet. Despite that, you can still set the Kotlin Gradle plugin version to 1.8.20 and try out composite builds in this IDE.
- If your projects include builds with a specified rootProject.name, composite builds may fail to resolve 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
<code>simple-names</code>	Variable names and simple function names are added. (Default)	<code>main</code>
<code>fully-qualified-names</code>	Variable names and fully qualified function names are added.	<code>com.example.kjs.playground.main</code>
<code>no</code>	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 {  
    compilerOptions.sourceMapNamesPolicy.set(org.jetbrains.kotlin.gradle.dsl.JsSourceMapNamesPolicy.SOURCE_MAP_NAMES_POLICY_FQ_NAMES  
        // 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](#).

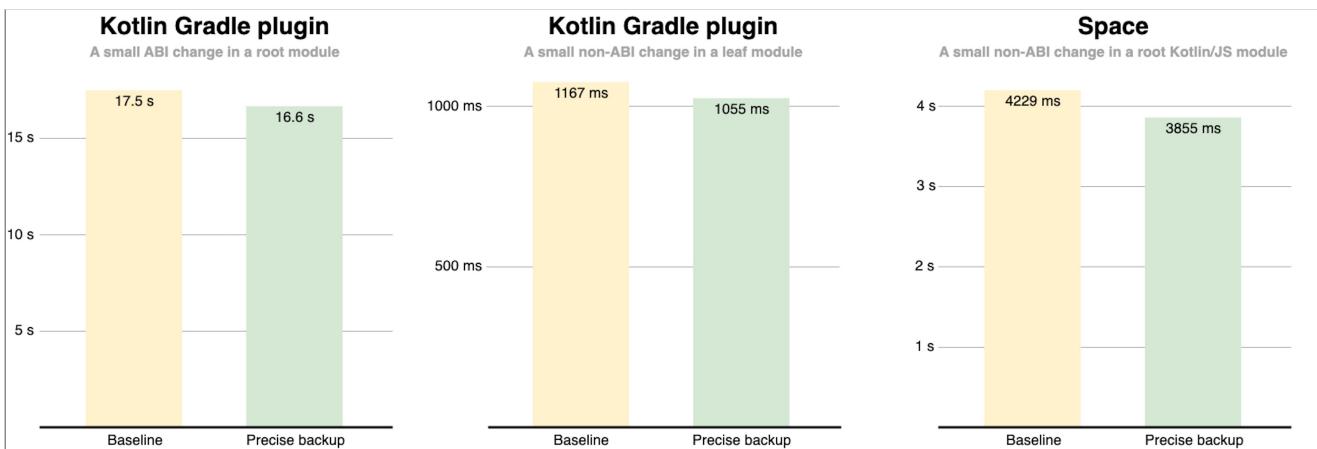
Starting with Kotlin 1.8.20, you can enable precise backup, whereby only those classes that Kotlin recompiles in the [incremental compilation](#) will be backed up. Both full and precise backups help to run builds incrementally again after compilation errors. Precise backup also saves build time compared 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.documentElement(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=".toByteArray()) // foBytes
// Alternatively:
// Base64.decode("Zm8=".toByteArray())

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](#)

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
------	-------------	-----------------

Current source set layout New source set layout

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 with the help of our [migration guide](#).

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. Learn more about these [execution strategies in the documentation](#). 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() {
    //sampleStart
    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)
//sampleEnd
}

```

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")
        OnErrorHandler.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,
            overwrite = true)
}

```

```

    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](#).

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.](#)

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 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.binaryobjcExportSuspendFunctionLaunchThreadRestriction=none
```

Do not add this option if you use the native-nt 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
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 optional parameters 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 using more [specialized ways to create targets](#).
- 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]")
//sampleEnd
}

```

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-nullables 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.](#)

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-nullables 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, [module-wide](#).

Stable definitely non-null types

In Kotlin 1.7.0, definitely non-null 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.nativedistribution.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-nullables](#)
- [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-nullables

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-nullables 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 modes](#).

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 via 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 [Gradle 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](#).

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.](#)

Language

In Kotlin 1.6.20, you can try two new language features:

- [Prototype of context receivers for Kotlin/JVM](#)
- [Definitely non-nullible 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-nullables types

Definitely non-nullables 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-null 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-null 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-nullables 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/stdcil/src/kotlin/util/Preconditions.kt:143:56) at 5 example.kexe 0x25fac8 kfun:#bar(){} + 104 [/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 0x10a8585486 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/stdcil/src/kotlin/util/Preconditions.kt:143:56) >> at 5 example.kexe 0x106689d35 kfun:#bar(){} + 117 [/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).lockFileDirectory =  
        file("my-kotlin-js-store")  
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension).lockFileName = '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).ignoreScripts = 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.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](#).

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 KEP](#).

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 KEP](#).

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 MinGW 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).download = 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).download = 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 'In'.

```
println("What is your nickname?")
val nickname = readln()
println("Hello, $nickname!")
```

```
fun main() {
    //sampleStart
    var sum = 0
    while (true) {
        val nextLine = readlnOrNull().takeUnless {
            it.isNullOrEmpty()
        } ?: break
        sum += nextLine.toInt()
    }
    println(sum)
}
```

```
//sampleEnd
}
```

The existing `readLine()` function will get a lower priority than `readIn()` and `readInOrNull()` 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() {
//sampleStart
    val x = listOf('b', 'c')
    val y = buildList {
        add('a')
        addAll(x)
        add('d')
    }
    println(y) // [a, b, c, d]
//sampleEnd
}
```

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() {
//sampleStart
    val duration = 10000
    println("There are ${duration.seconds.inWholeMinutes} minutes in $duration seconds")
    // There are 166 minutes in 10000 seconds
//sampleEnd
}
```

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() {
//sampleStart
    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"
//sampleEnd
}
```

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() {
//sampleStart
    val number: Short = 0b10001
    println(number
        .rotateRight(2)
        .toString(radix = 2)) // 10000000000100
    println(number
        .rotateLeft(2)
        .toString(radix = 2)) // 1000100
//sampleEnd
}
```

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() {
//sampleStart
    println(Regex("(.)").replace("Kotlin", """\$ \$1""")) // $ Kotlin
    println(Regex("(.)").replaceFirst("1.6.0", """\\ \$1""")) // \ 1.6.0
//sampleEnd
}
```

```
}
```

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.

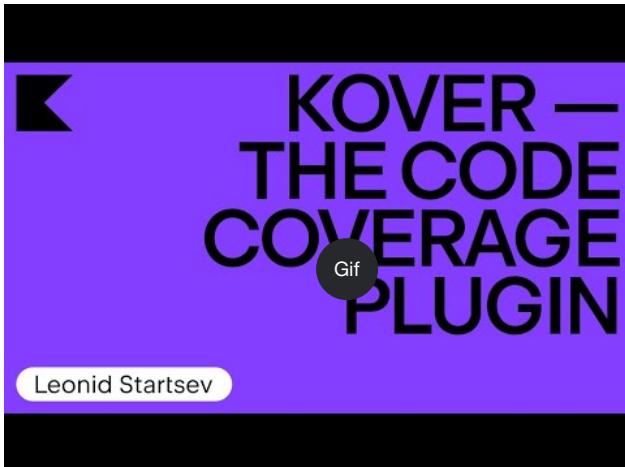
Tools

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](#).

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.](#)

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 all possible types or values of its subject or for some types plus an `else` branch. In other words, it covers all possible 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 dataSource = 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 KEP](#)

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'  
        baseUrl = "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

```
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 Ultimate](#).

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.jvmargs` property in the `gradle.properties` file:

```
kotlin.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

<code>Duration.days(45).toString()</code>	45.0d	45d
<code>Duration.days(1.5).toString()</code>	36.0h	1d 12h
<code>Duration.minutes(1230).toString()</code>	20.5h	20h 30m
<code>Duration.minutes(2415).toString()</code>	40.3h	1d 16h 15m
<code>Duration.minutes(920).toString()</code>	920m	15h 20m
<code>Duration.seconds(1.546).toString()</code>	1.55s	1.546s
<code>Duration.milliseconds(25.12).toString()</code>	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.toString() in certain cases, including serialization and interchange. Duration.toString() 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).
 - tolsoString().
- parseIsoString(), which only parses from the format produced by tolsoString().
- 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() {
    //sampleStart
    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"
    //sampleEnd
}
```

And here are some examples of `parseIsoString()` and `parseIsoStringOrNull()` usages:

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime

@ExperimentalTime
fun main() {
    //sampleStart
    val isoFormatString = "PT1H30M"
    val defaultFormatString = "1h 30m"
    println(Duration.parseIsoString(isoFormatString)) // "1h 30m"
    //println(Duration.parseIsoString(defaultFormatString)) // throws exception
    println(Duration.parseIsoStringOrNull(defaultFormatString)) // "null"
    //sampleEnd
}
```

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(){
//sampleStart
    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"
//sampleEnd
}
```

matchAt() returns the match if one is found or null if one isn't:

```
fun main(){
//sampleStart
    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"
//sampleEnd
}
```

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(){
//sampleStart
    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"
//sampleEnd
}
```

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.](#)

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.copyOf\(\)` 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.copyOf()` inside one array

We've improved the way `Array.copyOf()` 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() {
    //sampleStart
    val ten = '\u0661'.digitToInt() + '\u0039'.digitToInt() // ARABIC-INDIC DIGIT ONE + DIGIT NINE
    println(ten)
    //sampleEnd
}
```

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() {
//sampleStart
    val latinCapitalA = 'A' // has "Lu" general category
    val circledLatinCapitalA = 'Ⓐ' // has "Other_Uppercase" property
    println(latinCapitalA.isUpperCase() && circledLatinCapitalA.isUpperCase())
//sampleEnd
}
```

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](#).

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 `-ksam-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](#).

```
// 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")
```

[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() {  
    //sampleStart  
    println("Floored division -5/3: ${(-5).floorDiv(3)}")  
    println("Modulus: ${(-5).mod(3)}")  
  
    println("Truncated division -5/3: ${-5 / 3}")  
    println("Remainder: ${-5 % 3}")  
    //sampleEnd  
}
```

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() {  
    //sampleStart  
    val duration = Duration.milliseconds(120000)  
    println("There are ${duration.inWholeSeconds} seconds in ${duration.inWholeMinutes} minutes")  
    //sampleEnd  
}
```

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() {  
    //sampleStart  
    val chars = listOf('a', '1', '+')  
    val (letterOrDigitList, notLetterOrDigitList) = chars.partition { it.isLetterOrDigit() }  
    println(letterOrDigitList) // [a, 1]  
    println(notLetterOrDigitList) // [+]
```

```
//sampleEnd  
}
```

Functions for checking the case of a char:

- [Char.isLowerCase\(\)](#)
- [Char.isUpperCase\(\)](#)
- [Char.isTitleCase\(\)](#)

```
fun main() {  
//sampleStart  
    val chars = listOf('܂', '܄', '܆', '܂', '܄', '܆', '܂', '܄', '܆')  
    val (titleCases, notTitleCases) = chars.partition { it.isTitleCase() }  
    println(titleCases) // [܂, ܄, ܆]  
    println(notTitleCases) // [܂, ܄, ܆]  
//sampleEnd  
}
```

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() {  
//sampleStart  
    val data = listOf("Kotlin", "1.5")  
    println(data.firstNotNullOf(String::toDoubleOrNull))  
    println(data.firstNotNullOfOrNull(String::toIntOrNull))  
//sampleEnd  
}
```

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() {  
//sampleStart  
    println("true".toBooleanStrict())  
    println("1".toBooleanStrictOrNull())  
    // println("1".toBooleanStrict()) // Exception  
//sampleEnd  
}
```

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 }
    assertEquals(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](#).

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](#).

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()`, `decapsalize()` 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 "KOTLiN", 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](#).

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 kotlinc-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 your code even if it contains errors, for example, if you want to try certain things before the whole application is 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 [ignoring compilation errors](#) with 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 `NSError`'s into Kotlin exceptions of type `ForeignException`. They hold the references to the original `NSError`'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 capabilities.

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.

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.

```
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 {
    // 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 KEP.](#)

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 with default argument 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 with default argument values

Now you can use callable references to functions with default argument 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 additional overloads for the function apply to use the default argument values.

```
// 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.

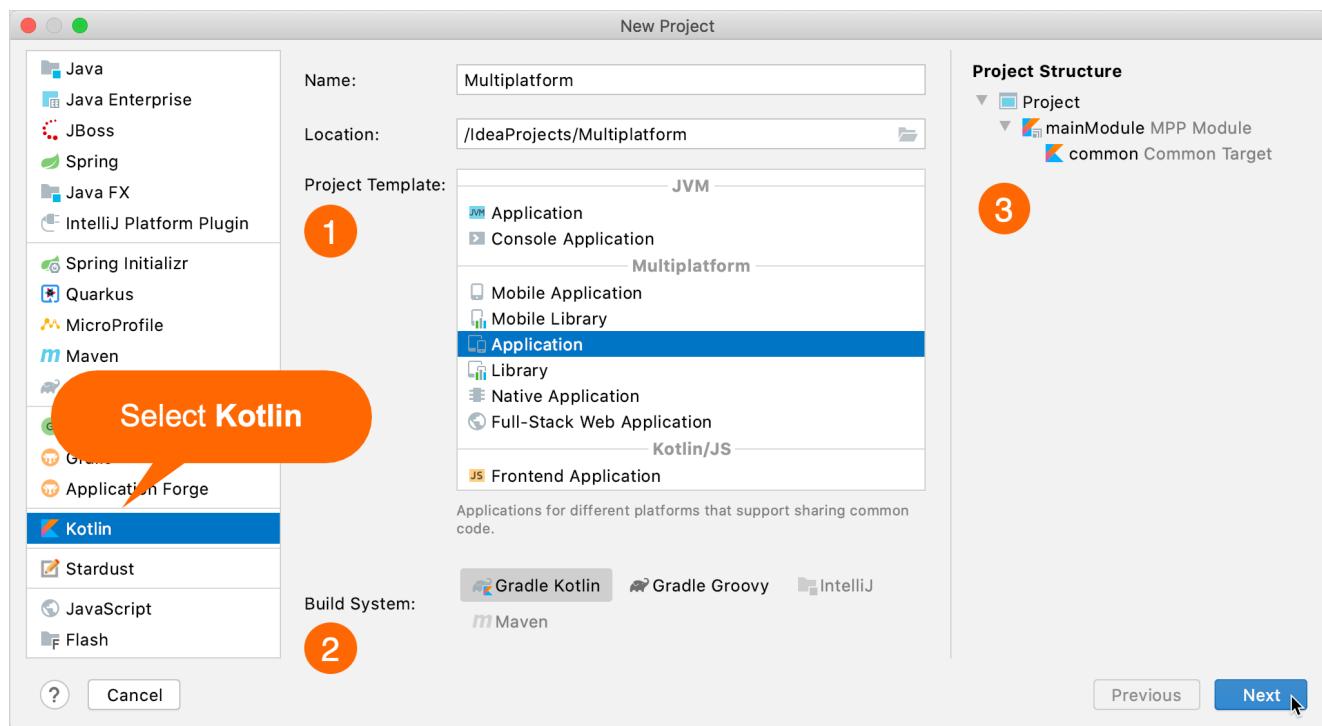
New tools in the IDE

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.



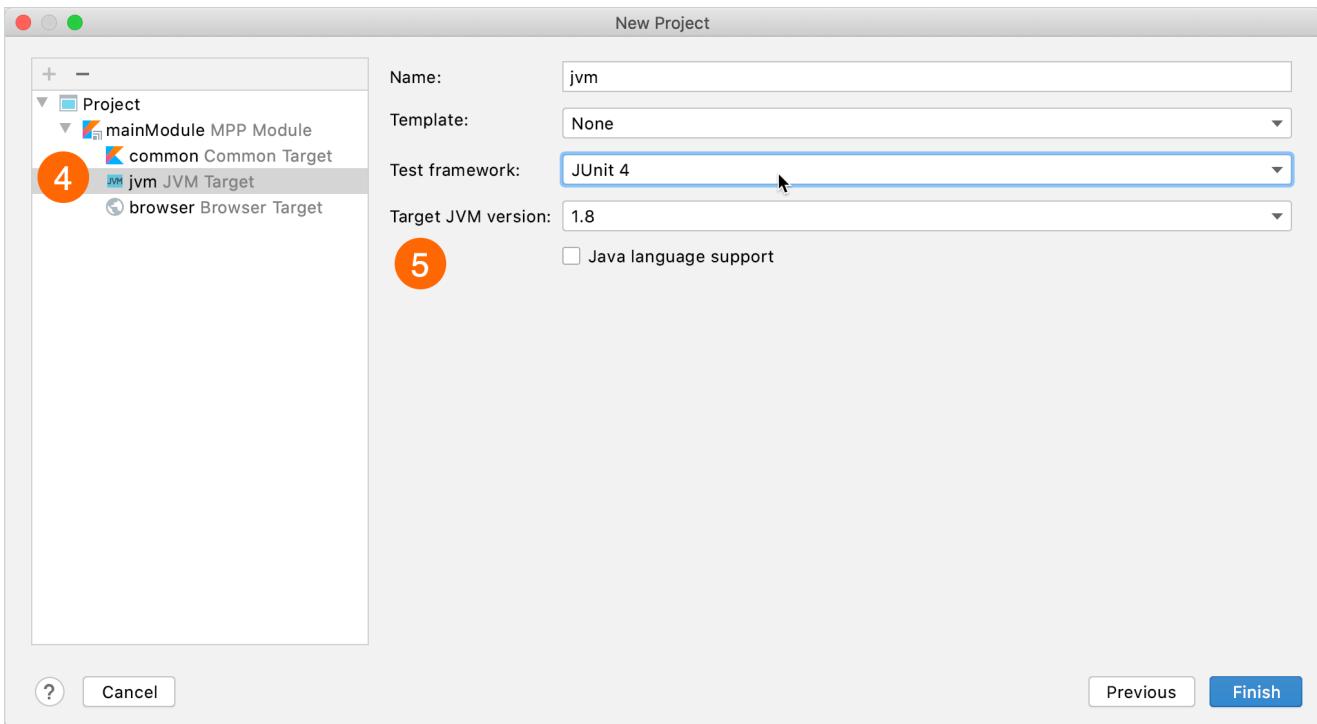
Kotlin Project Wizard – Multiplatform project

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](#)

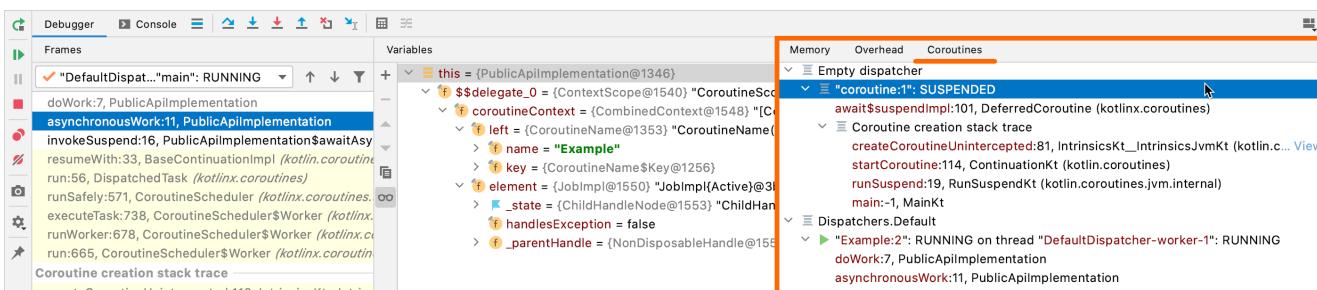
Coroutine Debugger

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.

The screenshot shows the IntelliJ IDEA interface with the Coroutines tab selected. A dump of a suspended coroutine is displayed. The dump shows the coroutine creation stack trace with frames like await\$suspendImpl, createCoroutineUnintercepted, startCoroutine, and runSuspend. A button labeled 'Get Coroutines Dump' is highlighted.

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?:

```
//sampleStart
val rulesMap: Map<String, (String?) -> Boolean> = mapOf(
```

```

    "weak" to { it != null },
    "medium" to { !it.isNullOrEmpty() },
    "strong" to { it != null && "^[a-zA-Z0-9]+$".toRegex().matches(it) }
)
//sampleEnd

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")
    }
}

//sampleStart
fun perform(animal: Animal) {
    val kFunction: KFunction<*> = when (animal) {
        is Cat -> animal::meow
        is Dog -> animal::woof
    }
    kFunction.call()
}
//sampleEnd

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`, `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` (.dSYMs) 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

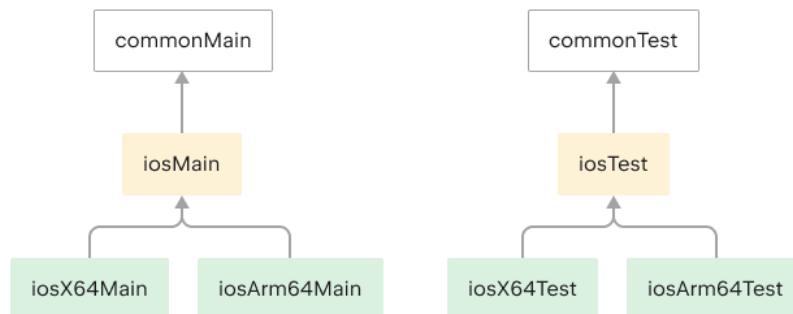
With the new hierarchical project structure support, you can share code among [several platforms](#) in a [multiplatform project](#).

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.



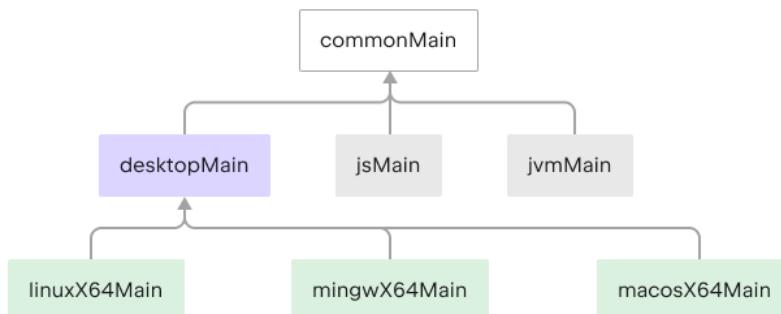
Code shared for iOS targets

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:1.8.1")
            }
        }
    }
}
```

Groovy

```
kotlin {
```

```

sourceSets {
    commonMain {
        dependencies {
            implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.8.1'
        }
    }
}

```

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.

Code insight is disabled to avoid the Gradle build configuration.

```
plugins {
    java
}

group = "org.example"
version = "1.0-SNAPSHOT"
```

The Gradle configuration phase needs to be run to get the Script Configuration. Loading the script configuration is disabled by default, since it can be resource-intensive for large Gradle projects.
Click "Load Configuration" to evaluate the Gradle Kotlin DSL script.

*.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.

```
plugins { this: PluginDependenciesSpecScope
    java
    application
}

group = "org.example"
version = "1.0-SNAPSHOT"
```

*.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.

Code insight unavailable (configuration for this script wasn't received during the last Gradle project import).

The Gradle project that evaluates this script needs to be imported to have it analyzed by the IDE. Try re-importing the linked Gradle project or link a new Gradle project that evaluates this script.
Alternatively, you can add it to standalone scripts, and its configuration will be loaded separately.
NOTE: Each standalone script requires a separate Gradle configuration phase to be executed on update. This can be resource intensive for large Gradle projects.

*.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() {
    //sampleStart
    val set = setOfNotNull(null, 1, 2, 0, null)
    println(set)
    //sampleEnd
}
```

- `shuffled()` for sequences.

```
fun main() {
    //sampleStart
    val numbers = (0 until 50).asSequence()
    val result = numbers.map { it * 2 }.shuffled().take(5)
    println(result.toList()) //five random even numbers below 100
    //sampleEnd
}
```

- `*Indexed()` counterparts for `onEach()` and `flatMap()`. The operation that they apply to the collection elements has the element index as a parameter.

```
fun main() {
    //sampleStart
    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()
        }
    //sampleEnd
        println(kotlin)
}
```

- *OrNull() counterparts randomOrNull(), reduceOrNull(), and reduceIndexedOrNull(). They return null on empty collections.

```
fun main() {
//sampleStart
    val empty = emptyList<Int>()
    empty.reduceOrNull { a, b -> a + b }
    //empty.reduce { a, b -> a + b } // Exception: Empty collection can't be reduced.
//sampleEnd
}
```

- 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() {
//sampleStart
    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 }
//sampleEnd
    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() {
//sampleStart
    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
//sampleEnd
    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() {
//sampleStart
    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 }
//sampleEnd
    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() {
```

```
//sampleStart
    val list = listOf("kot", "lin")
    val lettersList = list.flatMap { it.asSequence() }
    val lettersSeq = list.asSequence().flatMap { it.toList() }
//sampleEnd
    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() {
//sampleStart
    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]
//sampleEnd
}
```

Additionally, there are new functions for conversions between CharArray/ByteArray and String:

- ByteArray.decodeToString() and String.encodeToByteArray()
- CharArray.concatToString() and String.toCharArray()

```
fun main() {
//sampleStart
    val str = "kotlin"
    val array = str.toCharArray()
    println(array.concatToString())
//sampleEnd
}
```

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() {
//sampleStart
    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() {
//sampleStart
    println(buildString {
        appendLine("Hello,")
        appendLine("world")
    })
//sampleEnd
}

```

Bit operations

New functions for bit manipulations:

- `countOneBits()`
- `countLeadingZeroBits()`
- `countTrailingZeroBits()`
- `takeHighestOneBit()`
- `takeLowestOneBit()`
- `rotateLeft()` and `rotateRight()` (experimental)

```

fun main() {
//sampleStart
    val number = "1010000".toInt(radix = 2)
    println(number.countOneBits())
    println(number.countTrailingZeroBits())
    println(number.takeHighestOneBit().toString(2))
//sampleEnd
}

```

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.

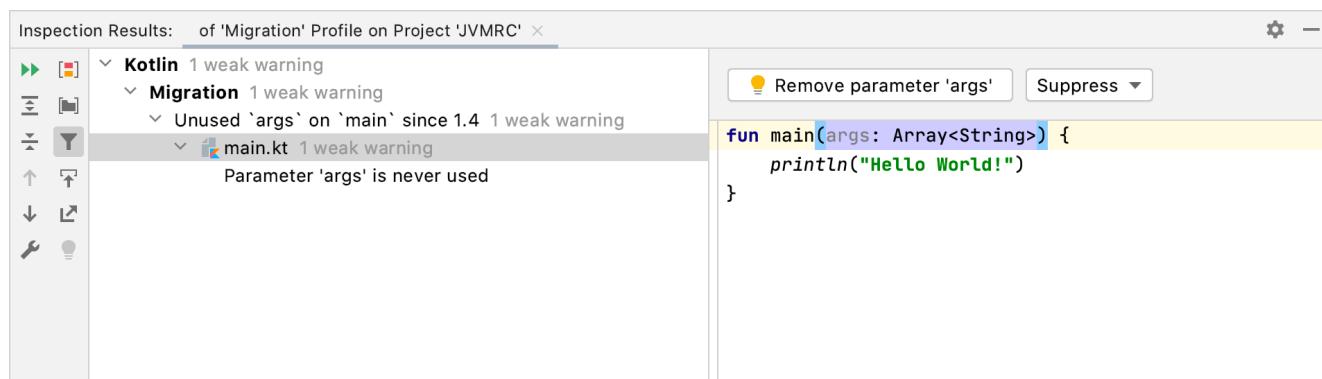
A Kotlin Migration

Migrations for Kotlin code are available...

[Run migrations](#)

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](#).

What's new in Kotlin 1.3

Released: 29 October 2018

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:

```

//sampleStart
fun bar(x: String?) {
    if (!x.isNullOrEmpty()) {
        println("length of '$x' is ${x.length}") // Yay, smartcast to not-null!
    }
}
//sampleEnd
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)
//sampleStart
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)
}
//sampleEnd
```

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() {
//sampleStart
// 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
//sampleEnd
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() {
//sampleStart
    val number = Random.nextInt(42) // number is in range [0, limit)
    println(number)
//sampleEnd
}
```

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.copyOf(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() {
//sampleStart
    val sourceArr = arrayOf("k", "o", "t", "l", "i", "n")
    val targetArr = sourceArr.copyOf(arrayOfNulls<String>(6), 3, startIndex = 3, endIndex = 6)
    println(targetArr.contentToString())

    sourceArr.copyOf(targetArr, startIndex = 0, endIndex = 3)
    println(targetArr.contentToString())
//sampleEnd
}
```

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() {
//sampleStart
    val keys = 'a'..'f'
    val map = keys.associateWith { it.toString().repeat(5).capitalize() }
    map.forEach { println(it) }
//sampleEnd
}
```

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() {
//sampleStart
    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"))
//sampleEnd
}
```

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() {
//sampleStart
    val s = "    \n"
    println(s.ifBlank { "<blank>" })
    println(s.ifBlank { null })
//sampleEnd
}
```

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.

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](#)

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() {
//sampleStart
        println("isInitialized before assignment: " + this::lateinitVar.isInitialized)
        lateinitVar = "value"
        println("isInitialized after assignment: " + this::lateinitVar.isInitialized)
//sampleEnd
    }
}

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:

```

//sampleStart
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)" }
//sampleEnd

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 {
//sampleStart
    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<*>
//sampleEnd
}

```

```

        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>) {
//sampleStart
    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
        }
    }
//sampleEnd
}

```

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>) {
    //sampleStart
    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 }

    //sampleEnd

    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>) {
    //sampleStart
    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")
    //sampleEnd
}
```

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
- Exponentiation: 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](#)

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:

```

//sampleStart
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"
//sampleEnd

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:

```

//sampleStart
val numberRegex = "\d+".toRegex()
val numbers = listOf("abc", "123", "456").filter(numberRegex::matches)
//sampleEnd

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:

```
//sampleStart
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)))
//sampleEnd

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>) {
//sampleStart
    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!" })
//sampleEnd
}
```

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")

//sampleStart
    map.forEach { _, value -> println("$value!") }
//sampleEnd
}
```

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>) {
//sampleStart
    val (_, status) = getResult()
//sampleEnd
    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:

```
//sampleStart
val oneMillion = 1_000_000
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
//sampleEnd

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:

```
//sampleStart
data class Person(val name: String, val age: Int) {
    val isAdult get() = age >= 20 // Property type inferred to be 'Boolean'
}
//sampleEnd

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](#).

```
//sampleStart
public val <T> List<T>.lastIndex: Int
    inline get() = this.size - 1
//sampleEnd

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>) {
//sampleStart
    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...")
    }
//sampleEnd
}
```

```
}
```

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.

```
//sampleStart
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}
//sampleEnd

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(), takelf(), 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
}

//sampleStart
fun Block.copy() = Block().also {
    it.content = this.content
}
//sampleEnd

// 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)
}
```

takelf 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"

    //sampleStart
    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
    //sampleEnd
```

```

    println("'$keyword' was found in '$input'")
    println(input)
    println(" ".repeat(index) + "^")
}

```

`takeUnless` is the same as `takelf`, 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) {
//sampleStart
    val result = string.takeUnless(String::isEmpty)
//sampleEnd

    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(' ')
//sampleStart
    val frequencies = words.groupingBy { it.first() }.eachCount()
//sampleEnd
    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>) {
//sampleStart
    val map = mapOf("key" to 42)
    val emptyMap = map - "key"
//sampleEnd

    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>) {
//sampleStart
    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 })
//sampleEnd

    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>) {
//sampleStart
    val squares = List(10) { index -> index * index }
    val mutable = MutableList(10) { 0 }
//sampleEnd

    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
//sampleEnd

    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>) {
//sampleStart
    val array = arrayOf("a", "b", "c")
    println(array.toString()) // JVM implementation: type-and-hash gibberish
}
```

```
    println(array.contentToString()) // nicely formatted as list
//sampleEnd
}
```

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 Ant/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)
    }
}
```

Kotlin releases

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 each language and tooling release, we also ship several preview (EAP) versions for you to try new features before they are released. See [Early Access Preview](#) for details.

Update to a new release

To upgrade your project to a new release, you need to update your build script file. For example, to update to Kotlin 2.0.0, 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 "2.0.0"  
    // For example, if your target environment is JVM:  
    // kotlin("jvm") version "2.0.0"  
    // If you target is Kotlin Multiplatform  
    // kotlin("multiplatform") version "2.0.0"  
}
```

Groovy

