

Kotlin Language Documentation 1.5.21

Table of Contents

Kotlin docs	42
Get started with Kotlin	43
Learn Kotlin fundamentals	43
Create your powerful application with Kotlin	43
Is anything missing?	48
Multiplatform programming	48
Use cases	52
What's next?	53
Sample projects	53
Kotlin for server side	54
Frameworks for server-side development with Kotlin	54
Deploying Kotlin server-side applications	55
Users of Kotlin on the server side	55
Next steps	55
Kotlin for Android	55
Kotlin for JavaScript	56
Use cases for Kotlin/JS	56
Kotlin/JS frameworks	58
Kotlin/JS, Today and Tomorrow	59
Get started with Kotlin/JS	59
Hands-on labs for Kotlin/JS	59
New Kotlin/JS IR compiler	60
Join the Kotlin/JS community	60
Kotlin Native	60
Why Kotlin/Native?	60
Target platforms	61

Interoperability	61
Sharing code between platforms	61
How to start	62
Sample projects	62
Kotlin for data science	63
Interactive editors	63
Libraries	65
Kotlin for competitive programming	66
Simple example: Reachable Numbers problem	67
Functional operators example: Long Number problem	68
More tips and tricks	69
Learning Kotlin	70
What's new in Kotlin 1.5.20	70
Kotlin/JVM	71
Kotlin/Native	73
Kotlin/JS	74
Gradle	75
Standard library	75
What's new in Kotlin 1.5.0	76
Language features	77
Kotlin/JVM	80
Kotlin/Native	83
Kotlin/JS	84
Kotlin Multiplatform	85
Standard library	85
kotlin-test library	91
kotlinx libraries	94
Migrating to Kotlin 1.5.0	96

What's new in Kotlin 1.4.30	97
Language features	97
Kotlin/JVM	100
Kotlin/Native	101
Kotlin/JS	102
Gradle project improvements	102
Standard library	102
Serialization updates	104
What's new in Kotlin 1.4.20	105
Kotlin/JVM	105
Kotlin/JS	106
Kotlin/Native	109
Kotlin Multiplatform	111
Standard library	112
Kotlin Android Extensions	113
What's new in Kotlin 1.4	113
Language features and improvements	114
New tools in the IDE	119
New compiler	122
Kotlin/JVM	125
Kotlin/JS	127
Kotlin/Native	128
Kotlin Multiplatform	131
Gradle project improvements	136
Standard library	138
Stable JSON serialization	146
Scripting and REPL	146
Migrating to Kotlin 1.4.0	147
What's new in Kotlin 1.3	148

Coroutines release	148
Kotlin/Native	149
Multiplatform projects	149
Contracts	149
Capturing when subject in a variable	151
@JvmStatic and @JvmField in companions of interfaces	151
Nested declarations in annotation classes	152
Parameterless main	152
Functions with big arity	152
Progressive mode	153
Inline classes	153
Unsigned integers	154
@JvmDefault	155
Standard library	155
Tooling	157
What's new in Kotlin 1.2	158
Table of contents	158
Multiplatform projects (experimental)	159
Other language features	160
Standard library	163
JVM backend	166
JavaScript backend	167
Tools	167
What's new in Kotlin 1.1	167
Table of contents	168
JavaScript	168
Coroutines (experimental)	168
Other language features	169
Standard library	175

JVM Backend	179
JavaScript backend	180
What's new in Kotlin plugin 2021.1	181
Performance improvements	182
Evaluation of custom getters during debugging	182
Improved Change Signature refactoring	183
Code completion for type arguments	184
UML diagrams for Kotlin classes	184
Other platform enhancements	185
What's new in Kotlin plugin 2020.3	186
New types of inline refactorings	186
Structural search and replace	186
EditorConfig support	186
Project templates for Jetpack Compose for Desktop	186
Basic syntax	187
Package definition and imports	187
Program entry point	188
Print to the standard output	188
Functions	188
Variables	190
Creating classes and instances	190
Comments	191
String templates	192
Conditional expressions	192
for loop	192
while loop	193
when expression	193
Ranges	194
Collections	195

Nullable values and null checks	196
Type checks and automatic casts	197
Idioms	198
Create DTOs (POJOs/POCOs)	199
Default values for function parameters	199
Filter a list	199
Check the presence of an element in a collection	199
String interpolation	199
Instance checks	200
Read-only list	200
Read-only map	200
Access a map entry	200
Traverse a map or a list of pairs	200
Iterate over a range	200
Lazy property	201
Extension functions	201
Create a singleton	201
Instantiate an abstract class	201
If-not-null shorthand	202
If-not-null-else shorthand	202
Execute a statement if null	202
Get first item of a possibly empty collection	202
Execute if not null	202
Map nullable value if not null	202
Return on when statement	203
try-catch expression	203
if expression	203
Builder-style usage of methods that return Unit	203
Single-expression functions	203

Call multiple methods on an object instance (with)	204
Configure properties of an object (apply)	204
Java 7's try-with-resources	205
Generic function that requires the generic type information	205
Nullable Boolean	205
Swap two variables	205
Mark code as incomplete (TODO)	205
What's next?	206
Coding conventions	206
Configure style in IDE	206
Source code organization	206
Naming rules	208
Formatting	210
Documentation comments	220
Avoid redundant constructs	221
Idiomatic use of language features	221
Coding conventions for libraries	226
Basic types	227
Numbers	227
Booleans	234
Characters	234
Strings	235
Arrays	237
Type checks and casts	238
is and !is operators	238
Smart casts	239
"Unsafe" cast operator	240
"Safe" (nullable) cast operator	240
Type erasure and generic type checks	240

Unchecked casts	241
Conditions and loops	242
If expression	242
When expression	243
For loops	245
While loops	246
Break and continue in loops	246
Returns and jumps	246
Break and continue labels	247
Return at labels	247
Exceptions	249
Exception classes	249
Checked exceptions	250
The Nothing type	251
Java interoperability	251
Packages and imports	251
Default imports	252
Imports	252
Visibility of top-level declarations	253
Classes	253
Constructors	253
Creating instances of classes	256
Class members	257
Inheritance	257
Abstract classes	257
Companion objects	257
Inheritance	257
Overriding methods	258

Overriding properties	259
Derived class initialization order	259
Calling the superclass implementation	260
Overriding rules	261
Properties	262
Declaring properties	262
Getters and setters	262
Compile-time constants	264
Late-initialized properties and variables	265
Overriding properties	266
Delegated properties	266
Interfaces	266
Implementing interfaces	266
Properties in interfaces	266
Interfaces Inheritance	267
Resolving overriding conflicts	267
Functional (SAM) interfaces	268
SAM conversions	268
Functional interfaces vs. type aliases	269
Visibility modifiers	270
Packages	270
Classes and interfaces	271
Modules	272
Extensions	272
Extension functions	273
Extensions are resolved statically	273
Nullable receiver	274
Extension properties	275

Companion object extensions	275
Scope of extensions	275
Declaring extensions as members	276
Note on visibility	277
Data classes	278
Properties declared in the class body	279
Copying	279
Data classes and destructuring declarations	280
Standard data classes	280
Sealed classes	280
Location of direct subclasses	281
Sealed classes and when expression	281
Generics: in, out, where	282
Variance	282
Type projections	285
Generic functions	287
Generic constraints	287
Type erasure	288
Nested and inner classes	288
Inner classes	289
Anonymous inner classes	289
Enum classes	290
Anonymous classes	290
Implementing interfaces in enum classes	291
Working with enum constants	291
Inline classes	292
Members	293
Inheritance	293

Representation	294
Inline classes vs type aliases	295
Object expressions and declarations	296
Object expressions	296
Object declarations	298
Delegation	300
Overriding a member of an interface implemented by delegation	301
Delegated properties	302
Standard delegates	303
Delegating to another property	304
Storing properties in a map	305
Local delegated properties	306
Property delegate requirements	306
Type aliases	310
Functions	311
Function usage	311
Function scope	316
Generic functions	317
Tail recursive functions	317
High-order functions and lambdas	318
Higher-order functions	318
Function types	320
Lambda expressions and anonymous functions	322
Inline functions	326
noinline	327
Non-local returns	327
Reified type parameters	328

Inline properties	329
Restrictions for public API inline functions	330
Null safety	330
Nullable types and non-null types	330
Checking for null in conditions	331
Safe calls	332
Elvis operator	333
The !! operator	333
Safe casts	333
Collections of a nullable type	334
Equality	334
Structural equality	334
Referential equality	334
Floating-point numbers equality	335
This expressions	335
Qualified this	335
Implicit this	336
Coroutines	336
How to start	336
Sample projects	337
Type-safe builders	337
How it works	338
Scope control: @DslMarker	340
Full definition of the com.example.html package	341
Operator overloading	343
Unary operations	344
Binary operations	345
Infix calls for named functions	348

Annotations	349
Usage	349
Constructors	350
Lambdas	351
Annotation use-site targets	351
Java annotations	352
Destructuring declarations	354
Example: returning two values from a function	355
Example: destructuring declarations and maps	355
Underscore for unused variables	355
Destructuring in lambdas	356
Reflection	356
JVM dependency	357
Class references	357
Callable references	358
Kotlin Multiplatform	363
Tutorials	365
Create a multiplatform library	365
What's next?	368
Discover your project	368
Multiplatform plugin	369
Targets	369
Source sets	370
Compilations	372
Share code on platforms	373
Share code on all platforms	373
Share code on similar platforms	374