```
plugins {  
    // Replace `<...>` with the plugin name appropriate for your target environment  
    id 'org.jetbrains.kotlin.<...>' version '2.0.0'  
    // For example, if your target environment is JVM:  
    // id 'org.jetbrains.kotlin.jvm' version '2.0.0'  
    // If you target is Kotlin Multiplatform  
    // id 'org.jetbrains.kotlin.multiplatform' version '2.0.0'  
}
```

If you have projects created with earlier Kotlin versions, change the Kotlin version in your projects and update kotlincx libraries if necessary.

If you are migrating to the new language release, Kotlin plugin's migration tools will help you with the migration.

IDE support

Even with the release of the K2 compiler, IntelliJ IDEA and Android Studio still use the previous compiler by default for code analysis, code completion, highlighting, and other IDE-related features.

Starting from 2024.1, IntelliJ IDEA can use the new K2 compiler to analyze your code with its K2 Kotlin mode. To enable it, go to [Settings | Languages & Frameworks | Kotlin](#) and select the Enable the K2-based Kotlin plugin option.

The K2 Kotlin mode is in Alpha. The performance and stability of code highlighting and code completion have been significantly improved, but not all IDE features are supported yet.

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 the [migration guide](#).

Kotlin release compatibility

Learn more about [types of Kotlin releases and their compatibility](#)

Release details

The following table lists details of the latest Kotlin releases:

You can also use [preview versions of Kotlin](#).

[Build info](#) [Build highlights](#)

[Build info](#) [Build highlights](#)

2.0.0 A language release with the Stable Kotlin K2 compiler.

Released: [Learn more about Kotlin 2.0.0 in What's new in Kotlin 2.0.0.](#)
May 21, 2024

[Release on](#)
[GitHub](#)

1.9.24 A bug fix release for Kotlin 1.9.20, 1.9.21, 1.9.22, and 1.9.23.

Released: [Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.](#)
May 7, 2024

[Release on](#)
[GitHub](#)

1.9.23 A bug fix release for Kotlin 1.9.20, 1.9.21, and 1.9.22.

Released: [Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.](#)
March 7, 2024

[Release on](#)
[GitHub](#)

1.9.22 A bug fix release for Kotlin 1.9.20 and 1.9.21.

Released: [Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.](#)
December 21,
2023

[Release on](#)
[GitHub](#)

1.9.21 A bug fix release for Kotlin 1.9.20.

Released: [Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.](#)
November 23,
2023

[Release on](#)
[GitHub](#)

1.9.20 A feature release with Kotlin K2 compiler in Beta and Stable Kotlin Multiplatform.

Released: Learn more in:
November 1,
2023 • [What's new in Kotlin 1.9.20](#)

[Release on](#)
[GitHub](#)

[Build info](#) [Build highlights](#)

1.9.10 A bug fix release for Kotlin 1.9.0.

Released: Learn more about Kotlin 1.9.0 in [What's new in Kotlin 1.9.0](#).
August 23,
2023

[Release on GitHub](#) For Android Studio Giraffe and Hedgehog, the Kotlin plugin 1.9.10 will be delivered with upcoming Android Studios updates.

1.9.0 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.

Released: July 6, 2023

Learn more in:

[Release on GitHub](#) • [What's new in Kotlin 1.9.0](#)
• [What's new in Kotlin YouTube video](#)

1.8.22 A bug fix release for Kotlin 1.8.20.

Released: Learn more about Kotlin 1.8.20 in [What's new in Kotlin 1.8.20](#).
June 8, 2023

[Release on GitHub](#)

1.8.21 A bug fix release for Kotlin 1.8.20.

Released: Learn more about Kotlin 1.8.20 in [What's new in Kotlin 1.8.20](#).
April 25, 2023

[Release on GitHub](#)

For Android Studio Flamingo and Giraffe, the Kotlin plugin 1.8.21 will be delivered with upcoming Android Studios updates.

1.8.20 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.

Released: April 3, 2023 Learn more in:

[Release on GitHub](#) • [What's new in Kotlin 1.8.20](#)
• [What's new in Kotlin YouTube video](#)

1.8.10 A bug fix release for Kotlin 1.8.0.

Released: Learn more about [Kotlin 1.8.0](#).
February 2,
2023

[Release on GitHub](#) For Android Studio Electric Eel and Flamingo, the Kotlin plugin 1.8.10 will be delivered with upcoming Android Studios updates.

Build info Build highlights

1.8.0 A feature release with improved kotlin-reflect performance, new recursively copy or delete directory content experimental functions for JVM, improved Objective-C/Swift interoperability.

Released:

December 28, Learn more in:

2022

- [What's new in Kotlin 1.8.0](#)

Release on

[GitHub](#)

- [Compatibility guide for Kotlin 1.8.0](#)

1.7.21 A bug fix release for Kotlin 1.7.20.

Released: Learn more about Kotlin 1.7.20 in [What's new in Kotlin 1.7.20](#).

November 9,
2022

Release on

[GitHub](#)

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 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.

Released:

September
29, 2022 Learn more in:

- [What's new in Kotlin 1.7.20](#)

Release on
[GitHub](#)

- [What's new in Kotlin YouTube video](#)
- [Compatibility guide for Kotlin 1.7.20](#)

Learn more about [Kotlin 1.7.20](#).

1.7.10 A bug fix release for Kotlin 1.7.0.

Released: Learn more about [Kotlin 1.7.0](#).
July 7, 2022

Release on
[GitHub](#)

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 A feature release with Kotlin K2 compiler in Alpha for JVM, stabilized language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs.

Released:

June 9, 2022 Learn more in:

Release on
[GitHub](#)

- [What's new in Kotlin 1.7.0](#)
- [What's new in Kotlin YouTube video](#)
- [Compatibility guide for Kotlin 1.7.0](#)

[Build info](#) [Build highlights](#)

1.6.21 A bug fix release for Kotlin 1.6.20.

Released: [Learn more about Kotlin 1.6.20.](#)

April 20, 2022

[Release on](#)

[GitHub](#)

1.6.20 An incremental release with various improvements such as:

Released:

- Prototype of context receivers

April 4, 2022

- Callable references to functional interface constructors

[Release on](#)

[GitHub](#)

- 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 A bug fix release for Kotlin 1.6.0.

Released: [Learn more about Kotlin 1.6.0.](#)

December 14,
2021

[Release on](#)

[GitHub](#)

1.6.0 A feature release with new language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs.

Released: Learn more in:

November 16,
2021

- [Release blog post](#)

[Release on](#)

[GitHub](#)

- [What's new in Kotlin 1.6.0](#)
- [Compatibility guide](#)

1.5.32 A bug fix release for Kotlin 1.5.31.

Released: [Learn more about Kotlin 1.5.30.](#)

November 29,
2021

[Release on](#)

[GitHub](#)

1.5.31 A bug fix release for Kotlin 1.5.30.

Released: Learn more about [Kotlin 1.5.30](#).
September
20, 2021

[Release on](#)
[GitHub](#)

1.5.30 An incremental release with various improvements such as:

Released: • Instantiation of annotation classes on JVM
August 23, • Improved opt-in requirement mechanism and type inference
2021

[Release on](#)
[GitHub](#) • Kotlin/JS IR backend in Beta
• Support for Apple Silicon targets

- Improved CocoaPods support
- 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 A bug fix release for Kotlin 1.5.20.

Released: Learn more about [Kotlin 1.5.20](#).
July 13, 2021

[Release on](#)
[GitHub](#)

1.5.20 An incremental release with various improvements such as:

Released: • String concatenation via invokedynamic on JVM by default
June 24, 2021 • Improved support for Lombok and support for JSpecify

[Release on](#)
[GitHub](#) • Kotlin/Native: KDoc export to Objective-C headers and faster `Array.copyOf()` 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 A bug fix release for Kotlin 1.5.0.

Released: Learn more about [Kotlin 1.5.0](#).
May 24, 2021

[Release on](#)
[GitHub](#)

1.5.0 A feature release with new language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs.

Released: Learn more in:
May 5, 2021

- [Release blog post](#)
- [What's new in Kotlin 1.5.0](#)
- [Compatibility guide](#)

1.4.32 A bug fix release for Kotlin 1.4.30.

Released: Learn more about [Kotlin 1.4.30](#).
March 22,
2021

[Release on](#)
[GitHub](#)

1.4.31 A bug fix release for Kotlin 1.4.30

Released: Learn more about [Kotlin 1.4.30](#).
February 25,
2021

[Release on](#)
[GitHub](#)

1.4.30 An incremental release with various improvements such as:

Released:

- 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 A bug fix release for Kotlin 1.4.20

Released: [Learn more about Kotlin 1.4.20.](#)

December 7,
2020

[Release on
GitHub](#)

1.4.20 An incremental release with various improvements such as:

Released:

- Supporting new JVM features, like string concatenation via invokedynamic
- Improved performance and exception handling for Kotlin Multiplatform Mobile projects

November 23,
2020 [Release on
GitHub](#)

Learn more in:

- [Release blog post](#)
- [What's new in Kotlin 1.4.20](#)

1.4.10 A bug fix release for Kotlin 1.4.0.

Released: [Learn more about Kotlin 1.4.0.](#)

September 7,
2020

[Release on
GitHub](#)

1.4.0 A feature release with many features and improvements that mostly focus on quality and performance.

Released: Learn more in:

August 17,
2020 [Release blog post](#)

[Release on
GitHub](#)

- [What's new in Kotlin 1.4.0](#)
- [Compatibility guide](#)
- [Migrating to Kotlin 1.4.0](#)

1.3.72 A bug fix release for Kotlin 1.3.70.

Released: [Learn more about Kotlin 1.3.70.](#)

April 15, 2020

[Release on
GitHub](#)

Kotlin roadmap

Last modified on December 2023

Next update June 2024

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 a big picture. Here's a list of our key projects – the most important things we focus on delivering:

- K2 compiler: a rewrite of the Kotlin compiler optimized for speed, parallelism, and unification. It will also allow us to introduce many highly-anticipated language features.
- K2-based IntelliJ plugin: faster code completion, highlighting, and search, together with more stable code analysis.
- Kotlin Multiplatform: streamline build setup and enhance the iOS development experience.
- Experience of library authors: a set of documentation and tools helping to set up, develop, and publish Kotlin libraries.

Kotlin roadmap by subsystem

To view the biggest projects we're working on, visit the [YouTrack board](#) or the [Roadmap details](#) table.

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](#)).

YouTrack board

Visit the [roadmap board](#) in our issue tracker YouTrack 

Roadmap details

Subsystem In focus now

Language

[List of all upcoming language features](#)

Compiler

-   [Kotlin/Wasm: Switch wasm-wasi target of libraries to WASI Preview 2](#)
-   [Kotlin/Wasm: Support Component Model](#)
- [Promote K2 compiler to Stable](#)
- [Support debugging inline functions on Android](#)
- [Make Kotlin/Wasm suitable for standalone Wasm VMs](#)

Subsystem In focus now

- Multiplatform
- [Unify inline semantics between all Kotlin targets](#)
 - [Support SwiftPM for Kotlin Multiplatform users](#)
 - [Swift export: Design and implement support for Kotlin classes and interfaces](#)
 - [Improve the new Kotlin/Native memory manager's robustness and performance, and deprecate the old one](#)
 - [Stabilize klib: Make binary compatibility easier for library authors](#)
 - [Improve Kotlin/Native compilation time](#)

[Kotlin Multiplatform development roadmap for 2024](#)

- Tooling
- [Support Gradle project isolation](#)
 - [Improve integration of Kotlin/Native toolchain into Gradle](#)
 - [Kotlin Notebook: Light Notebooks and improved experience exploring data from HTTP endpoints](#)
 - [Improve Kotlin build reports](#)
 - [First public release of K2-based IntelliJ plugin](#)
 - [Improve performance and code analysis stability of the current IDE plugin](#)
 - [Expose stable compiler arguments in Gradle DSL](#)
 - [Improve Kotlin scripting and experience with .gradle.kts](#)

- Library ecosystem
- [Promote kotlinx-datetime to Beta](#)
 - [Provide initial series of kotlinx-io releases](#)
 - [Release kotlinx-metadata-jvm as Stable](#)
 - [Promote kotlinx-kover to Beta](#)
 - [Release Dokka as Stable](#)

Ktor and Exposed roadmaps:

- [Ktor framework roadmap](#)
- [Exposed library roadmap](#)

- 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 July 2023

Completed items

We've completed the following items from the previous roadmap:

- Compiler: [Promote Kotlin/Wasm to Alpha](#)
- Multiplatform: [Promote Kotlin Multiplatform to Stable](#)

New items

We've added the following items to the roadmap:

- Compiler: [Kotlin/Wasm: Switch wasm-wasi target of libraries to WASI Preview 2](#)
- Compiler: [Kotlin/Wasm: Support Component Model](#)
- Multiplatform: [Unify inline semantics between all Kotlin targets](#)
- Multiplatform: [Support SwiftPM for Kotlin Multiplatform users](#)
- Multiplatform: [Swift export: Design and implement support for Kotlin classes and interfaces](#)
- Tooling: [Support Gradle project isolation](#)
- Tooling: [Improve integration of Kotlin/Native toolchain into Gradle](#)
- Tooling: [Kotlin Notebook: Light Notebooks and improved experience exploring data from HTTP endpoints](#)
- Library ecosystem [Promote kotlinx-datetime to Beta](#)

Removed items

We've removed the following item from the roadmap:

- Multiplatform: [Improve exporting Kotlin code to Objective-C](#)

Some items were removed from the roadmap but not dropped completely. In some cases, we've merged previous roadmap items with the current ones.

Items in progress

All other previously identified roadmap items are in progress. You can check their [YouTrack tickets](#) for updates.

Basic syntax

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() {
//sampleStart
    print("Hello ")
    print("world!")
//sampleEnd
}
```

println prints its arguments and adds a line break, so that the next thing you print appears on the next line:

```
fun main() {
//sampleStart
    println("Hello world!")
    println(42)
//sampleEnd
}
```

Functions

A function with two Int parameters and Int return type:

```
//sampleStart
fun sum(a: Int, b: Int): Int {
    return a + b
}
//sampleEnd

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:

```
//sampleStart
fun sum(a: Int, b: Int) = a + b
//sampleEnd

fun main() {
    println("sum of 19 and 23 is ${sum(19, 23)}")
}
```

A function that returns no meaningful value:

```
//sampleStart
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

```
//sampleEnd

fun main() {
    printSum(-1, 8)
}
```

Unit return type can be omitted:

```
//sampleStart
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
//sampleEnd

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() {
//sampleStart
    // Declares the variable x and initializes it with the value of 5
    val x: Int = 5
    // 5
//sampleEnd
    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() {
//sampleStart
    // 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
//sampleEnd
    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() {
//sampleStart
    // Declares the variable x with the value of 5; `Int` type is inferred
    val x = 5
    // 5
//sampleEnd
    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() {
//sampleStart
    // 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
//sampleEnd
    println(x)
    println(c)
}

```

You can declare variables at the top level:

```

//sampleStart
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
// x = 0; PI = 3.14
// incrementX()
// x = 1; PI = 3.14
//sampleEnd

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() {  
    //sampleStart  
    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"  
    //sampleEnd  
    println(s2)  
}
```

See [String templates](#) for details.

Conditional expressions

```
//sampleStart  
fun maxOf(a: Int, b: Int): Int {  
    if (a > b) {  
        return a  
    } else {  
        return b  
    }  
}  
//sampleEnd  
  
fun main() {  
    println("max of 0 and 42 is ${maxOf(0, 42)})  
}
```

In Kotlin, if can also be used as an expression:

```
//sampleStart  
fun maxOf(a: Int, b: Int) = if (a > b) a else b  
//sampleEnd  
  
fun main() {  
    println("max of 0 and 42 is ${maxOf(0, 42)})  
}
```

See [if-expressions](#).

for loop

```
fun main() {  
    //sampleStart  
    val items = listOf("apple", "banana", "kiwifruit")
```

```

    for (item in items) {
        println(item)
    }
//sampleEnd
}

```

or:

```

fun main() {
//sampleStart
    val items = listOf("apple", "banana", "kiwifruit")
    for (index in items.indices) {
        println("item at $index is ${items[index]}")
    }
//sampleEnd
}

```

See [for loop](#).

while loop

```

fun main() {
//sampleStart
    val items = listOf("apple", "banana", "kiwifruit")
    var index = 0
    while (index < items.size) {
        println("item at $index is ${items[index]}")
        index++
    }
//sampleEnd
}

```

See [while loop](#).

when expression

```

//sampleStart
fun describe(obj: Any): String =
    when (obj) {
        1           -> "One"
        "Hello"     -> "Greeting"
        is Long      -> "Long"
        !is String   -> "Not a string"
        else         -> "Unknown"
    }
//sampleEnd

fun main() {
    println(describe(1))
    println(describe("Hello"))
    println(describe(1000L))
    println(describe(2))
    println(describe("other"))
}

```

See [when expression](#).

Ranges

Check if a number is within a range using in operator:

```

fun main() {
//sampleStart
    val x = 10
    val y = 9
    if (x in 1..y+1) {
        println("fits in range")
    }
}

```

```
    }
    //sampleEnd
}
```

Check if a number is out of range:

```
fun main() {
//sampleStart
    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")
    }
//sampleEnd
}
```

Iterate over a range:

```
fun main() {
//sampleStart
    for (x in 1..5) {
        print(x)
    }
//sampleEnd
}
```

Or over a progression:

```
fun main() {
//sampleStart
    for (x in 1..10 step 2) {
        print(x)
    }
    println()
    for (x in 9 downTo 0 step 3) {
        print(x)
    }
//sampleEnd
}
```

See [Ranges and progressions](#).

Collections

Iterate over a collection:

```
fun main() {
    val items = listOf("apple", "banana", "kiwifruit")
//sampleStart
    for (item in items) {
        println(item)
    }
//sampleEnd
}
```

Check if a collection contains an object using in operator:

```
fun main() {
    val items = setOf("apple", "banana", "kiwifruit")
//sampleStart
    when {
        "orange" in items -> println("juicy")
        "apple" in items -> println("apple is fine too")
    }
//sampleEnd
}
```

Use [lambda expressions](#) to filter and map collections:

```
fun main() {
//sampleStart
    val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
    fruits
        .filter { it.startsWith("a") }
        .sortedBy { it }
        .map { it.uppercase() }
        .forEach { println(it) }
//sampleEnd
}
```

See [Collections overview](#).

Nullable values and null checks

A reference must be explicitly marked as nullable when null value is possible. Nullable type names have ? at the end.

Return null if str does not hold an integer:

```
fun parseInt(str: String): Int? {
    // ...
}
```

Use a function returning nullable value:

```
fun parseInt(str: String): Int? {
    return str.toIntOrNull()
}

//sampleStart
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")
    }
}
//sampleEnd

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)

//sampleStart
    // ...
    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)
//sampleEnd
}

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:

```

//sampleStart
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
}
//sampleEnd

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())
}

```

or:

```

//sampleStart
fun getStringLength(obj: Any): Int? {
    if (obj !is String) return null

    // `obj` is automatically cast to `String` in this branch
    return obj.length
}
//sampleEnd

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())
}

```

or even:

```

//sampleStart
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
}
//sampleEnd

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](#).

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 vars) 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](#).

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..  
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 fileSize = files?.size ?: run {
    val someSize = getSomeSize()
    someSize * 2
}
```

```
println(filesSize)
```

Execute a statement 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 = 0xFADADA
}
```

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](#).

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.

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 an upper camel case with an uppercase first letter (also known as Pascal case), 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:

- jvmMain/kotlin/Platform.jvm.kt
- androidMain/kotlin/Platform.android.kt
- iosMain/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 cases when 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 the camel case (org.example.myProject).
- Names of classes and objects start with an uppercase letter and use the camel case:

```
open class DeclarationProcessor { /*...*/ }

object EmptyDeclarationProcessor : DeclarationProcessor() { /*...*/ }
```

Function names

Names of functions, properties and local variables start with a lowercase letter and use the camel case and no underscores:

```
fun processDeclarations() { /*...*/ }
var declarationCount = 1
```

Exception: factory functions used to create instances of classes can have the same name as the abstract return type:

```
interface Foo { /*...*/ }

class FooImpl : Foo { /*...*/ }

fun Foo(): Foo { return FooImpl() }
```

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 uppercase underscore-separated ([screaming snake case](#)) names:

```
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 uppercase underscore-separated names ([screaming snake case](#)) (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, capitalize it if it consists of two letters (IOStream); capitalize only the first letter if it is longer (XmlFormatter, 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 cases:

- 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().getOrDefaultCharsetForPropertiesFiles(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 + z
}
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, car) in cars) {
        meanValue += car.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")
```

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.