Connect to platform-specific APIs	381
Rules for expected and actual declarations	382
Set up targets manually	383
Distinguish several targets for one platform	384
Add dependencies	384
Dependency on the standard library	385
Set dependencies on test libraries	385
Set a dependency on a kotlinx library	385
Configure compilations	387
Configure all compilations	388
Configure compilations for one target	388
Configure one compilation	389
Create a custom compilation	389
Include Java sources in JVM compilations	391
Configure interop with native languages	392
Compilation for Android	394
Compilation of the source set hierarchy	395
Run tests	396
Required dependencies	396
Run tests for one or more targets	396
Test shared code	396
Publish a multiplatform library	397
Structure of publications	398
Avoid duplicate publications	398
Publish an Android library	399
Create and publish a multiplatform library – tutorial	401
Set up the environment	401
Create a project	401

Write cross-platform code	403
Provide platform-specific implementations	404
Test your library	406
Publish your library to the local Maven repository	408
Add a dependency on the published library	409
Summary	409
What's next?	410
Build final native binaries	410
Declare binaries	410
Access binaries	412
Export dependencies to binaries	414
Build universal frameworks	416
Supported platforms	417
Multiplatform Gradle DSL reference	418
Id and version	418
Top-level blocks	419
Targets	419
Source sets	428
Compilations	430
Dependencies	434
Language settings	435
Migrating multiplatform projects to Kotlin 1.4.0	436
For multiplatform project authors	436
For library authors	438
For build authors	440
For using the Kotlin/JS target	440
Tutorials for Kotlin Multiplatform	441
Get started with Kotlin/JVM	442

Create an application	442
Run the application	445
What's next?	446
Test code using JUnit in JVM – tutorial	446
Add dependencies	447
Add the code to test it	448
Create a test	448
Run a test	449
What's next	451
Create a RESTful web service with Spring Boot – tutorial	451
Bootstrap the project	452
Explore the project build file	453
Explore the Spring Boot application	454
Create a data class and a controller	454
Run the application	455
Proceed to the next tutorial	457
Add a database to a Spring Boot RESTful web service – tutorial	457
Add database support	458
Configure the database	459
Run the application	460
What's next?	463
Records	463
Using Java records from Kotlin code	463
Declare records in Kotlin	464
Further discussion	465
Calling Java from Kotlin	465
Getters and setters	465
Methods returning void	466

Escaping for Java identifiers that are keywords in Kotlin	466
Null-safety and platform types	466
Mapped types	473
Java generics in Kotlin	475
Java arrays	476
Java varargs	477
Operators	477
Checked exceptions	477
Object methods	477
Inheritance from Java classes	478
Accessing static members	478
Java reflection	479
SAM conversions	479
Using JNI with Kotlin	480
Using Lombok-generated declarations in Kotlin	480
Calling Kotlin from Java	480
Properties	480
Package-level functions	481
Instance fields	482
Static fields	483
Static methods	484
Default methods in interfaces	485
Visibility	487
KClass	487
Handling signature clashes with @JvmName	487
Overloads generation	488
Checked exceptions	489
Null-safety	489
Variant generics	490

Mixing Java and Kotlin in one project – tutorial	491
Adding Java source code to an existing Kotlin project	491
Adding Kotlin source code to an existing Java project	493
Converting an existing Java file to Kotlin with J2K	494
Comparison to Java	495
Some Java issues addressed in Kotlin	495
What Java has that Kotlin does not	496
What Kotlin has that Java does not	496
What's next?	497
Migrating from Java to Kotlin: Strings	497
Concatenate strings	497
Build a string	498
Create a string from collection items	498
Set default value if the string is blank	499
Replace characters at the beginning and end of a string	499
Replace occurrences	500
Split a string	500
Take a substring	501
Use multiline strings	502
What's next?	503
Get started with Kotlin/JS for React	503
Create an application	503
Run the application	505
Update the application	505
What's next?	509
Set up a Kotlin/JS project	510
Execution environments	512
Dependencies	512

run task	515
test task	515
webpack bundling	517
CSS	518
Yarn	519
Distribution target directory	520
Module name	520
package.json customization	521
Troubleshooting	521
 Run Kotlin/JS	521
Run the Node.js target	522
Run the browser target	523
 Development server and continuous compilation	524
 Debug Kotlin/JS in the browser	527
 Run tests in Kotlin/JS	529
 Kotlin/JS dead code elimination	532
Exclude declarations from DCE	533
Disable DCE	533
 Kotlin/JS IR compiler	534
Ignoring compilation errors	534
Lazy initialization of top-level properties	535
Preview: generation of TypeScript declaration files (d.ts)	536
Current limitations of the IR compiler	536
Migrating existing projects to the IR compiler	537
Authoring libraries for the IR compiler with backwards compatibility	537
 Migrating Kotlin/JS projects to the IR compiler	538
Convert JS- and React-related classes and interfaces to external interfaces	538

Convert properties of external interfaces to var	539
Make boolean properties nullable in external interfaces	539
Convert functions with receivers in external interfaces to regular functions	540
Create plain JS objects for interoperability	540
Replace <code>toString()</code> calls on function references with <code>.name</code>	541
Browser and DOM API	541
Interaction with the DOM	542
Use JavaScript code from Kotlin	542
Inline JavaScript	543
<code>external</code> modifier	543
Dynamic type	546
Use dependencies from npm	548
Use Kotlin code from JavaScript	549
Isolating declarations in a separate JavaScript object in plain mode	549
Package structure	549
Kotlin types in JavaScript	551
JavaScript modules	552
Browser targets	552
JavaScript libraries and Node.js files	553
<code>@JsModule</code> annotation	554
Kotlin/JS reflection	556
Class references	556
<code>KType</code> and <code>typeOf()</code>	556
Example	557
Typesafe HTML DSL	557
Generation of external declarations with Dukat	559
Generate external declarations at build time	559

Manually generate external declarations via Gradle task	560
Hands-on labs for Kotlin/JS	560
Get started with Kotlin/Native in IntelliJ IDEA	560
Create a new Kotlin/Native project in IntelliJ IDEA	560
Run the application	562
Update the application	563
What's next?	565
Get started with Kotlin/Native using Gradle	566
Create project files	566
Build the project	568
Open the project in an IDE	568
What's next?	568
Get started with Kotlin/Native using the command-line compiler	568
Obtain the compiler	568
Write "Hello Kotlin/Native" program	568
Compile the code from the console	569
Concurrency in Kotlin/Native	569
Workers	569
Object subgraph detachment	571
Raw shared memory	571
Atomic primitives and references	572
Immutability in Kotlin/Native	573
Kotlin/Native libraries	574
Kotlin compiler specifics	574
cinterop tool specifics	574
klib utility	574
Several examples	575

Advanced topics	576
Platform libraries	577
POSIX bindings	577
Popular native libraries	577
Availability by default	577
Examples	578
Kotlin/Native as a dynamic library – tutorial	578
Create a Kotlin library	578
Generated headers file	581
Use generated headers from C	585
Compile and run the example on Linux and macOS	586
Compile and run the example on Windows	586
Next steps	587
Interoperability with C	587
Platform libraries	587
Simple example	587
Create bindings for a new library	588
Bindings	590
Mapping primitive data types from C – tutorial	597
Types in C language	597
Example C library	597
Inspect generated Kotlin APIs for a C library	598
Primitive types in kotlin	600
Fix the code	601
Next steps	602
Mapping struct and union types from C – tutorial	602
Mapping struct and union C types	602
Inspect Generated Kotlin APIs for a C library	603

Primitive types in Kotlin	605
Use struct and union types from Kotlin	606
Run the code	608
Next steps	609
Mapping function pointers from C – tutorial	609
Mapping function pointer types from C	609
Inspect generated Kotlin APIs for a C library	610
C function pointers in Kotlin	612
Pass Kotlin function as C function pointer	613
Use the C function pointer from Kotlin	613
Fix the code	613
Next Steps	614
Mapping Strings from C – tutorial	614
Working with C strings	614
Inspect generated Kotlin APIs for a C library	615
Primitive types in Kotlin	617
Pass Kotlin string to C	618
Read C Strings in Kotlin	618
Receive C string bytes from Kotlin	619
Fix the Code	619
Next steps	620
Using C Interop and libcurl for an app – tutorial	620
Generate bindings	620
Consume the Kotlin API	623
Compile and link the library	624
Interoperability with Swift/Objective-C	625
Usage	625
Mappings	626

Casting between mapped types	632
Subclassing	632
C features	633
Unsupported	633
CocoaPods integration	633
Install the CocoaPods dependency manager and plugin	633
Add dependencies on Pod libraries	634
Use a Kotlin Gradle project as a CocoaPods dependency	644
Kotlin/Native as an Apple framework – tutorial	646
Create a Kotlin library	646
Generated framework headers	649
Garbage collection and reference counting	653
Use the code from Objective-C	653
Use the code from Swift	653
Xcode and framework dependencies	654
Next steps	655
Debugging Kotlin/Native	655
Produce binaries with debug info with Kotlin/Native compiler	655
Breakpoints	656
Stepping	657
Variable inspection	657
Known issues	659
Symbolicating iOS crash reports	659
Producing .dSYM for release Kotlin binaries	659
Make frameworks static when using rebuild from bitcode	660
Decode inlined stack frames	661
Tips for improving Kotlin/Native compilation times	661
General recommendations	661