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..
```

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() {
    //sampleStart
    println("""
        Not
        trimmed
        text
    """)
}

println("""
    Trimmed
    text
    """.trimIndent())
}

println()

val a = """Trimmed to margin text:
    |if(a > 1) {
    |    return a
    |}""".trimMargin()

println(a)
//sampleEnd
}
```

Learn the difference between [Java and Kotlin multiline strings](#).

Functions vs properties

In some cases, 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 with default argument 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 [library creators' guidelines](#).

Basic types

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, booleans and others), 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](#)

[Learn how to perform type checks and casts in Kotlin.](#)

Numbers

Integer types

Kotlin provides a set of built-in types that represent numbers.

For integer numbers, there are four types with different sizes and, hence, value ranges:

Type	Size (bits)	Min value	Max value
Byte	8	-128	127
Short	16	-32768	32767
Int	32	-2,147,483,648 (-2 ³¹)	2,147,483,647 (2 ³¹ - 1)
Long	64	-9,223,372,036,854,775,808 (-2 ⁶³)	9,223,372,036,854,775,807 (2 ⁶³ - 1)

When you initialize a variable with no explicit type specification, the compiler automatically infers the type with the smallest range enough to represent the value starting from Int. If it is not exceeding the range of Int, the type is Int. If it exceeds, the type is Long. To specify the Long value explicitly, append the suffix L to the value. Explicit type specification triggers the compiler to check the value not to exceed the range of the specified type.

```
val one = 1 // Int
val threeBillion = 3000000000 // Long
val oneLong = 1L // Long
val oneByte: Byte = 1
```

In addition to integer types, Kotlin also provides unsigned integer types. For more information, see [Unsigned integer types](#).

Floating-point types

For real numbers, Kotlin provides floating-point types `Float` and `Double` that adhere to the [IEEE 754 standard](#). `Float` reflects the IEEE 754 single precision, while `Double` reflects double precision.

These types differ in their size and provide storage for floating-point numbers with different precision:

Type	Size (bits)	Significant bits	Exponent bits	Decimal digits
Float	32	24	8	6-7
Double	64	53	11	15-16

You can initialize `Double` and `Float` variables with numbers having a fractional part. It's separated from the integer part by a period (.) For variables initialized with fractional numbers, the compiler infers the `Double` type:

```
val pi = 3.14 // Double
// val one: Double = 1 // Error: type mismatch
val oneDouble = 1.0 // Double
```

To explicitly specify the `Float` type for a value, add the suffix `f` or `F`. If such a value contains more than 6-7 decimal digits, it will be rounded:

```
val e = 2.7182818284 // Double
val eFloat = 2.7182818284f // Float, actual value is 2.7182817
```

Unlike some other languages, there are no implicit widening conversions for numbers in Kotlin. For example, a function with a `Double` parameter can be called only on `Double` values, but not `Float`, `Int`, or other numeric values:

```
fun main() {
    fun printDouble(d: Double) { print(d) }

    val i = 1
    val d = 1.0
    val f = 1.0f

    printDouble(d)
    // printDouble(i) // Error: Type mismatch
    // printDouble(f) // Error: Type mismatch
}
```

To convert numeric values to different types, use [explicit conversions](#).

Literal constants for numbers

There are the following kinds of literal constants for integral values:

- Decimals: 123
- Longs are tagged by a capital L: 123L
- Hexadecimals: 0x0F
- Binaries: 0b00001011

Octal literals are not supported in Kotlin.

Kotlin also supports a conventional notation for floating-point numbers:

- Doubles by default: 123.5, 123.5e10
- Floats are tagged by f or F: 123.5f

You can use underscores to make number constants more readable:

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

There are also special tags for unsigned integer literals.

Read more about [literals for unsigned integer types](#).

Numbers representation on the JVM

On the JVM platform, numbers are stored as primitive types: int, double, and so on. Exceptions are cases when you create a nullable number reference such as Int? or use generics. In these cases numbers are boxed in Java classes Integer, Double, and so on.

Nullable references to the same number can refer to different objects:

```
fun main() {
//sampleStart
    val a: Int = 100
    val boxedA: Int? = a
    val anotherBoxedA: Int? = a

    val b: Int = 10000
    val boxedB: Int? = b
    val anotherBoxedB: Int? = b

    println(boxedA === anotherBoxedA) // true
    println(boxedB === anotherBoxedB) // false
//sampleEnd
}
```

All nullable references to a are actually the same object because of the memory optimization that JVM applies to Integers between -128 and 127. It doesn't apply to the b references, so they are different objects.

On the other hand, they are still equal:

```
fun main() {
//sampleStart
    val b: Int = 10000
    println(b == b) // Prints 'true'
    val boxedB: Int? = b
    val anotherBoxedB: Int? = b
    println(boxedB == anotherBoxedB) // Prints 'true'
//sampleEnd
}
```

Explicit number conversions

Due to different representations, smaller types are not subtypes of bigger ones. If they were, we would have troubles of the following sort:

```
// Hypothetical code, does not actually compile:
val a: Int? = 1 // A boxed Int (java.lang.Integer)
val b: Long? = a // Implicit conversion yields a boxed Long (java.lang.Long)
print(b == a) // Surprise! This prints "false" as Long's equals() checks whether the other is Long as well
```

So equality would have been lost silently, not to mention identity.

As a consequence, smaller types are NOT implicitly converted to bigger types. This means that assigning a value of type Byte to an Int variable requires an explicit conversion:

```
val b: Byte = 1 // OK, literals are checked statically
// val i: Int = b // ERROR
```

```
val i1: Int = b.toInt()
```

All number types support conversions to other types:

- `toByte(): Byte`
- `toShort(): Short`
- `toInt(): Int`
- `toLong(): Long`
- `toFloat(): Float`
- `toDouble(): Double`

In many cases, there is no need for explicit conversions because the type is inferred from the context, and arithmetical operations are overloaded for appropriate conversions, for example:

```
val l = 1L + 3 // Long + Int => Long
```

Operations on numbers

Kotlin supports the standard set of arithmetical operations over numbers: `+`, `-`, `*`, `/`, `%`. They are declared as members of appropriate classes:

```
fun main() {  
    //sampleStart  
    println(1 + 2)  
    println(2_500_000_000L - 1L)  
    println(3.14 * 2.71)  
    println(10.0 / 3)  
    //sampleEnd  
}
```

You can also override these operators for custom classes. See [Operator overloading](#) for details.

Division of integers

Division between integers numbers always returns an integer number. Any fractional part is discarded.

```
fun main() {  
    //sampleStart  
    val x = 5 / 2  
    //println(x == 2.5) // ERROR: Operator '==' cannot be applied to 'Int' and 'Double'  
    println(x == 2)  
    //sampleEnd  
}
```

This is true for a division between any two integer types:

```
fun main() {  
    //sampleStart  
    val x = 5L / 2  
    println(x == 2L)  
    //sampleEnd  
}
```

To return a floating-point type, explicitly convert one of the arguments to a floating-point type:

```
fun main() {  
    //sampleStart  
    val x = 5 / 2.toDouble()  
    println(x == 2.5)  
    //sampleEnd  
}
```

Bitwise operations

Kotlin provides a set of bitwise operations on integer numbers. They operate on the binary level directly with bits of the numbers' representation. Bitwise operations are represented by functions that can be called in infix form. They can be applied only to Int and Long:

```
val x = (1 shl 2) and 0x000FF000
```

Here is the complete list of bitwise operations:

- shl(bits) – signed shift left
- shr(bits) – signed shift right
- ushr(bits) – unsigned shift right
- and(bits) – bitwise AND
- or(bits) – bitwise OR
- xor(bits) – bitwise XOR
- inv() – bitwise inversion

Floating-point numbers comparison

The operations on floating-point numbers discussed in this section are:

- Equality checks: a == b and a != b
- Comparison operators: a < b, a > b, a <= b, a >= b
- Range instantiation and range checks: a..b, x in a..b, x !in a..b

When the operands a and b are statically known to be Float or Double or their nullable counterparts (the type is declared or inferred or is a result of a [smart cast](#)), the operations on the numbers and the range that they form follow the [IEEE 754 Standard for Floating-Point Arithmetic](#).

However, to support generic use cases and provide total ordering, the behavior is different for operands that are not statically typed as floating-point numbers. For example, Any, Comparable<...>, or Collection<T> types. In this case, the operations use 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

Here is an example that shows the difference in behavior between operands statically typed as floating-point numbers (Double.NaN) and operands not statically typed as floating-point numbers (listOf(T)).

```
fun main() {  
    //sampleStart  
    // Operand statically typed as floating-point number  
    println(Double.NaN == Double.NaN) // false  
    // Operand NOT statically typed as floating-point number  
    // So NaN is equal to itself  
    println(listOf(Double.NaN) == listOf(Double.NaN)) // true  
  
    // Operand statically typed as floating-point number  
    println(0.0 == -0.0) // true  
    // Operand NOT statically typed as floating-point number  
    // So -0.0 is less than 0.0  
    println(listOf(0.0) == listOf(-0.0)) // false  
  
    println(listOf(Double.NaN, Double.POSITIVE_INFINITY, 0.0, -0.0).sorted())  
    // [-0.0, 0.0, Infinity, NaN]  
    //sampleEnd  
}
```

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 ($2^{32} - 1$)
ULong	64	0	18,446,744,073,709,551,615 ($2^{64} - 1$)

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, Kotlin provides an ability to tag an integer literal with a suffix indicating a specific unsigned type (similarly to Float or Long):

- u and U tag is for unsigned literals. The exact type is determined based on the expected type. If no expected type is provided, compiler will use UInt or ULong depending on the size of 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 tag literal as unsigned long:

```
val a = 1UL // UInt, even though no expected type provided and 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 type `Boolean` represents boolean objects that can have two values: `true` and `false`. `Boolean` has a `nullable` counterpart declared as `Boolean?`.

On the JVM, booleans stored as the primitive boolean type typically use 8 bits.

Built-in operations on booleans include:

- `||` – disjunction (logical OR)
- `&&` – conjunction (logical AND)
- `!` – negation (logical NOT)

For example:

```
fun main() {
//sampleStart
    val myTrue: Boolean = true
    val myFalse: Boolean = false
    val boolNull: Boolean? = null

    println(myTrue || myFalse)
    // true
    println(myTrue && myFalse)
    // false
    println(!myTrue)
    // false
}
```

```
    println(boolNull)
    // null
//sampleEnd
}
```

The || and && operators work lazily, which means:

- If the first operand is true, the || operator does not evaluate the second operand.
- If the first operand is false, the && operator does not evaluate the second operand.

On the JVM, nullable references to boolean objects are boxed in Java classes, just like with [numbers](#).

Characters

Characters are represented by the type Char. Character literals go in single quotes: '1'.

On the JVM, a character stored as primitive type: char, represents a 16-bit Unicode character.

Special characters start from an escaping backslash \. The following escape sequences are supported:

- \t – tab
- \b – backspace
- \n – new line (LF)
- \r – carriage return (CR)
- \' – single quotation mark
- \" – double quotation mark
- \\ – backslash
- \\$ – dollar sign

To encode any other character, use the Unicode escape sequence syntax: '\uFF00'.

```
fun main() {
//sampleStart
    val aChar: Char = 'a'

    println(aChar)
    println('\n') // Prints an extra newline character
    println('\uFF00')
//sampleEnd
}
```

If a value of character variable is a digit, you can explicitly convert it to an Int number using the [digitToInt\(\)](#) function.

On the JVM, characters are boxed in Java classes when a nullable reference is needed, just like with [numbers](#). Identity is not preserved by the boxing operation.

Strings

Strings in Kotlin are represented by the type [String](#).

On the JVM, an object of String type in UTF-16 encoding uses approximately 2 bytes per character.

Generally, a string value is a sequence of characters in double quotes ("):

```
val str = "abcd 123"
```

Elements of a string are characters that you can access via the indexing operation: `s[i]`. You can iterate over these characters with a for loop:

```
fun main() {
    val str = "abcd"
    //sampleStart
    for (c in str) {
        println(c)
    }
    //sampleEnd
}
```

Strings are immutable. Once you initialize a string, you can't change its value or assign a new value to it. All operations that transform strings return their results in a new String object, leaving the original string unchanged:

```
fun main() {
    //sampleStart
    val str = "abcd"

    // Creates and prints a new String object
    println(str.uppercase())
    // ABCD

    // The original string remains the same
    println(str)
    // abcd
    //sampleEnd
}
```

To concatenate strings, use the `+` operator. This also works for concatenating strings with values of other types, as long as the first element in the expression is a string:

```
fun main() {
    //sampleStart
    val s = "abc" + 1
    println(s + "def")
    // abc1def
    //sampleEnd
}
```

In most cases using [string templates](#) or [multiline strings](#) is preferable to string concatenation.

String literals

Kotlin has two types of string literals:

- [Escaped strings](#)
- [Multiline strings](#)

Escaped strings

Escaped strings can contain escaped characters.

Here's an example of an escaped string:

```
val s = "Hello, world!\n"
```

Escaping is done in the conventional way, with a backslash (\).

See [Characters](#) page for the list of supported escape sequences.

Multiline strings

Multiline strings can contain newlines and arbitrary text. It is delimited by a triple quote ("""), contains no escaping and can contain newlines and any other characters:

```
val text = """
    for (c in "foo")
        print(c)
"""

```

To remove leading whitespace from multiline strings, use the [trimMargin\(\)](#) function:

```
val text = """
|Tell me and I forget.
|Teach me and I remember.
|Involve me and I learn.
|(Benjamin Franklin)
""".trimMargin()
```

By default, a pipe symbol | is used as margin prefix, but you can choose another character and pass it as a parameter, like trimMargin(">").

String templates

String literals may contain template expressions – pieces of code that are evaluated and whose results are concatenated into a string. When a template expression is processed, Kotlin automatically calls the `.toString()` function on the expression's result to convert it into a string. A template expression starts with a dollar sign (\$) and consists of either a variable name:

```
fun main() {
//sampleStart
    val i = 10
    println("i = $i")
    // i = 10

    val letters = listOf("a", "b", "c", "d", "e")
    println("Letters: $letters")
    // Letters: [a, b, c, d, e]

//sampleEnd
}
```

or an expression in curly braces:

```
fun main() {
//sampleStart
    val s = "abc"
    println("${s.length} is ${s.length}")
    // abc.length is 3
//sampleEnd
}
```

You can use templates both in multiline and escaped strings. To insert the dollar sign \$ in a multiline string (which doesn't support backslash escaping) before any symbol, which is allowed as a beginning of an [identifier](#), use the following syntax:

```
val price = """
${'$'}_9.99
"""

```

String formatting

String formatting with the `String.format()` function is only available in Kotlin/JVM.

To format a string to your specific requirements, use the [String.format\(\)](#) function.

The `String.format()` function accepts a format string and one or more arguments. The format string contains one placeholder (indicated by %) for a given argument, followed by format specifiers. Format specifiers are formatting instructions for the respective argument, consisting of flags, width, precision, and conversion type. Collectively, format specifiers shape the output's formatting. Common format specifiers include `%d` for integers, `%f` for floating-point numbers, and `%s` for strings. You can also use the `argument_index$` syntax to reference the same argument multiple times within the format string in different formats.

For a detailed understanding and an extensive list of format specifiers, see [Java's Class Formatter documentation](#).

Let's look at an example:

```
fun main() {
    //sampleStart
    // Formats an integer, adding leading zeroes to reach a length of seven characters
    val integerNumber = String.format("%07d", 31416)
    println(integerNumber)
    // 0031416

    // Formats a floating-point number to display with a + sign and four decimal places
    val floatNumber = String.format("%+.4f", 3.141592)
    println(floatNumber)
    // +3.1416

    // Formats two strings to uppercase, each taking one placeholder
    val helloString = String.format("%S %S", "hello", "world")
    println(helloString)
    // HELLO WORLD

    // Formats a negative number to be enclosed in parentheses, then repeats the same number in a different format (without
    // parentheses) using `argument_index$`.
    val negativeNumberInParentheses = String.format("(%d means %1\$d", -31416)
    println(negativeNumberInParentheses)
    //(-31416) means -31416
    //sampleEnd
}
```

The `String.format()` function provides similar functionality to string templates. However, the `String.format()` function is more versatile because there are more formatting options available.

In addition, you can assign the format string from a variable. This can be useful when the format string changes, for example, in localization cases that depend on the user locale.

Be careful when using the `String.format()` function because it can be easy to mismatch the number or position of the arguments with their corresponding placeholders.

Arrays

An array is a data structure that holds a fixed number of values of the same type or its subtypes. The most common type of array in Kotlin is the object-type array, represented by the `Array` class.

If you use primitives in an object-type array, this has a performance impact because your primitives are boxed into objects. To avoid boxing overhead, use primitive-type arrays instead.

When to use arrays

Use arrays in Kotlin when you have specialized low-level requirements that you need to meet. For example, if you have performance requirements beyond what is needed for regular applications, or you need to build custom data structures. If you don't have these sorts of restrictions, use `collections` instead.

Collections have the following benefits compared to arrays:

- Collections can be read-only, which gives you more control and allows you to write robust code that has a clear intent.
- It is easy to add or remove elements from collections. In comparison, arrays are fixed in size. The only way to add or remove elements from an array is to create a new array each time, which is very inefficient:

```

fun main() {
    //sampleStart
    var riversArray = arrayOf("Nile", "Amazon", "Yangtze")

    // Using the += assignment operation creates a new riversArray,
    // copies over the original elements and adds "Mississippi"
    riversArray += "Mississippi"
    println(riversArray.joinToString())
    // Nile, Amazon, Yangtze, Mississippi
    //sampleEnd
}

```

- You can use the equality operator (==) to check if collections are structurally equal. You can't use this operator for arrays. Instead, you have to use a special function, which you can read more about in [Compare arrays](#).

For more information about collections, see [Collections overview](#).

Create arrays

To create arrays in Kotlin, you can use:

- functions, such as `arrayOf()`, `arrayOfNulls()` or `emptyArray()`.
- the `Array` constructor.

This example uses the `arrayOf()` function and passes item values to it:

```

fun main() {
    //sampleStart
    // Creates an array with values [1, 2, 3]
    val simpleArray = arrayOf(1, 2, 3)
    println(simpleArray.joinToString())
    // 1, 2, 3
    //sampleEnd
}

```

This example uses the `arrayOfNulls()` function to create an array of a given size filled with null elements:

```

fun main() {
    //sampleStart
    // Creates an array with values [null, null, null]
    val nullArray: Array<Int?> = arrayOfNulls(3)
    println(nullArray.joinToString())
    // null, null, null
    //sampleEnd
}

```

This example uses the `emptyArray()` function to create an empty array :

```
var exampleArray = emptyArray<String>()
```

You can specify the type of the empty array on the left-hand or right-hand side of the assignment due to Kotlin's type inference.

For example:

```

var exampleArray = emptyArray<String>()

var exampleArray: Array<String> = emptyArray()

```

The `Array` constructor takes the array size and a function that returns values for array elements given its index:

```

fun main() {
    //sampleStart
    // Creates an Array<Int> that initializes with zeros [0, 0, 0]
    val initArray = Array<Int>(3) { 0 }
}

```

```

println(initArray.joinToString())
// 0, 0, 0

// Creates an Array<String> with values ["0", "1", "4", "9", "16"]
val asc = Array(5) { i -> (i * i).toString() }
asc.forEach { print(it) }
// 014916
//sampleEnd
}

```

Like in most programming languages, indices start from 0 in Kotlin.

Nested arrays

Arrays can be nested within each other to create multidimensional arrays:

```

fun main() {
//sampleStart
    // 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]]]
//sampleEnd
}

```

Nested arrays don't have to be the same type or the same size.

Access and modify elements

Arrays are always mutable. To access and modify elements in an array, use the [indexed access operator](#) []:

```

fun main() {
//sampleStart
    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].toString()) // 10
    println(twoDArray[0][0].toString()) // 2
//sampleEnd
}

```

Arrays in Kotlin are invariant. This means that Kotlin doesn't allow you to assign an `Array<String>` to an `Array<Any>` to prevent a possible runtime failure. Instead, you can use `Array<out Any>`. For more information, see [Type Projections](#).

Work with arrays

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. For example, comparing arrays, transforming their contents or converting them to collections.

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 \(varargs\)](#).

Compare arrays

To compare whether two arrays have the same elements in the same order, use the `.contentEquals()` and `.contentDeepEquals()` functions:

```
fun main() {
//sampleStart
    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
//sampleEnd
}
```

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 document highlights a few but this isn't an exhaustive list. For the full list of functions, see our [API reference](#).

Sum

To return the sum of all elements in an array, use the `.sum()` function:

```
fun main() {
//sampleStart
    val sumArray = arrayOf(1, 2, 3)

    // Sums array elements
    println(sumArray.sum())
    // 6
//sampleEnd
}
```

The `.sum()` function can only be used with arrays of [numeric data types](#), such as Int.

Shuffle

To randomly shuffle the elements in an array, use the `.shuffle()` function:

```
fun main() {
//sampleStart
    val simpleArray = arrayOf(1, 2, 3)

    // Shuffles elements [3, 2, 1]
    simpleArray.shuffle()
    println(simpleArray.joinToString())

    // Shuffles elements again [2, 3, 1]
    simpleArray.shuffle()
    println(simpleArray.joinToString())
//sampleEnd
}
```

Convert arrays to collections

If you work with different APIs where some use arrays and some use collections, then you can convert your arrays to [collections](#) and vice versa.

Convert to List or Set

To convert an array to a List or Set, use the `.toList()` and `.toSet()` functions.

```
fun main() {
//sampleStart
    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]
//sampleEnd
}
```

Convert to Map

To convert an array to a Map, use the `.toMap()` function.

Only an array of `Pair<K,V>` can be converted to a Map. The first value of a Pair instance becomes a key, and the second becomes a value. This example uses the [infix notation](#) to call the `to` function to create tuples of Pair:

```
fun main() {
//sampleStart
    val pairArray = arrayOf("apple" to 120, "banana" to 150, "cherry" to 90, "apple" to 140)

    // Converts to a Map
    // The keys are fruits and the values are their number of calories
    // Note how keys must be unique, so the latest value of "apple"
    // overwrites the first
    println(pairArray.toMap())
    // {apple=140, banana=150, cherry=90}

//sampleEnd
}
```

Primitive-type arrays

If you use the `Array` class with primitive values, these values are boxed into objects. As an alternative, you can use primitive-type arrays, which allow you to store primitives in an array without the side effect of boxing overhead:

Primitive-type array Equivalent in Java

Primitive-type array Equivalent in Java

BooleanArray boolean[]

ByteArray byte[]

CharArray char[]

DoubleArray double[]

FloatArray float[]

IntArray int[]

LongArray long[]

ShortArray short[]

These classes have no inheritance relation to the Array class, but they have the same set of functions and properties.

This example creates an instance of the IntArray class:

```
fun main() {
    //sampleStart
    // Creates an array of Int of size 5 with the values initialized to zero
    val exampleArray = IntArray(5)
    println(exampleArray.joinToString())
    // 0, 0, 0, 0, 0
    //sampleEnd
}
```

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.

What's next?

- To learn more about why we recommend using collections for most use cases, read our [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 perform type checks to check the type of an object at runtime. Type casts enable you to convert objects to a different type.

To learn specifically about generics type checks and casts, for example `List<T>`, `Map<K,V>`, see [Generics type checks and casts](#).

is and !is operators

To perform a runtime check that identifies whether an object conforms to a given type, use the `is` operator or its negated form `!is`:

```
if (obj is String) {
    print(obj.length)
}

if (obj !is String) { // Same as !(obj is String)
    print("Not a String")
} else {
    print(obj.length)
}
```

Smart casts

In most cases, you don't need to use explicit cast operators because the compiler automatically casts objects for you. This is called smart-casting. The compiler tracks the type checks and [explicit casts](#) for immutable values and inserts implicit (safe) casts automatically when necessary:

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x is automatically cast to String
    }
}
```

The compiler is even smart enough to know that a cast is safe if a negative check leads to a return:

```
if (x !is String) return

print(x.length) // x is automatically cast to String
```

Control flow

Smart casts work not only for if conditional expressions but also for [when expressions](#) and [while loops](#):

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is IntArray -> print(x.sum())
}
```

If you declare a variable of Boolean type 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) {
        // 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 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](#).

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 change makes your code safer as the compiler tracks whether your object has a nullable type. For example:

```
//sampleStart
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
    }
}
//sampleEnd
fun main() {
    testString()
}
```

Smart cast prerequisites

Note that smart casts work only when the compiler can guarantee that the variable won't change between the check and its usage.

Smart casts can be used in the following conditions:

val local Always, except [local delegated properties](#).
variables

val 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
properties open properties or properties that have custom getters.

var local 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.
variables

var Never, because the variable can be modified at any time by other code.
properties

"Unsafe" cast operator

To explicitly cast an object to a non-nullable type, use the unsafe cast operator as:

```
val x: String = y as String
```

If the cast isn't possible, the compiler throws an exception. This is why it's called unsafe.

In the previous example, if `y` is null, the code above also throws an exception. This is because null can't be cast to String, as null isn't nullable. To make the example work for possible null values, use a nullable type on the right-hand side of the cast:

```
val x: String? = y as String?
```

"Safe" (nullable) cast operator

To avoid exceptions, use the safe cast operator `as?`, which returns null on failure.

```
val x: String? = y as? String
```

Note that despite the fact that the right-hand side of `as?` is a non-nullable type String, the result of the cast is nullable.

Conditions and loops

If expression

In Kotlin, `if` is an expression: it returns a value. Therefore, there is no ternary operator (`condition ? then : else`) because ordinary `if` works fine in this role.

```
fun main() {
    val a = 2
    val b = 3

    //sampleStart
    var max = a
    if (a < b) max = b

    // With else
    if (a > b) {
        max = a
    } else {
        max = b
    }

    // As expression
    max = if (a > b) a else b

    // You can also use `else if` in expressions:
    val maxLimit = 1
    val maxOrLimit = if (maxLimit > a) maxLimit else if (a > b) a else b

    //sampleEnd
    println("max is $max")
    println("maxOrLimit is $maxOrLimit")
}
```

Branches of an `if` expression can be blocks. In this case, the last expression is the value of a block:

```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

If you're using `if` as an expression, for example, for returning its value or assigning it to a variable, the `else` branch is mandatory.

When expression

`when` defines a conditional expression with multiple branches. It is similar to the `switch` statement in C-like languages. Its simple form looks like this.

```
when (x) {
```

```

1 -> print("x == 1")
2 -> print("x == 2")
else -> {
    print("x is neither 1 nor 2")
}
}

```

when matches its argument against all branches sequentially until some branch condition is satisfied.

when can be used either as an expression or as a statement. If it is used as an expression, the value of the first matching branch becomes the value of the overall expression. If it is used as a statement, the values of individual branches are ignored. Just like with if, each branch can be a block, and its value is the value of the last expression in the block.

The else branch is evaluated if none of the other branch conditions are satisfied.

If when is used as an expression, the else branch is mandatory, unless the compiler can prove that all possible cases are covered with branch conditions, for example, with [enum class](#) entries and [sealed class](#) subtypes).

```

enum class Bit {
    ZERO, ONE
}

val numericValue = when (getRandomBit()) {
    Bit.ZERO -> 0
    Bit.ONE -> 1
    // 'else' is not required because all cases are covered
}

```

In when statements, the else branch is mandatory in the following conditions:

- when has a subject of a Boolean, [enum](#), or [sealed](#) type, or their nullable counterparts.
- branches of when don't cover all possible cases for this subject.

```

enum class Color {
    RED, GREEN, BLUE
}

when (getColor()) {
    Color.RED -> println("red")
    Color.GREEN -> println("green")
    Color.BLUE -> println("blue")
    // 'else' is not required because all cases are covered
}

when (getColor()) {
    Color.RED -> println("red") // no branches for GREEN and BLUE
    else -> println("not red") // 'else' is required
}

```

To define a common behavior for multiple cases, combine their conditions in a single line with a comma:

```

when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}

```

You can use arbitrary expressions (not only constants) as branch conditions

```

when (x) {
    s.toInt() -> print("s encodes x")
    else -> print("s does not encode x")
}

```

You can also check a value for being in or !in a [range](#) or a collection:

```

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")
}

```

```
}
```

Another option is checking that a value is or !is of a particular type. Note that, due to [smart casts](#), you can access the methods and properties of the type without any extra checks.

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

when can also be used as a replacement for an if-else if chain. If no argument is supplied, the branch conditions are simply boolean expressions, and a branch is executed when its condition is true:

```
when {
    x.isOdd() -> print("x is odd")
    y.isEven() -> print("y is even")
    else -> print("x+y is odd")
}
```

You can capture when subject in a variable using following syntax:

```
fun Request.getBody() =
    when (val response = executeRequest()) {
        is Success -> response.body
        is HttpError -> throw HttpException(response.status)
    }
```

The scope of variable introduced in when subject is restricted to the body of this when.

For loops

The for loop iterates through anything that provides an iterator. This is equivalent to the foreach loop in languages like C#. The syntax for is the following:

```
for (item in collection) print(item)
```

The body of for can be a block.

```
for (item: Int in ints) {
    // ...
}
```

As mentioned before, for iterates through anything that provides an iterator. This means that it:

- has a member or an extension function iterator() that returns Iterator<>, which:
 - has a member or an extension function next()
 - has a member or an extension function hasNext() that returns Boolean.

All of these three functions need to be marked as operator.

To iterate over a range of numbers, use a [range expression](#):

```
fun main() {
//sampleStart
    for (i in 1..3) {
        println(i)
    }
    for (i in 6 downTo 0 step 2) {
        println(i)
    }
//sampleEnd
}
```

A for loop over a range or an array is compiled to an index-based loop that does not create an iterator object.

If you want to iterate through an array or a list with an index, you can do it this way:

```
fun main() {
    val array = arrayOf("a", "b", "c")
    //sampleStart
    for (i in array.indices) {
        println(array[i])
    }
    //sampleEnd
}
```

Alternatively, you can use the `withIndex` library function:

```
fun main() {
    val array = arrayOf("a", "b", "c")
    //sampleStart
    for ((index, value) in array.withIndex()) {
        println("the element at $index is $value")
    }
    //sampleEnd
}
```

While loops

while and do-while loops execute their body continuously while their condition is satisfied. The difference between them is the condition checking time:

- while checks the condition and, if it's satisfied, executes the body and then returns to the condition check.
- do-while executes the body and then checks the condition. If it's satisfied, the loop repeats. So, the body of do-while executes at least once regardless of the condition.

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

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, we 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.

Return to labels

In Kotlin, functions can be nested using function literals, local functions, and object expressions. Qualified returns allow us to return from an outer function. The most important use case is returning from a lambda expression. Recall that when we write the following, the return-expression returns from the nearest enclosing function - foo:

```
//sampleStart  
fun foo() {  
    listOf(1, 2, 3, 4, 5).forEach {  
        if (it == 3) return // non-local return directly to the caller of foo()  
        print(it)  
    }  
    println("this point is unreachable")  
}  
//sampleEnd  
  
fun main() {  
    foo()  
}
```

Note that such non-local returns are supported only for lambda expressions passed to [inline functions](#). To return from a lambda expression, label it and qualify the return:

```
//sampleStart  
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)  
    }  
    println(" done with explicit label")  
}  
//sampleEnd  
  
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.

```
//sampleStart  
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)  
    }  
    println(" done with implicit label")  
}  
//sampleEnd  
  
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.

```
//sampleStart
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach { 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")
}
//sampleEnd

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 another nesting lambda and non-locally returning from it:

```
//sampleStart
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")
}
//sampleEnd

fun main() {
    foo()
}
```

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)".

Exceptions

Exception classes

All exception classes in Kotlin inherit the `Throwable` class. Every exception has a message, a stack trace, and an optional cause.

To throw an exception object, use the `throw` expression:

```
fun main() {
//sampleStart
    throw Exception("Hi There!")
//sampleEnd
}
```

To catch an exception, use the `try...catch` expression:

```
try {
    // some code
} catch (e: SomeException) {
    // handler
} finally {
    // optional finally block
}
```

There may be zero or more catch blocks, and the finally block may be omitted. However, at least one catch or finally block is required.

Try is an expression

try is an expression, which means it can have a return value:

```
val a: Int? = try { input.toInt() } catch (e: NumberFormatException) { null }
```

The returned value of a try expression is either the last expression in the try block or the last expression in the catch block (or blocks). The contents of the finally block don't affect the result of the expression.

Checked exceptions

Kotlin does not have checked exceptions. There are many reasons for this, but we will provide a simple example that illustrates why it is the case.

The following is an example interface from the JDK implemented by the StringBuilder class:

```
Appendable append(CharSequence csq) throws IOException;
```

This signature says that every time I append a string to something (a StringBuilder, some kind of a log, a console, etc.), I have to catch the IOExceptions. Why? Because the implementation might be performing IO operations (Writer also implements Appendable). The result is code like this all over the place:

```
try {
    log.append(message)
} catch (IOException e) {
    // Must be safe
}
```

And that's not good. Just take a look at [Effective Java, 3rd Edition](#), Item 77: Don't ignore exceptions.

Bruce Eckel says this about checked exceptions:

Examination of small programs leads to the conclusion that requiring exception specifications could both enhance developer productivity and enhance code quality, but experience with large software projects suggests a different result – decreased productivity and little or no increase in code quality.

And here are some additional thoughts on the matter:

- [Java's checked exceptions were a mistake](#) (Rod Waldhoff)
- [The Trouble with Checked Exceptions](#) (Anders Hejlsberg)

If you want to alert callers about possible exceptions when calling Kotlin code from Java, Swift, or Objective-C, you can use the @Throws annotation. Read more about using this annotation [for Java](#) and [for Swift and Objective-C](#).

The Nothing type

throw is an expression in Kotlin, so you can use it, for example, as part of an Elvis expression:

```
val s = person.name ?: throw IllegalArgumentException("Name required")
```

The throw expression has the type Nothing. This type has no values and is used to mark code locations that can never be reached. In your own code, you can use Nothing to mark a function that never returns:

```
fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
}
```

When you call this function, the compiler will know that the execution doesn't continue beyond the call:

```
val s = person.name ?: fail("Name required")
println(s)      // 's' is known to be initialized at this point
```

You may also encounter this type when dealing with type inference. The nullable variant of this type, `Nothing?`, has exactly one possible value, which is null. If you use null to initialize a value of an inferred type and there's no other information that can be used to determine a more specific type, the compiler will infer the `Nothing?` type:

```
val x = null      // 'x' has type `Nothing?`
val l = listOf(null)  // 'l' has type `List<Nothing?>`
```

Java interoperability

Please see the section on exceptions in the [Java interoperability page](#) for information about Java interoperability.

Packages and imports

A source file may start with a package declaration:

```
package org.example

fun printMessage() { /*...*/ }
class Message { /*...*/ }

// ...
```

All the contents, such as classes and functions, of the source file are included in this package. So, in the example above, the full name of `printMessage()` is `org.example.printMessage`, and the full name of `Message` is `org.example.Message`.

If the package is not specified, the contents of such a file belong to the default package with no name.

Default imports

A number of packages are imported into every Kotlin file by default:

- `kotlin.*`
- `kotlin.annotation.*`
- `kotlin.collections.*`
- `kotlin.comparisons.*`
- `kotlin.io.*`
- `kotlin.ranges.*`
- `kotlin.sequences.*`
- `kotlin.text.*`

Additional packages are imported depending on the target platform:

- JVM:
 - `java.lang.*`
 - `kotlin.jvm.*`
- JS:
 - `kotlin.js.*`

Imports

Apart from the default imports, each file may contain its own import directives.

You can import either a single name:

```
import org.example.Message // Message is now accessible without qualification
```

or all the accessible contents of a scope: package, class, object, and so on:

```
import org.example.* // everything in 'org.example' becomes accessible
```

If there is a name clash, you can disambiguate by using `as` keyword to locally rename the clashing entity:

```
import org.example.Message // Message is accessible
import org.test.Message as TestMessage // TestMessage stands for 'org.test.Message'
```

The import keyword is not restricted to importing classes; you can also use it to import other declarations:

- top-level functions and properties
- functions and properties declared in [object declarations](#)
- [enum constants](#)

Visibility of top-level declarations

If a top-level declaration is marked private, it is private to the file it's declared in (see [Visibility modifiers](#)).

Classes

Classes in Kotlin are declared using the keyword `class`:

```
class Person { /*...*/ }
```

The class declaration consists of the class name, the class header (specifying its type parameters, the primary constructor, and some other things), and the class body surrounded by curly braces. Both the header and the body are optional; if the class has no body, the curly braces can be omitted.