Gradle configuration	662
Kotlin/Native FAQ	663
How do I run my program?	663
What is Kotlin/Native memory management model?	663
How do I create a shared library?	663
How do I create a static library or an object file?	664
How do I run Kotlin/Native behind a corporate proxy?	664
How do I specify a custom Objective-C prefix/name for my Kotlin framework?	664
How do I rename the iOS framework?	665
How do I enable bitcode for my Kotlin framework?	665
Why do I see InvalidMutabilityException?	665
How do I make a singleton object mutable?	666
How can I compile my project with unreleased versions of Kotlin/Native?	666
Kotlin releases	666
Update to a new release	666
IDE support	667
Release details	667
Kotlin plugin releases	673
Update to a new release	673
Release details	674
Kotlin roadmap	674
Key priorities	675
Kotlin roadmap by subsystem	675
What's changed since January 2021	679
Collections overview	681
Collection types	681
Constructing collections	687
Construct from elements	687

Empty collections	687
Initializer functions for lists	687
Concrete type constructors	688
Copy	688
Invoke functions on other collections	689
Iterators	690
List iterators	691
Mutable iterators	692
Ranges and progressions	692
Range	693
Progression	694
Sequences	695
Construct	696
Sequence operations	697
Sequence processing example	698
Collection operations overview	699
Extension and member functions	700
Common operations	700
Write operations	701
Collection transformation operations	702
Map	702
Zip	703
Associate	704
Flatten	705
String representation	706
Filtering collections	707
Filter by predicate	707
Partition	709

Test predicates	709
Plus and minus operators	710
Grouping	710
Retrieve collection parts	711
Slice	711
Take and drop	712
Chunked	713
Windowed	713
Retrieve single elements	714
Retrieve by position	715
Retrieve by condition	716
Retrieve with selector	716
Random element	717
Check element existence	717
Ordering	718
Natural order	719
Custom orders	720
Reverse order	720
Random order	721
Aggregate operations	721
Fold and reduce	723
Collection write operations	725
Adding elements	725
Removing elements	726
Updating elements	727
List-specific operations	727
Retrieve elements by index	727

Retrieve list parts	728
Find element positions	728
List write operations	730
Set-specific operations	733
Map-specific operations	733
Retrieve keys and values	733
Filter	734
Plus and minus operators	735
Map write operations	735
Scope functions	738
Function selection	739
Distinctions	740
Functions	743
takeIf and takeUnless	747
Opt-in requirements	748
Opt in to using API	749
Require opt-in for API	752
Opt-in requirements for pre-stable APIs	753
Experimental status of the opt-in requirements	753
Coroutines guide	753
Table of contents	754
Additional references	754
Asynchronous programming techniques	755
Threading	755
Callbacks	756
Futures, promises, and others	756
Reactive extensions	757
Coroutines	757

Coroutines basics	758
Your first coroutine	758
Extract function refactoring	760
Scope builder	760
Scope builder and concurrency	761
An explicit job	762
Coroutines ARE light-weight	763
Cancellation and timeouts	763
Cancelling coroutine execution	763
Cancellation is cooperative	764
Making computation code cancellable	765
Closing resources with finally	766
Run non-cancellable block	766
Timeout	767
Asynchronous timeout and resources	769
Composing suspending functions	770
Sequential by default	770
Concurrent using <code>async</code>	772
Lazily started <code>async</code>	773
<code>Async</code> -style functions	774
Structured concurrency with <code>async</code>	775
Coroutine context and dispatchers	777
Dispatchers and threads	777
Unconfined vs confined dispatcher	778
Debugging coroutines and threads	779
Jumping between threads	781
Job in the context	782
Children of a coroutine	783

Parental responsibilities	784
Naming coroutines for debugging	784
Combining context elements	785
Coroutine scope	786
Asynchronous Flow	789
Representing multiple values	789
Flows are cold	792
Flow cancellation basics	793
Flow builders	793
Intermediate flow operators	794
Terminal flow operators	796
Flows are sequential	797
Flow context	798
Buffering	801
Composing multiple flows	804
Flattening flows	806
Flow exceptions	809
Exception transparency	811
Flow completion	813
Imperative versus declarative	816
Launching flow	816
Flow and Reactive Streams	819
Channels	820
Channel basics	820
Closing and iteration over channels	821
Building channel producers	821
Pipelines	822
Prime numbers with pipeline	823
Fan-out	824

Fan-in	826
Buffered channels	827
Channels are fair	828
Ticker channels	829
Coroutine exceptions handling	830
Exception propagation	830
CoroutineExceptionHandler	831
Cancellation and exceptions	832
Exceptions aggregation	834
Supervision	836
Shared mutable state and concurrency	839
The problem	839
Volatiles are of no help	840
Thread-safe data structures	841
Thread confinement fine-grained	842
Thread confinement coarse-grained	843
Mutual exclusion	844
Actors	845
Select expression (experimental)	847
Selecting from channels	847
Selecting on close	849
Selecting to send	850
Selecting deferred values	852
Switch over a channel of deferred values	853
Debug coroutines using IntelliJ IDEA – tutorial	855
Create coroutines	855
Debug coroutines	856
Debug Kotlin Flow using IntelliJ IDEA – tutorial	858

Create a Kotlin flow	858
Debug the coroutine	860
Add a concurrently running coroutine	862
Debug a Kotlin flow with two coroutines	862
Serialization	863
Libraries	864
Formats	864
Example: JSON serialization	864
Keywords and operators	866
Hard keywords	866
Soft keywords	868
Modifier keywords	869
Special identifiers	870
Operators and special symbols	870
Gradle	872
Plugin and versions	872
Targeting multiple platforms	872
Targeting the JVM	873
Targeting JavaScript	874
Targeting Android	875
Configuring dependencies	875
Annotation processing	880
Incremental compilation	880
Gradle build cache support	881
Gradle configuration cache support	881
Compiler options	881
Generating documentation	885
OSGi	885
Using Gradle Kotlin DSL	885

Maven	885
Plugin and versions	885
Dependencies	886
Compile Kotlin-only source code	886
Compile Kotlin and Java sources	887
Incremental compilation	888
Annotation processing	889
Jar file	889
Self-contained Jar file	889
Specifying compiler options	890
Generating documentation	892
OSGi	892
Examples	892
Ant	892
Getting the Ant tasks	892
Targeting JVM with Kotlin-only source	893
Targeting JVM with Kotlin-only source and multiple roots	893
Targeting JVM with Kotlin and Java source	893
Targeting JavaScript with single source folder	894
Targeting JavaScript with Prefix, PostFix and sourcemap options	894
Targeting JavaScript with single source folder and metaInfo option	894
References	895
Run code snippets – tutorial	896
Scratches and worksheets	896
REPL	899
Migrate to Kotlin code style	900
Kotlin coding conventions and IntelliJ IDEA formatter	900
Differences between "Kotlin coding conventions" and "IntelliJ IDEA default code style"	901

Migration to a new code style discussion	901
Migration to a new code style	902
Store old code style in project	903
Eclipse IDE	903
Set up the environment	904
Create a new project	906
Run the application	910
Kotlin command-line compiler	912
Install the compiler	912
Create and run an application	913
Compile a library	914
Run the REPL	914
Run scripts	914
Kotlin compiler options	915
Compiler options	915
Common options	916
Kotlin/JVM compiler options	917
Kotlin/JS compiler options	919
Kotlin/Native compiler options	920
All-open compiler plugin	923
Gradle	923
Maven	924
Spring support	925
Command-line compiler	926
No-arg compiler plugin	926
Gradle	926
Maven	927
JPA support	928

Command-line compiler	928
SAM-with-receiver compiler plugin	928
Gradle	929
Maven	929
Command-line compiler	930
Using kapt	930
Using in Gradle	930
Annotation processor arguments	931
Gradle build cache support	931
Improving the speed of builds that use kapt	932
Compile avoidance for kapt	933
Incremental annotation processing	933
Java compiler options	933
Non-existent type correction	934
Using in Maven	934
Using in CLI	934
Generating Kotlin sources	935
AP/Javac options encoding	936
Keeping Java compiler's annotation processors	936
Lombok compiler plugin	936
Supported annotations	937
Gradle	937
Maven	938
Using with kapt	939
Kotlin and continuous integration with TeamCity	940
Gradle, Maven, and Ant	940
IntelliJ IDEA Build System	941
Other CI servers	942

Document Kotlin code: KDoc and Dokka	943
Generate the documentation	943
KDoc syntax	943
Inline markup	945
Module and package documentation	945
Kotlin and OSGi	946
Maven	946
Gradle	947
FAQ	947
Learning materials overview	947
Kotlin Koans	948
Kotlin books	948
Learning Kotlin with EduTools plugin	952
Teaching Kotlin with EduTools plugin	952
FAQ	952
What is Kotlin?	953
What is the current version of Kotlin?	953
Is Kotlin free?	953
Is Kotlin an object-oriented language or a functional one?	953
What advantages does Kotlin give me over the Java programming language?	953
Is Kotlin compatible with the Java programming language?	953
What can I use Kotlin for?	953
Can I use Kotlin for Android development?	954
Can I use Kotlin for server-side development?	954
Can I use Kotlin for web development?	954
Can I use Kotlin for desktop development?	954
Can I use Kotlin for native development?	954
What IDEs support Kotlin?	954