```
class Empty
```

Constructors

A class in Kotlin has a primary constructor and possibly one or more secondary constructors. The primary constructor is declared in the class header, and it goes after the class name and optional type parameters.

```
class Person constructor(firstName: String) { /*...*/ }
```

If the primary constructor does not have any annotations or visibility modifiers, the `constructor` keyword can be omitted:

```
class Person(firstName: String) { /*...*/ }
```

The primary constructor initializes a class instance and its properties in the class header. The class header can't contain any runnable code. If you want to run some code during object creation, use initializer blocks inside the class body. Initializer blocks are declared with the `init` keyword followed by curly braces. Write any code that you want to run within the curly braces.

During the initialization of an instance, the initializer blocks are executed in the same order as they appear in the class body, interleaved with the property initializers:

```
//sampleStart
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name".also(::println)

    init {
        println("First initializer block that prints $name")
    }

    val secondProperty = "Second property: ${name.length}".also(::println)

    init {
        println("Second initializer block that prints ${name.length}")
    }
}
//sampleEnd

fun main() {
    InitOrderDemo("hello")
}
```

Primary constructor parameters can be used in the initializer blocks. They can also be used in property initializers declared in the class body:

```
class Customer(name: String) {
    val customerKey = name.uppercase()
}
```

Kotlin has a concise syntax for declaring properties and initializing them from the primary constructor:

```
class Person(val firstName: String, val lastName: String, var age: Int)
```

Such declarations can also include default values of the class properties:

```
class Person(val firstName: String, val lastName: String, var isEmployed: Boolean = true)
```

You can use a trailing comma when you declare class properties:

```
class Person(
    val firstName: String,
    val lastName: String,
    var age: Int, // trailing comma
) { /*...*/ }
```

Much like regular properties, properties declared in the primary constructor can be mutable (var) or read-only (val).

If the constructor has annotations or visibility modifiers, the constructor keyword is required and the modifiers go before it:

```
class Customer public @Inject constructor(name: String) { /*...*/ }
```

Learn more about [visibility modifiers](#).

Secondary constructors

A class can also declare secondary constructors, which are prefixed with constructor:

```
class Person(val pets: MutableList<Pet> = mutableListOf())

class Pet {
    constructor(owner: Person) {
        owner.pets.add(this) // adds this pet to the list of its owner's pets
    }
}
```

If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor, either directly or indirectly through another secondary constructor(s). Delegation to another constructor of the same class is done using the this keyword:

```
class Person(val name: String) {
    val children: MutableList<Person> = mutableListOf()
```

```

constructor(name: String, parent: Person) : this(name) {
    parent.children.add(this)
}

```

Code in initializer blocks effectively becomes part of the primary constructor. Delegation to the primary constructor happens at the moment of access to the first statement of a secondary constructor, so the code in all initializer blocks and property initializers is executed before the body of the secondary constructor.

Even if the class has no primary constructor, the delegation still happens implicitly, and the initializer blocks are still executed:

```

//sampleStart
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor $i")
    }
}
//sampleEnd

fun main() {
    Constructors(1)
}

```

If a non-abstract class does not declare any constructors (primary or secondary), it will have a generated primary constructor with no arguments. The visibility of the constructor will be public.

If you don't want your class to have a public constructor, declare an empty primary constructor with non-default visibility:

```
class DontCreateMe private constructor() { /*...*/ }
```

On the JVM, if all of the primary constructor parameters have default values, the compiler will generate an additional parameterless constructor which will use the default values. This makes it easier to use Kotlin with libraries such as Jackson or JPA that create class instances through parameterless constructors.

```
class Customer(val customerName: String = "")
```

Creating instances of classes

To create an instance of a class, call the constructor as if it were a regular function. You can assign the created instance to a [variable](#):

```

val invoice = Invoice()
val customer = Customer("Joe Smith")

```

Kotlin does not have a new keyword.

The process of creating instances of nested, inner, and anonymous inner classes is described in [Nested classes](#).

Class members

Classes can contain:

- [Constructors and initializer blocks](#)
- [Functions](#)
- [Properties](#)

- [Nested and inner classes](#)
- [Object declarations](#)

Inheritance

Classes can be derived from each other and form inheritance hierarchies. [Learn more about inheritance in Kotlin](#).

Abstract classes

A class may be declared abstract, along with some or all of its members. An abstract member does not have an implementation in its class. You don't need to annotate abstract classes or functions with open.

```
abstract class Polygon {
    abstract fun draw()
}

class Rectangle : Polygon() {
    override fun draw() {
        // draw the rectangle
    }
}
```

You can override a non-abstract open member with an abstract one.

```
open class Polygon {
    open fun draw() {
        // some default polygon drawing method
    }
}

abstract class WildShape : Polygon() {
    // Classes that inherit WildShape need to provide their own
    // draw method instead of using the default on Polygon
    abstract override fun draw()
}
```

Companion objects

If you need to write a function that can be called without having a class instance but that needs access to the internals of a class (such as a factory method), you can write it as a member of an [object declaration](#) inside that class.

Even more specifically, if you declare a [companion object](#) inside your class, you can access its members using only the class name as a qualifier.

Inheritance

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
```

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)
}
```

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.

```
//sampleStart
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") }
}
//sampleEnd

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"
}

//sampleStart
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
        }
    }
}
//sampleEnd
```

```

        of borderColor's get()
    }
}
//sampleEnd

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.

Properties

Declaring properties

Properties in Kotlin classes can be declared either as mutable, using the var keyword, or as read-only, using the val keyword.

```

class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
    var state: String? = null
    var zip: String = "123456"
}

```

To use a property, simply refer to it by its name:

```

fun copyAddress(address: Address): Address {
    val result = Address() // there's no 'new' keyword in Kotlin
    result.name = address.name // accessors are called
    result.street = address.street
    // ...
    return result
}

```

Getters and setters

The full syntax for declaring a property is as follows:

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

The initializer, getter, and setter are optional. The property type is optional if it can be inferred from the initializer or the getter's return type, as shown below:

```
var initialized = 1 // has type Int, default getter and setter
// var allByDefault // ERROR: explicit initializer required, default getter and setter implied
```

The full syntax of a read-only property declaration differs from a mutable one in two ways: it starts with val instead of var and does not allow a setter:

```
val simple: Int? // has type Int, default getter, must be initialized in constructor
val inferredType = 1 // has type Int and a default getter
```

You can define custom accessors for a property. If you define a custom getter, it will be called every time you access the property (this way you can implement a computed property). Here's an example of a custom getter:

```
//sampleStart
class Rectangle(val width: Int, val height: Int) {
    val area: Int // property type is optional since it can be inferred from the getter's return type
        get() = this.width * this.height
}
//sampleEnd
fun main() {
    val rectangle = Rectangle(3, 4)
    println("Width=${rectangle.width}, height=${rectangle.height}, area=${rectangle.area}")
}
```

You can omit the property type if it can be inferred from the getter:

```
val area get() = this.width * this.height
```

If you define a custom setter, it will be called every time you assign a value to the property, except its initialization. A custom setter looks like this:

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // parses the string and assigns values to other properties
    }
```

By convention, the name of the setter parameter is value, but you can choose a different name if you prefer.

If you need to annotate an accessor or change its visibility, but you don't want to change the default implementation, you can define the accessor without defining its body:

```
var setterVisibility: String = "abc"
    private set // the setter is private and has the default implementation

var setterWithAnnotation: Any? = null
    @Inject set // annotate the setter with Inject
```

Backing fields

In Kotlin, a field is only used as a part of a property to hold its value in memory. Fields cannot be declared directly. However, when a property needs a backing field, Kotlin provides it automatically. This backing field can be referenced in the accessors using the field identifier:

```
var counter = 0 // the initializer assigns the backing field directly
    set(value) {
        if (value >= 0)
            field = value
        // counter = value // ERROR StackOverflow: Using actual name 'counter' would make setter recursive
    }
```

The field identifier can only be used in the accessors of the property.

A backing field will be generated for a property if it uses the default implementation of at least one of the accessors, or if a custom accessor references it through the field identifier.

For example, there would be no backing field in the following case:

```
val isEmpty: Boolean
    get() = this.size == 0
```

Backing properties

If you want to do something that does not fit into this implicit backing field scheme, you can always fall back to having a backing property:

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null) {
            _table = HashMap() // Type parameters are inferred
        }
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

On the JVM: Access to private properties with default getters and setters is optimized to avoid function call overhead.

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. Such a property needs to fulfil the following requirements:

- It must be a top-level property, or a member of an [object declaration](#) or a [companion object](#).
- It must be initialized with a value of type String or a primitive type
- It cannot be a custom getter

The compiler will inline usages of the constant, replacing the reference to the constant with its actual value. However, the field will not be removed and therefore can be interacted with using [reflection](#).

Such properties can also be used in annotations:

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"
@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

Late-initialized properties and variables

Normally, properties declared as having a non-nullable type must be initialized in the constructor. However, it is often the case that doing so is not convenient. For example, properties can be initialized through dependency injection, or in the setup method of a unit test. In these cases, you cannot supply a non-nullable initializer in the constructor, but you still want to avoid null checks when referencing the property inside the body of a class.

To handle such cases, you can mark the property with the `lateinit` modifier:

```
public class MyTest {
    lateinit var subject: TestSubject

    @setUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method() // dereference directly
    }
}
```

This modifier can be used on var properties declared inside the body of a class (not in the primary constructor, and only when the property does not have a custom getter or setter), as well as for top-level properties and local variables. The type of the property or variable must be non-nullable, and it must not be a primitive type.

Accessing a lateinit property before it has been initialized throws a special exception that clearly identifies the property being accessed and the fact that it hasn't been initialized.

Checking whether a lateinit var is initialized

To check whether a lateinit var has already been initialized, use `.isInitialized` on the reference to that property:

```
if (foo::bar.isInitialized) {  
    println(foo.bar)  
}
```

This check is only available for properties that are lexically accessible when declared in the same type, in one of the outer types, or at top level in the same file.

Overriding properties

See [Overriding properties](#)

Delegated properties

The most common kind of property simply reads from (and maybe writes to) a backing field, but custom getters and setters allow you to use properties so one can implement any sort of behavior of a property. Somewhere in between the simplicity of the first kind and variety of the second, there are common patterns for what properties can do. A few examples: lazy values, reading from a map by a given key, accessing a database, notifying a listener on access.

Such common behaviors can be implemented as libraries using [delegated properties](#).

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()).

Functional (SAM) interfaces

An interface with only one abstract method is called a functional interface, or a Single Abstract Method (SAM) interface. The functional interface can have several non-abstract members but only one abstract member.

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 members and one abstract member. 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.

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).
- A set of files compiled with one invocation of the <kotlinc> Ant task.

Extensions

Kotlin provides the ability to extend a class or an interface with new functionality without having to inherit from the class or use design patterns such as Decorator. This is done via special declarations called extensions.

For example, you can write new functions for a class or an interface from a third-party library that you can't modify. Such functions can be called in the usual way, as if they were methods of the original class. This mechanism is called an extension function. There are also extension properties that let you define new properties for existing classes.

Extension functions

To declare an extension function, prefix its name with a receiver type, which refers to the type being extended. The following adds a swap function to `MutableList<Int>`:

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}
```

The `this` keyword inside an extension function corresponds to the receiver object (the one that is passed before the dot). Now, you can call such a function on any `MutableList<Int>`:

```
val list = mutableListOf(1, 2, 3)
list.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'list'
```

This function makes sense for any MutableList<T>, and you can make it generic:

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}
```

You need to declare the generic type parameter before the function name to make it available in the receiver type expression. For more information about generics, see [generic functions](#).

Extensions are resolved statically

Extensions do not actually modify the classes they extend. By defining an extension, you are not inserting new members into a class, only making new functions callable with the dot-notation on variables of this type.

Extension functions are dispatched statically. So which extension function is called is already known at compile time based on the receiver type. For example:

```
fun main() {
//sampleStart
    open class Shape
    class Rectangle: Shape()

    fun Shape.getName() = "Shape"
    fun Rectangle.getName() = "Rectangle"

    fun printClassName(s: Shape) {
        println(s.getName())
    }

    printClassName(Rectangle())
//sampleEnd
}
```

This example prints Shape, because the extension function called depends only on the declared type of the parameter s, which is the Shape class.

If a class has a member function, and an extension function is defined which has the same receiver type, the same name, and is applicable to given arguments, the member always wins. For example:

```
fun main() {
//sampleStart
    class Example {
        fun printFunctionType() { println("Class method") }
    }

    fun Example.printFunctionType() { println("Extension function") }

    Example().printFunctionType()
//sampleEnd
}
```

This code prints Class method.

However, it's perfectly OK for extension functions to overload member functions that have the same name but a different signature:

```
fun main() {
//sampleStart
    class Example {
        fun printFunctionType() { println("Class method") }
    }

    fun Example.printFunctionType(i: Int) { println("Extension function #$i") }

    Example().printFunctionType(1)
//sampleEnd
}
```

Nullable receiver

Note that extensions can be defined with a nullable receiver type. These extensions can be called on an object variable even if its value is null. If the receiver is null, then this is also null. So when defining an extension with a nullable receiver type, we recommend performing a `this == null` check inside the function body to avoid compiler errors.

You can call `toString()` in Kotlin without checking for null, as the check already happens inside the extension function:

```
fun Any?.toString(): String {
    if (this == null) return "null"
    // After the null check, 'this' is autocast to a non-nullable type, so the toString() below
    // resolves to the member function of the Any class
    return toString()
}
```

Extension properties

Kotlin supports extension properties much like it supports functions:

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

Since extensions do not actually insert members into classes, there's no efficient way for an extension property to have a [backing field](#). This is why initializers are not allowed for extension properties. Their behavior can only be defined by explicitly providing getters/setters.

Example:

```
val House.number = 1 // error: initializers are not allowed for extension properties
```

Companion object extensions

If a class has a [companion object](#) defined, you can also define extension functions and properties for the companion object. Just like regular members of the companion object, they can be called using only the class name as the qualifier:

```
class MyClass {
    companion object { } // will be called "Companion"
}

fun MyClass.Companion.printCompanion() { println("companion") }

fun main() {
    MyClass.printCompanion()
}
```

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()
}

```

See [Imports](#) for more information.

Declaring extensions as members

You can declare extensions for one class inside another class. Inside such an extension, there are multiple implicit receivers - objects whose members can be accessed without a qualifier. An instance of a class in which the extension is declared is called a dispatch receiver, and an instance of the receiver type of the extension method is called an extension receiver.

```

class Host(val hostname: String) {
    fun printHostname() { print(hostname) }
}

class Connection(val host: Host, val port: Int) {
    fun printPort() { print(port) }

    fun Host.printConnectionString() {
        printHostname() // calls Host.printHostname()
        print(":")
        printPort() // calls Connection.printPort()
    }

    fun connect() {
        /* */
        host.printConnectionString() // calls the extension function
    }
}

fun main() {
    Connection(Host("kotlin"), 443).connect()
    //Host("kotlin").printConnectionString() // error, the extension function is unavailable outside Connection
}

```

In the event of a name conflict between the members of a dispatch receiver and an extension receiver, the extension receiver takes precedence. To refer to the member of the dispatch receiver, you can use the [qualified this syntax](#).

```

class Connection {
    fun Host.getConnectionString() {
        toString() // calls Host.toString()
        this@Connection.toString() // calls Connection.toString()
    }
}

```

Extensions declared as members can be declared as open and overridden in subclasses. This means that the dispatch of such functions is virtual with regard to the dispatch receiver type, but static with regard to the extension receiver type.

```

open class Base {}

class Derived : Base() {}

open class BaseCaller {
    open fun Base.printFunctionInfo() {
        println("Base extension function in BaseCaller")
    }

    open fun Derived.printFunctionInfo() {
        println("Derived extension function in BaseCaller")
    }

    fun call(b: Base) {
        b.printFunctionInfo() // call the extension function
    }
}

class DerivedCaller: BaseCaller() {
    override fun Base.printFunctionInfo() {
        println("Base extension function in DerivedCaller")
    }

    override fun Derived.printFunctionInfo() {

```

```

        println("Derived extension function in DerivedCaller")
    }
}

fun main() {
    BaseCaller().call(Base()) // "Base extension function in BaseCaller"
    DerivedCaller().call(Base()) // "Base extension function in DerivedCaller" - dispatch receiver is resolved virtually
    DerivedCaller().call(Derived()) // "Base extension function in DerivedCaller" - extension receiver is resolved statically
}

```

Note on visibility

Extensions utilize the same [visibility modifiers](#) as regular functions declared in the same scope would. For example:

- An extension declared at the top level of a file has access to the other private top-level declarations in the same file.
- If an extension is declared outside its receiver type, it cannot access the receiver's private or protected members.

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 (see below).

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() {  
    //sampleStart  
    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)  
    //sampleEnd  
}
```

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)
```

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.

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

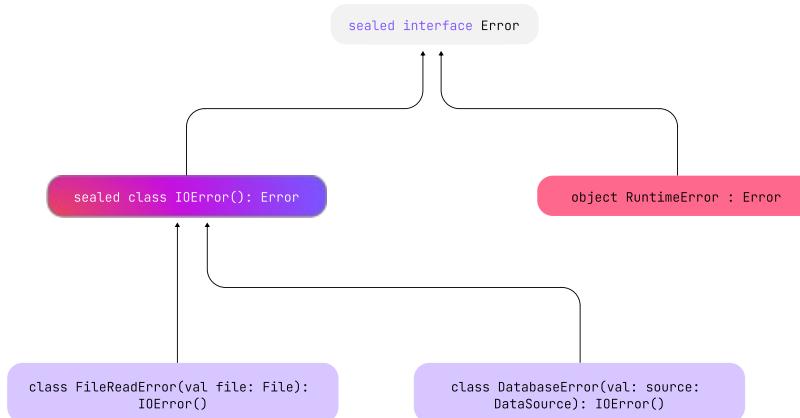
// Create a sealed class that implements sealed interface Error
sealed class IOError(): Error

// 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.

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()  
}  
  
//sampleStart  
// 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  
}  
//sampleEnd
```

```
// List all errors
fun main() {
    val errors = listOf(
        Error.FileReadError("example.txt"),
        Error.DatabaseError("usersDatabase"),
        Error.RuntimeError
    )

    errors.forEach { log(it) }
}
```

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:

```
// 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 getUserId call
fun getUserId(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 = getUserId("validUserId")
    println(userResponse)

    val userNotFoundResponse = getUserId("invalidId")
    println(userNotFoundResponse)
}
```

```
}
```

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)
```

But if the parameters can be inferred, for example, from the constructor arguments, you can omit the type arguments:

```
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  
  
// 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
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 Strings and reading Objects from it is fine. Conversely, if you can only put items into the collection, it's okay to take a collection of Objects and put Strings into it: in Java there is `List<? super String>`, which accepts Strings 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
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
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 `CharSequence` or an array of `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.

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-null types

To make interoperability with generic Java classes and interfaces easier, Kotlin supports declaring a generic type parameter as definitely non-null.

To declare a generic type T as definitely non-null, declare the type with & Any. For example: T & Any.

A definitely non-null type must have a nullable [upper bound](#).

The most common use case for declaring definitely non-null 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-null 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.

```
//sampleStart
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

//sampleEnd

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)
}
```

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") }
```

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

//sampleStart
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)
}
//sampleEnd

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() {
    //sampleStart
    println(RGB.RED.name)      // prints RED
}

```

```
    println(RGB.RED.ordinal) // prints 0
    //sampleEnd
}
```

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 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) {}

```

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"
}

```

Object expressions and declarations

Sometimes you need to create an object that is a slight modification of some class, without explicitly declaring a new subclass for it. Kotlin can handle this with object expressions and object declarations.

Object expressions

Object expressions create objects of anonymous classes, that is, classes that aren't explicitly declared with the class declaration. Such classes are useful for one-time use. You can define them from scratch, inherit from existing classes, or implement interfaces. Instances of anonymous classes are also called anonymous objects because they are defined by an expression, not a name.

Creating anonymous objects from scratch

Object expressions start with the object keyword.

If you just need an object that doesn't have any nontrivial supertypes, write its members in curly braces after object:

```

fun main() {
//sampleStart
    val helloWorld = object {
        val hello = "Hello"
        val world = "World"
        // object expressions extend Any, so `override` is required on `toString()`
        override fun toString() = "$hello $world"
    }

    print(helloWorld)
//sampleEnd
}

```

Inheriting anonymous objects from supertypes

To create an object of an anonymous class 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 appropriate constructor parameters to it. Multiple supertypes can be specified as a comma-delimited list after the colon:

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B { /*...*/ }

val ab: A = object : A(1), B {
    override val y = 15
}
```

Using anonymous objects as return and value types

When an anonymous object is used as a type of a local or `private` but not `inline` declaration (function or property), all its members are accessible via this function or property:

```
class C {
    private fun getObject() = object {
        val x: String = "x"
    }

    fun printX() {
        println(getObject().x)
    }
}
```

If this function or property is public or private inline, 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:

```
interface A {
    fun funFromA() {}
}
interface B

class C {
    // The return type is Any; x is not accessible
    fun getObject() = object {
        val x: String = "x"
    }

    // The return type is A; x is not accessible
    fun getObjectA() = object: A {
        override fun funFromA() {}
        val x: String = "x"
    }

    // The return type is B; funFromA() and x are not accessible
    fun getObjectB(): B = object: A, B { // explicit return type is required
        override fun funFromA() {}
        val x: String = "x"
    }
}
```

Accessing variables from anonymous objects

The code in object expressions can access variables from the enclosing scope:

```
fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // ...
}
```

Object declarations

The [Singleton](#) pattern can be useful in several cases, and Kotlin makes it easy to declare singletons:

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ...
}
```

This is called an object declaration, and it always has a name following the `object` keyword. Just like a variable declaration, an object declaration is not an expression, and it cannot be used on the right-hand side of an assignment statement.

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(...)
```

Such objects can have supertypes:

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ... }

    override fun mouseEntered(e: MouseEvent) { ... }
}
```

Object declarations can't be local (that is, they can't be nested directly inside a function), but they can be nested into 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@1f32e575
}
```

Just like [data classes](#), you can mark an object declaration with the `data` modifier. This instructs the compiler to generate a number of functions for your object:

- `toString()` returns the name of the data object

- equals()/hashCode() pair

You can't provide a custom equals or hashCode implementation for a data object.

The `toString()` function of a data object returns the name of the object:

```
data object MyDataObject {
    val x: Int = 3
}

fun main() {
    println(MyDataObject) // MyDataObject
}
```

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, 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` 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.getDeclaredConstructors()[0].apply { isAccessible = true } as Constructor<MySingleton>).newInstance()
}
```

The generated `hashCode()` function has 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 singleton objects, no `copy()` function is generated. The singleton pattern restricts 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 would not make sense, no `componentN()` functions are generated.

Using data objects with sealed hierarchies

Data object declarations are particularly useful for sealed hierarchies like [sealed classes](#) or [sealed interfaces](#), since 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

An object declaration inside a class can be marked with the companion keyword:

```

class MyClass {
    companion object {
        fun create(): MyClass = MyClass()
    }
}

```

Members of the companion object can be called simply by using the class name as the qualifier:

```
val instance = MyClass.create()
```

The name of the companion object can be omitted, in which case the name Companion will be used:

```

class MyClass {
    companion object {}
}

val x = MyClass.Companion

```

Class members can access the private members of the corresponding companion object.

The name of a class used by itself (not as a qualifier to another name) acts as a reference to the companion object of the class (whether named or not):

```

class MyClass1 {
    companion object Named {}
}

val x = MyClass1

class MyClass2 {
    companion object {}
}

val y = MyClass2

```

Note that even though the members of companion objects look like static members in other languages, at runtime those are still instance members of real objects, and can, for example, implement interfaces:

```

interface Factory<T> {
    fun create(): T
}

class MyClass {
    companion object : Factory<MyClass> {
        override fun create(): MyClass = MyClass()
    }
}

val f: Factory<MyClass> = MyClass

```

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.

Semantic difference between object expressions and declarations

There is one important semantic difference between object expressions and object declarations:

- 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.

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 b = BaseImpl(10)  
    Derived(b).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 b = BaseImpl(10)  
    Derived(b).printMessage()  
    Derived(b).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(val 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](#).

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 vars).

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 to 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
    ))
    //sampleStart
    println(user.name) // Prints "John Doe"
    println(user.age) // Prints 25
    //sampleEnd
}
```

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 the 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
```

Type aliases

Type aliases provide alternative names for existing types. If the type name is too long you can introduce a different shorter name and use the new one instead.

It's useful to shorten long generic types. For instance, it's often tempting to shrink collection types:

```
typealias NodeSet = Set<Network.Node>

typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

You can provide different aliases for function types:

```
typealias MyHandler = (Int, String, Any) -> Unit

typealias Predicate<T> = (T) -> Boolean
```

You can have new names for inner and nested classes:

```
class A {
    inner class Inner
}
class B {
    inner class Inner
}

typealias AInner = A.Inner
typealias BInner = B.Inner
```

Type aliases do not introduce new types. They are equivalent to the corresponding underlying types. When you add typealias Predicate<T> and use Predicate<Int> in your code, the Kotlin compiler always expands it to (Int) -> Boolean. Thus you can pass a variable of your type whenever a general function type is required and vice versa:

```
typealias Predicate<T> = (T) -> Boolean

fun foo(p: Predicate<Int>) = p(42)

fun main() {
    val f: (Int) -> Boolean = { it > 0 }
    println(foo(f)) // prints "true"

    val p: Predicate<Int> = { it > 0 }
    println(listOf(1, -2).filter(p)) // prints "[1]"
}
```

Functions

Kotlin functions are declared using the fun keyword:

```
fun double(x: Int): Int {
    return 2 * x
}
```

Function usage

Functions are called using the standard approach:

```
val result = double(2)
```

Calling member functions uses dot notation:

```
Stream().read() // create instance of class Stream and call read()
```

Parameters

Function parameters are defined using Pascal notation - name: type. Parameters are separated using commas, and each parameter must be explicitly typed:

```
fun powerOf(number: Int, exponent: Int): Int { /*...*/ }
```

You can use a trailing comma when you declare function parameters:

```
fun powerOf(
    number: Int,
    exponent: Int, // trailing comma
) { /*...*/ }
```

Default arguments

Function parameters can have default values, which are used when you skip the corresponding argument. This reduces the number of overloads:

```
fun read(
    b: ByteArray,
    off: Int = 0,
    len: Int = b.size,
) { /*...*/ }
```

A default value is set by appending = to the type.

Overriding methods always use the base method's default parameter values. When overriding a method that has default parameter values, the default parameter values must be omitted from the signature:

```
open class A {
    open fun foo(i: Int = 10) { /*...*/ }
}

class B : A() {
    override fun foo(i: Int) { /*...*/ } // No default value is allowed.
}
```

If a default parameter precedes a parameter with no default value, the default value can only be used by calling the function with [named arguments](#):

```
fun foo(
    bar: Int = 0,
    baz: Int,
) { /*...*/ }

foo(baz = 1) // The default value bar = 0 is used
```

If the last argument after default parameters is a [lambda](#), you can pass it either as a named argument or [outside the parentheses](#):

```
fun foo(
    bar: Int = 0,
    baz: Int = 1,
    qux: () -> Unit,
) { /*...*/ }

foo(1) { println("hello") } // Uses the default value baz = 1
foo(qux = { println("hello") }) // Uses both default values bar = 0 and baz = 1
foo { println("hello") } // Uses both default values bar = 0 and baz = 1
```

Named arguments

You can name one or more of a function's arguments when calling it. This can be helpful when a function has many arguments and it's difficult to associate a value with an argument, especially if it's a boolean or null value.

When you use named arguments in a function call, you can freely change the order that they are listed in. If you want to use their default values, you can just leave these arguments out altogether.

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 don't have to name all its arguments:

```
reformat(
    "String!",
    false,
    upperCaseFirstLetter = false,
    divideByCamelHumps = true,
    '_'
```

You can skip all the ones with default values:

```
reformat("This is a long String!")
```

You are also able to skip specific 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) with names using the spread operator:

```
fun foo(vararg strings: String) { /*...*/ }

foo(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.

Unit-returning functions

If a function does not return a useful value, its return type is Unit. Unit is a type with only one value - Unit. This value does not have to be returned explicitly:

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello $name")
    else
        println("Hi there!")
    // `return Unit` or `return` is optional
}
```

The Unit return type declaration is also optional. The above code is equivalent to:

```
fun printHello(name: String?) { ... }
```

Single-expression functions

When the function body consists of a single expression, the curly braces can be omitted and the body specified after an = symbol:

```
fun double(x: Int): Int = x * 2
```

Explicitly declaring the return type is optional when this can be inferred by the compiler:

```
fun double(x: Int) = x * 2
```

Explicit return types

Functions with block body must always specify return types explicitly, unless it's intended for them to return Unit, in which case specifying the return type is optional.

Kotlin does not infer return types for functions with block bodies because such functions may have complex control flow in the body, and the return type will be non-obvious to the reader (and sometimes even for the compiler).

Variable number of arguments (varargs)

You can mark a parameter of a function (usually the last one) with the vararg modifier:

```
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
}
```

In this case, you can pass a variable number of arguments to the function:

```
val list = asList(1, 2, 3)
```

Inside a function, a vararg-parameter of type T is visible as an array of T, as in the example above, where the ts variable has type `Array<out T>`.

Only one parameter can be marked as vararg. If a vararg parameter is not the last one in the list, values for the subsequent parameters can be passed using named argument syntax, or, if the parameter has a function type, by passing a lambda outside the parentheses.

When you call a vararg-function, you can pass arguments individually, for example `asList(1, 2, 3)`. If you already have an array and want to pass its contents to the function, use the spread operator (prefix the array with *):

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

If you want to pass a primitive type array into vararg, you need to convert it to a regular (typed) array using the `toTypedArray()` function:

```
val a = intArrayOf(1, 2, 3) // IntArray is a primitive type array
val list = asList(-1, 0, *a.toTypedArray(), 4)
```

Infix notation

Functions marked with the `infix` keyword can also be called using the infix notation (omitting the dot and the parentheses for the call). Infix functions must meet the following requirements:

- They must be member functions or extension functions.
- They must have a single parameter.
- The parameter must not accept variable number of arguments and must have no default value.

```
infix fun Int.shl(x: Int): Int { ... }

// calling the function using the infix notation
1 shl 2

// is the same as
1.shl(2)
```

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're calling a method on the current receiver using the infix notation, use this explicitly. This is required to ensure unambiguous parsing.

```
class MyStringCollection {
    infix fun add(s: String) { /*...*/ }
```

```

    fun build() {
        this.add("abc") // Correct
        add("abc") // Correct
        //add "abc" // Incorrect: the receiver must be specified
    }
}

```

Function scope

Kotlin functions can be declared at the top level in a file, meaning you do not need to create a class to hold a function, which you are required to do in languages such as Java, C#, and Scala ([top level definition is available since Scala 3](#)). In addition to top level functions, Kotlin functions can also be declared locally as member functions and extension functions.

Local functions

Kotlin supports local functions, which are functions inside other functions:

```

fun dfs(graph: Graph) {
    fun dfs(current: Vertex, visited: MutableSet<Vertex>) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v, visited)
    }

    dfs(graph.vertices[0], HashSet())
}

```

A local function can access local variables of outer functions (the closure). In the case above, `visited` can be a local variable:

```

fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}

```

Member functions

A member function is a function that is defined inside a class or object:

```

class Sample {
    fun foo() { print("Foo") }
}

```

Member functions are called with dot notation:

```
Sample().foo() // creates instance of class Sample and calls foo
```

For more information on classes and overriding members see [Classes](#) and [Inheritance](#).

Generic functions

Functions can have generic parameters, which are specified 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:

```
val eps = 1E-10 // "good enough", could be 10^-15

tailrec fun findFixPoint(x: Double = 1.0): Double =
    if (Math.abs(x - Math.cos(x)) < eps) x else findFixPoint(Math.cos(x))
```

This code calculates the fixpoint of cosine, which is a mathematical constant. It simply calls Math.cos repeatedly starting at 1.0 until the result no longer changes, yielding a result of 0.7390851332151611 for the specified eps precision. The resulting code is equivalent to this more traditional style:

```
val eps = 1E-10 // "good enough", could be 10^-15

private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if (Math.abs(x - y) < eps) return x
        x = Math.cos(x)
    }
}
```

To be eligible for the tailrec modifier, a function must call itself as the last operation it performs. You cannot use tail recursion when there is more code after the recursive call, within try/catch/finally blocks, or on open functions. Currently, tail recursion is supported by Kotlin for the JVM and Kotlin/Native.