What build tools support Kotlin?	955
What does Kotlin compile down to?	955
Which versions of JVM does Kotlin target?	955
Is Kotlin hard?	955
What companies are using Kotlin?	955
Who develops Kotlin?	955
Where can I learn more about Kotlin?	955
Are there any books on Kotlin?	956
Are there any online courses available for Kotlin?	956
Does Kotlin have a community?	956
Are there Kotlin events?	956
Is there a Kotlin conference?	956
Is Kotlin on social media?	956
Any other online Kotlin resources?	956
Where can I get an HD Kotlin logo?	957
Participate in the Kotlin Early Access Preview	957
Build details	957
Install the EAP Plugin for IntelliJ IDEA or Android Studio	958
Configure your build for EAP	959
Configure in Gradle	960
Configure in Maven	961
Contribution	962
Participate in Early Access Preview	962
Contribute to the compiler, standard library, and tooling	962
Contribute to other libraries and create your own	963
Contribute to the documentation	963
Create tutorials or videos	963
Translate documentation to other languages	963

Hold events and presentations	963
Kotlin Evolution	964
Principles of Pragmatic Evolution	964
Incompatible changes	965
Decision making	966
Feature releases and incremental releases	966
Libraries	967
Compiler keys	967
Compatibility tools	968
Stability of Kotlin components	969
Stability levels explained	969
Stability of subcomponents	969
Current stability of Kotlin components	970
Stability of Kotlin components (pre 1.4)	971
Compatibility guide for Kotlin 1.5	972
Basic terms	973
Language and stdlib	973
Tools	982
Compatibility guide for Kotlin 1.4	983
Basic terms	984
Language and stdlib	984
Tools	1004
Compatibility guide for Kotlin 1.3	1005
Basic terms	1006
Incompatible changes	1006
Compatibility modes	1018
Binary compatibility warnings	1018

Kotlin Foundation	1019
Scope	1019
Structure	1020
Current personnel	1020
Language Committee guidelines	1021
Review process	1021
Scope	1022
Migration aids and deprecation	1022
Changes to these guidelines	1024
Appendix A. Examples of non-issues	1024
Appendix B. Assumptions for pragmatic language evolution	1025
Guide to submitting incompatible changes	1026
Issue template	1026
Classification	1026
Kotlin brand usage guidelines	1029
General goals	1030
Permitted uses	1030
Unpermitted uses	1030
Usage guidelines	1030
Example usages	1031
Kotlin Foundation FAQ	1031
How can I contribute or provide feedback to Kotlin?	1031
How can I get more involved with the Kotlin community?	1032
How is the Kotlin Foundation governed?	1032
Do JetBrains and Google contribute financially to the foundation?	1032
Does the foundation generate any revenue?	1032
Can I license the Kotlin trademark?	1032
How can a company or an individual join the foundation?	1032

What resources should I follow to stay up-to-date on Kotlin?	1033
Security	1033
Kotlin documentation as PDF	1033

Kotlin docs

Get started with Kotlin

You can get started with Kotlin using an online editor. If you already have an IDE or you are ready to install one, here are also some ways to begin using Kotlin on your local machine.

- [Get started with Kotlin](#)
- [Try Kotlin online](#)
- [Get started with KMM in Android Studio](#)

What's new in Kotlin

Discover what features are available in the newest Kotlin release. You can try out upcoming features in the [preview versions](#) before they are released.

- [What's new in Kotlin 1.5.20](#)
- [What's new in Kotlin plugin 2021.1](#)
- [Kotlin public roadmap](#)
- [KMM plugin releases for Android Studio](#)

Basics & concepts

Acquaint yourself with some of the concepts and the basics of Kotlin.

- [Basic syntax](#)
- [Basic types](#)
- [Control flow](#)
- [Null safety](#)
- [Coroutines](#)
- [Coding conventions](#)

Ways to learn

Find a way to learn Kotlin that works for you. There are lots of learning materials available from our team and other authors.

- [Kotlin by example](#)
- [Kotlin Koans](#)
- [Hands-on tutorials](#)
- [EduTools in IntelliJ IDEA](#)
- [Books](#)

Kotlin YouTube Channel

Our [YouTube channel](#) is packed with resources for learning Kotlin! [Subscribe](#) to stay updated for shows, the latest news, and live events!