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](#) $(R, T) \rightarrow R$, 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() {
    //sampleStart
    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() {
    //sampleStart
    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() {
    //sampleStart
    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
```

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
}
```

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 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!")
}
//sampleStart
fun foo() {
    ordinaryFunction {
        return // ERROR: cannot make `foo` return here
    }
}
//sampleEnd
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!")
}
//sampleStart
fun foo() {
    inlined {
        return // OK: the lambda is inlined
    }
}
//sampleEnd
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 are not yet available in inlined lambdas, but we are planning to support them, too.

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()

Expression Translated to

!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)

a..b a.rangeTo(b)

a..**a** 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

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)

Expression Translated to

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 can be overridden to provide custom equality check implementation. Any other function with the same name (like `equals(other: Foo)`) will not be called.

`==` and `!=` (identity checks) are not overloadable, so no conventions exist for them.

The `==` operation is special: it is translated to a complex expression that screens for null's. `null == null` is always true, and `x == null` for a non-null `x` is always false and won't invoke `x.equals()`.

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](#).

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:

```
import com.example.html.* // see declarations below

fun result() =
    html {
        head {
            title {"XML encoding with Kotlin"}
        }
        body {
            h1 {"XML encoding with Kotlin"}
            p {"this format can be used as an alternative markup to XML"}

            // an element with attributes and text content
            a(href = "https://kotlinlang.org") {"Kotlin"}

            // mixed content
            p {
                +"This is some"
                b {"mixed"}
                +"text. For more see the"
                a(href = "https://kotlinlang.org") {"Kotlin"}
                +"project"
            }
            p {"some text"}

            // content generated by
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

This is completely legitimate Kotlin code. You can [play with this code online \(modify it and run in the browser\) here](#).

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  
annotation class HtmlTagMarker
```

An annotation class is called a DSL marker if it is annotated with the @DslMarker annotation.

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
    }
    // ...
}

```

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
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>.() -> 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>.

Null safety

Nullable types and non-nullable types

Kotlin's type system is aimed at eliminating the danger 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 will result in a null reference exception. In Java this would be the equivalent of a NullPointerException, or an NPE for short.

The only possible causes of an NPE in Kotlin are:

- An explicit call to throw NullPointerException().
- Usage of the !! operator that is described below.
- Data inconsistency with regard to initialization, such as when:
 - An uninitialized this available in a constructor is passed and used somewhere (a "leaking this").
 - A [superclass constructor calls 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 being used for Java interoperation. For example, a piece of Java code might add null into a Kotlin MutableList<String>, therefore requiring a MutableList<String?> for working with it.
 - Other issues caused by external Java code.

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:

```
fun main() {
    //sampleStart
    var a: String = "abc" // Regular initialization means non-nullable by default
    a = null // compilation error
    //sampleEnd
}
```

To allow nulls, you can declare a variable as a nullable string by writing String?:

```
fun main() {
    //sampleStart
    var b: String? = "abc" // can be set to null
    b = null // ok
    print(b)
    //sampleEnd
}
```

Now, if you call a method or access a property on a, it's guaranteed not to cause an NPE, so you can safely say:

```
val l = a.length
```

But if you want to access the same property on b, that would not be safe, and the compiler reports an error:

```
val l = b.length // error: variable 'b' can be null
```

But you still need to access that property, right? There are a few ways to do so.

Checking for null in conditions

First, you can explicitly check whether b is null, and handle the two options separately:

```
val l = if (b != null) b.length else -1
```

The compiler tracks the information about the check you performed, and allows the call to length inside the if. More complex conditions are supported as well:

```
fun main() {
//sampleStart
    val b: String? = "Kotlin"
    if (b != null && b.length > 0) {
        print("String of length ${b.length}")
    } else {
        print("Empty string")
    }
//sampleEnd
}
```

Note that this only works where b is immutable (meaning it is a local variable that is not modified between the check and its usage or it is a member val that has a backing field and is not overridable), because otherwise it could be the case that b changes to null after the check.

Safe calls

Your second option for accessing a property on a nullable variable is using the safe call operator ?:.

```
fun main() {
//sampleStart
    val a = "Kotlin"
    val b: String? = null
    println(b?.length)
    println(a?.length) // Unnecessary safe call
//sampleEnd
}
```

This returns b.length if b is not null, and null otherwise. The type of this expression is Int?.

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
```

Such a chain returns null if any of the properties in it is null.

To perform a certain operation only for non-null values, you can use the safe call operator together with let:

```
fun main() {
//sampleStart
    val listWithNulls: List<String?> = listOf("Kotlin", null)
    for (item in listWithNulls) {
        item?.let { println(it) } // prints Kotlin and ignores null
    }
//sampleEnd
}
```

A safe call can also be placed on the left side of an assignment. Then, if one of the receivers in the safe calls chain is null, the assignment is skipped and the expression on the right is not evaluated at all:

```
// If either `person` or `person.department` is null, the function is not called:
person?.department?.head = managersPool.getManager()
```

Nullable receiver

Extension functions can be defined on a nullable receiver. This way you can specify behaviour for null values without the need to use null-checking logic at each

call-site.

For example, the `toString()` function is defined on a nullable receiver. It returns the String "null" (as opposed to a null value). This can be helpful in certain situations, for example, logging:

```
val person: Person? = null
logger.debug(person.toString()) // Logs "null", does not throw an exception
```

If you want your `toString()` invocation to return a nullable string, use the [safe-call operator `?.`](#):

```
var timestamp: Instant? = null
val isoTimestamp = timestamp?.toString() // Returns a String? object which is `null`
if (isoTimestamp == null) {
    // Handle the case where timestamp was `null`
}
```

Elvis operator

When you have a nullable reference, `b`, you can say "if `b` is not null, use it, otherwise use some non-null value":

```
val l: Int = if (b != null) b.length else -1
```

Instead of writing the complete if expression, you can also express this with the Elvis operator `?::`

```
val l = b?.length ?: -1
```

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.

Since `throw` and `return` are expressions in Kotlin, they can also be used on the right-hand side of the Elvis operator. This can be handy, for example, when checking function arguments:

```
fun foo(node: Node): String? {
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // ...
}
```

The `!!` operator

The third option is for NPE-lovers: the not-null assertion operator `!!` converts any value to a non-nullable type and throws an exception if the value is null. You can write `b!!`, and this will return a non-null value of `b` (for example, a String in our example) or throw an NPE if `b` is null:

```
val l = b!!.length
```

Thus, if you want an NPE, you can have it, but you have to ask for it explicitly and it won't appear out of the blue.

Safe casts

Regular casts may result in a `ClassCastException` if the object is not of the target type. Another option is to use safe casts that return null if the attempt was not successful:

```
val aInt: Int? = a as? Int
```

Collections of a nullable type

If you have a collection of elements of a nullable type and want to filter non-nullable elements, you can do so by using `filterNotNull`:

```
val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()
```

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 a 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](#).

This expressions

To denote the current receiver, you use this expressions:

- In a member of a class, this refers to the current object of that class.
- In an extension function or a function literal with receiver this denotes the receiver parameter that is passed on the left-hand side of a dot.

If this has no qualifiers, it refers to the innermost enclosing scope. To refer to this in other scopes, label qualifiers are used:

Qualified this

To access this from an outer scope (a class, extension function, or labeled function literal with receiver) you write this@label, where @label is a label on the scope this is meant to be from:

```
class A { // implicit label @A
    inner class B { // implicit label @B
        fun Int.foo() { // implicit label @foo
            val a = this@A // A's this
            val b = this@B // B's this

            val c = this // foo()'s receiver, an Int
            val c1 = this@foo // foo()'s receiver, an Int

            val funLit = lambda@ fun String.() {
                val d = this // funLit's receiver, a String
            }

            val funLit2 = { s: String ->
                // foo()'s receiver, since enclosing lambda expression
                // doesn't have any receiver
                val d1 = this
            }
        }
    }
}
```

Implicit this

When you call a member function on this, you can skip the this. part. If you have a non-member function with the same name, use this with caution because in some cases it can be called instead:

```
fun main() {
//sampleStart
    fun println() { println("Top-level function") }

    class A {
        fun println() { println("Member function") }

        fun invokePrintLine(omitThis: Boolean = false) {
            if (omitThis) println()
            else this.println()
        }
    }

    A().invokePrintLine() // Member function
    A().invokePrintLine(omitThis = true) // Top-level function
//sampleEnd()
}
```

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 server-side applications, we want to avoid having the user wait or what's worse cause bottlenecks that would prevent an 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 server-side 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 multi-threaded 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 the titled christmas tree (braces represent branches of the tree).
- 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 (there are also other terms these can be referred to depending on language/platform), is that when we make a call, we're promised that at some point it will return with an object called a Promise, which can then be operated 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.

One of the benefits however 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 in 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

Asynchronous or non-blocking programming is an important part of the development landscape. When creating server-side, desktop, or mobile applications, it's important to provide an experience that is not only fluid from the user's perspective, but also scalable when needed.

Kotlin solves this problem in a flexible way by providing [coroutine](#) support at the language level and delegating most of the functionality to libraries.

In addition to opening the doors to asynchronous programming, coroutines also provide a wealth of other possibilities, such as concurrency and actors.

How to start

New to Kotlin? Take a look at the [Getting started](#) page.

Documentation

- [Coroutines guide](#)
- [Basics](#)
- [Channels](#)
- [Coroutine context and dispatchers](#)
- [Shared mutable state and concurrency](#)
- [Asynchronous flow](#)

Tutorials

- [Asynchronous programming techniques](#)
- [Introduction to coroutines and channels](#)
- [Debug coroutines using IntelliJ IDEA](#)
- [Debug Kotlin Flow using IntelliJ IDEA – tutorial](#)
- [Testing Kotlin coroutines on Android](#)

Sample projects

- [kotlinx.coroutines examples and sources](#)
- [KotlinConf app](#)

Annotations

Annotations are means of attaching metadata to code. To declare an annotation, put the annotation modifier in front of a class:

```
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 KEP](#).

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 which 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 Java field
             @get:Ann val bar,       // annotate Java getter
             @param:Ann val quux)    // annotate 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:

```
class Example {
    @set:[Inject VisibleForTesting]
    var collaborator: Collaborator
}
```

The full list of supported use-site targets is:

- file
- property (annotations with this target are not visible to Java)
- field
- get (property getter)
- set (property setter)
- receiver (receiver parameter of an extension function or property)
- param (constructor parameter)
- setparam (property setter parameter)
- delegate (the field storing the delegate instance for a delegated property)

To annotate the receiver parameter of an extension function, use the following syntax:

```
fun @receiver:Fancy String.myExtension() { ... }
```

If you don't specify a use-site target, the target is chosen according to the `@Target` annotation of the annotation being used. If there are multiple applicable targets, the first applicable target from the following list is used:

- param
- property
- field

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(annotation: Ann) {
    val i = annotation.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 KEP](#).

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!" }
```

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:2.0.0"
}
```

- 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 or Ant), it is added by default. In the command-line compiler and Ant, 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() {
    //sampleStart
    val numbers = listOf(1, 2, 3)
    println(numbers.filter(::isOdd))
    //sampleEnd
}
```

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() {
    //sampleStart
    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)
//sampleEnd
}

```

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() {
//sampleStart
    fun length(s: String) = s.length

    val oddLength = compose(::isOdd, ::length)
    val strings = listOf("a", "ab", "abc")

    println(strings.filter(oddLength))
//sampleEnd
}

```

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() {
//sampleStart
    val strs = listOf("a", "bc", "def")
    println(strs.map(String::length))
//sampleEnd
}
```

To access a property that is a member of a class, qualify it as follows:

```
fun main() {
//sampleStart
    class A(val p: Int)
    val prop = A::p
    println(prop.get(A(1)))
//sampleEnd
}
```

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() {
    //sampleStart
    val numberRegex = "\\d+".toRegex()
    println(numberRegex.matches("29"))

    val isNumber = numberRegex::matches
    println(isNumber("29"))
    //sampleEnd
}
```

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() {
    //sampleStart
    val numberRegex = "\\d+".toRegex()
    val strings = listOf("abc", "124", "a70")
    println(strings.filter(numberRegex::matches))
    //sampleEnd
}
```

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() {
    //sampleStart
    val prop = "abc)::length
    println(prop.get())
    //sampleEnd
}
```

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
```

Get started with Kotlin Multiplatform

Support for multiplatform programming is one of Kotlin's key benefits. It reduces time spent writing and maintaining the same code for [different platforms](#) while retaining the flexibility and benefits of native programming.

Learn more about [Kotlin Multiplatform](#) benefits.

Start from scratch

- [Get started with Kotlin Multiplatform](#). Create your first cross-platform application that works on Android and iOS with the help of the [Kotlin Multiplatform Mobile](#)

[plugin for Android Studio](#). Learn how to create, run, and add dependencies to multiplatform mobile applications.

- [Share UIs between iOS and Android](#). Create a Kotlin Multiplatform application that uses the [Compose Multiplatform UI framework](#) for sharing business logic and UIs among the iOS, Android, and desktop platforms.

Dive deep into Kotlin Multiplatform

Once you have gained some experience with Kotlin Multiplatform and want to know how to solve particular cross-platform development tasks:

- [Share code on platforms](#) in your Kotlin Multiplatform project.
- [Connect to platform-specific APIs](#) when developing multiplatform applications and libraries.
- [Set up targets manually](#) for your Kotlin Multiplatform project.
- [Add dependencies](#) on the standard, test, or another kotlinc library.
- [Configure compilations](#) for production and test purposes in your project.
- [Publish a multiplatform library](#) to the Maven repository.
- [Build native binaries](#) as executables or shared libraries, like universal frameworks or XCFrameworks.

Get help

- Kotlin Slack: Get an [invite](#) and join the [#multiplatform](#) channel
- StackOverflow: Subscribe to the ["kotlin-multiplatform"](#) tag
- Kotlin issue tracker: [Report a new issue](#)

The basics of Kotlin Multiplatform project structure

With Kotlin Multiplatform, you can share code among different platforms. This article explains the constraints of the shared code, how to distinguish between shared and platform-specific parts of your code, and how to specify the platforms on which this shared code works.

You'll also learn the core concepts of Kotlin Multiplatform project setup, such as common code, targets, platform-specific and intermediate source sets, and test integration. That will help you set up your multiplatform projects in the future.

The model presented here is simplified compared to the one used by Kotlin. However, this basic model should be adequate for the majority of cases.

Common code

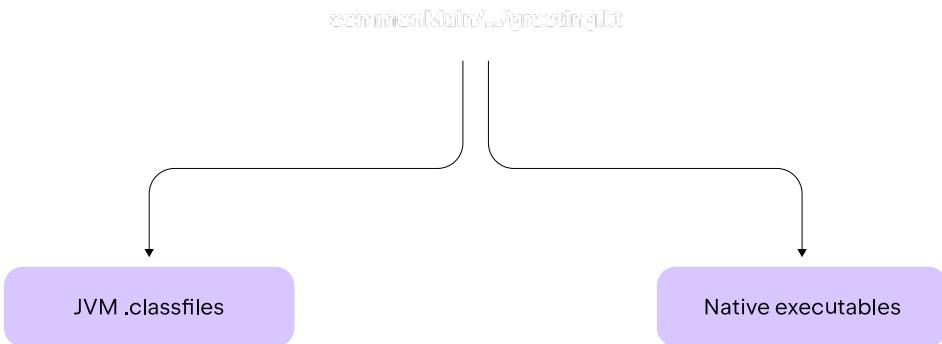
Common code is the Kotlin code shared among different platforms.

Consider the simple "Hello, World" example:

```
fun greeting() {  
    println("Hello, Kotlin Multiplatform!")  
}
```

Kotlin code shared among platforms is typically located in the `commonMain` directory. The location of code files is important, as it affects the list of platforms to which this code is compiled.

The Kotlin compiler gets the source code as input and produces a set of platform-specific binaries as a result. When compiling multiplatform projects, it can produce multiple binaries from the same code. For example, the compiler can produce JVM .class files and native executable files from the same Kotlin file:

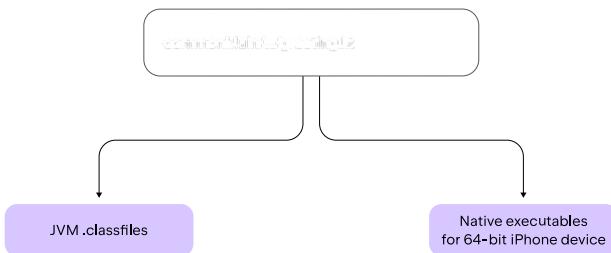


You should first declare a target to instruct Kotlin to compile code for that specific target. In Gradle, you declare targets using predefined DSL calls inside the `kotlin` {} block:

```
kotlin {  
    jvm() // Declares a JVM target  
    iosArm64() // Declares a target that corresponds to 64-bit iPhones  
}
```

This way, each multiplatform project defines a set of supported targets. See the [Hierarchical project structure](#) section to learn more about declaring targets in your build scripts.

With the `jvm` and `iosArm64` targets declared, the common code in `commonMain` will be compiled to these targets:





The screenshot shows a code editor interface with a file tree on the left and a code editor on the right.

File Tree:

- > gradle
- < shared
- > build
- < src
 - > commonMain
 - > iosMain
 - > jvmMain
- build.gradle.kts 20.09.23, 13:30, 114 B

Shared source

Code Editor:

```
In Gradle scripts, you access source sets by name inside the kotlinSourceSets block.  
    // targets declaration:  
    // -  
    // source set declaration:  
    sourceSets {  
        common // configure the commonMain source set  
        ...  
    }
```

Code from commonMain, other source sets can be either platform-specific or intermediate.

sourceSets, and jdkMain source sets for specific targets.

Compilation is a separate target.

During compilation to the JVM, Kotlin selects all source sets labeled with "JVM", namely, jvmMain and commonMain. It then compiles them together to the JVM code first.