- [Kotlin in Spring Framework](#)
- [Webinars with Experts](#)
- [Kotlin Multiplatform Multiverse](#)
- [Competitive Programming](#)
- [Kotlin Standard Library](#)
- [Talking Kotlin Podcast](#)
- [Kotlin for Educators](#)
- [Kotlin Online Event 2020 playlist](#)

Stay in touch and contribute

If you are interested in what's going with Kotlin, join us on social media, and get involved.

- [Contribute to Kotlin](#)
- [Participate in Early Access Program](#)
- [Join Kotlin Slack](#)
- [Follow Kotlin on Twitter](#)
- [Chat on Reddit](#)
- [Participate in Stack Overflow discussions](#)

Get started with Kotlin

Kotlin is a modern but already mature programming language aimed 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.

Pick it up to start building powerful applications!

Learn Kotlin fundamentals

- If you're already familiar with one or more programming languages and want to learn Kotlin, start with these [Kotlin learning materials](#).
- If you are at the very beginning and Kotlin is your first programming language, you can start with reading the [Atomic Kotlin book](#) or finding a Kotlin course.

Create your powerful application with Kotlin

Backend app

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

1. Install the [latest version of IntelliJ IDEA](#).

2. 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

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](#)

Ktor

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](#)

3. 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](#)

4. Learn more about Kotlin for server-side:

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

5. 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.

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](#).

Cross-platform mobile app

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

1. [Set up your environment for cross-platform development](#)

2. Create your first KMM application:

- To start from scratch, [create a basic KMM 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 [hands-on tutorial](#) or any [KMM sample](#).

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

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

4. Learn more about KMM:

- Learn more about [Kotlin Multiplatform](#).
- Look through [KMM samples on GitHub](#).
- [Create and publish a multiplatform library](#).
- Learn how KMM is used at [Netflix](#), [VWWare](#), [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](#).

Frontend web app

Kotlin provides an ability to transpile your Kotlin code, the Kotlin standard library, and any compatible dependencies to JavaScript.

Here you'll learn how to develop and improve your frontend web application using [Kotlin/JS](#).

1. Install the [latest version of IntelliJ IDEA](#).
2. Create your first frontend web application:

- To start from scratch, [create a basic browser application with the IntelliJ IDEA project wizard](#).
- If you prefer more robust examples, complete the [Building Web Applications with React and Kotlin/JS](#) hands-on tutorial. It includes a sample project that can serve as a good starting point for your own projects, and contains useful snippets and

templates.

3. Use libraries in your application. Learn more about [adding dependencies](#).

Library	Details
stdlib	The Kotlin standard library included in all projects by default.
kotlinx.browser	The Kotlin library for accessing browser-specific functionality, including typical top-level objects such as document and window.
kotlinx.html	The Kotlin library for generating DOM elements using statically-typed HTML builders.
Ktor	The Kotlin multiplatform library for networking.
KVision	A third-party object-oriented web framework for Kotlin/JS.
fritz2	A third-party lightweight, high-performance, independent library for building reactive web apps in Kotlin that are heavily dependent on coroutines and flows.
Doodle	A third-party vector-based UI framework that uses browser's capabilities to draw user interfaces.
Compose for Web	Brings Google's Jetpack Compose UI toolkit to the browser. A Technology Preview is currently available.
kotlin-wrappers	Provide convenient abstractions and deep integrations for one of the most popular JavaScript frameworks. Kotlin wrappers also provide support for a number of adjacent technologies like react-redux, react-router, or styled-components.

4. Learn more about Kotlin for frontend web development:

- The [new Kotlin/JS IR compiler](#) (currently with [Alpha](#) stability).
- [Using dependencies from npm](#).
- [Using Kotlin code from JavaScript](#).

5. Join the Kotlin frontend web community:

-  Slack: get an invite and join the [#getting-started](#) and [#javascript](#) channels.
-  StackOverflow: subscribe to the “[kotlin-js](#)” 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 app

- 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.

Multiplatform library

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.

Here you'll learn how to develop and publish a multiplatform library:

1. Install the [latest version of IntelliJ IDEA](#).
2. Create a multiplatform library:
 - To start from scratch, [create a basic project](#).
 - If you prefer more robust examples, complete the [Create and publish a multiplatform library](#) tutorial. It shows how to create a multiplatform library for JVM, JS, and Native platforms, test it and publish to a local Maven repository.
 - Build a full stack web application using [this hands-on](#).
3. Use libraries in your application. Learn more about [adding dependencies on libraries](#).

Library	Details
Ktor	Docs and sample .
Serialization	Docs and sample .
Coroutines	Docs .
DateTime	Docs .

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

4. Learn more about Kotlin Multiplatform programming:
 - [Introduction to Kotlin Multiplatform](#).
 - [Kotlin Multiplatform supported platforms](#).
 - [Kotlin Multiplatform programming benefits](#).
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](#).

Is anything missing?

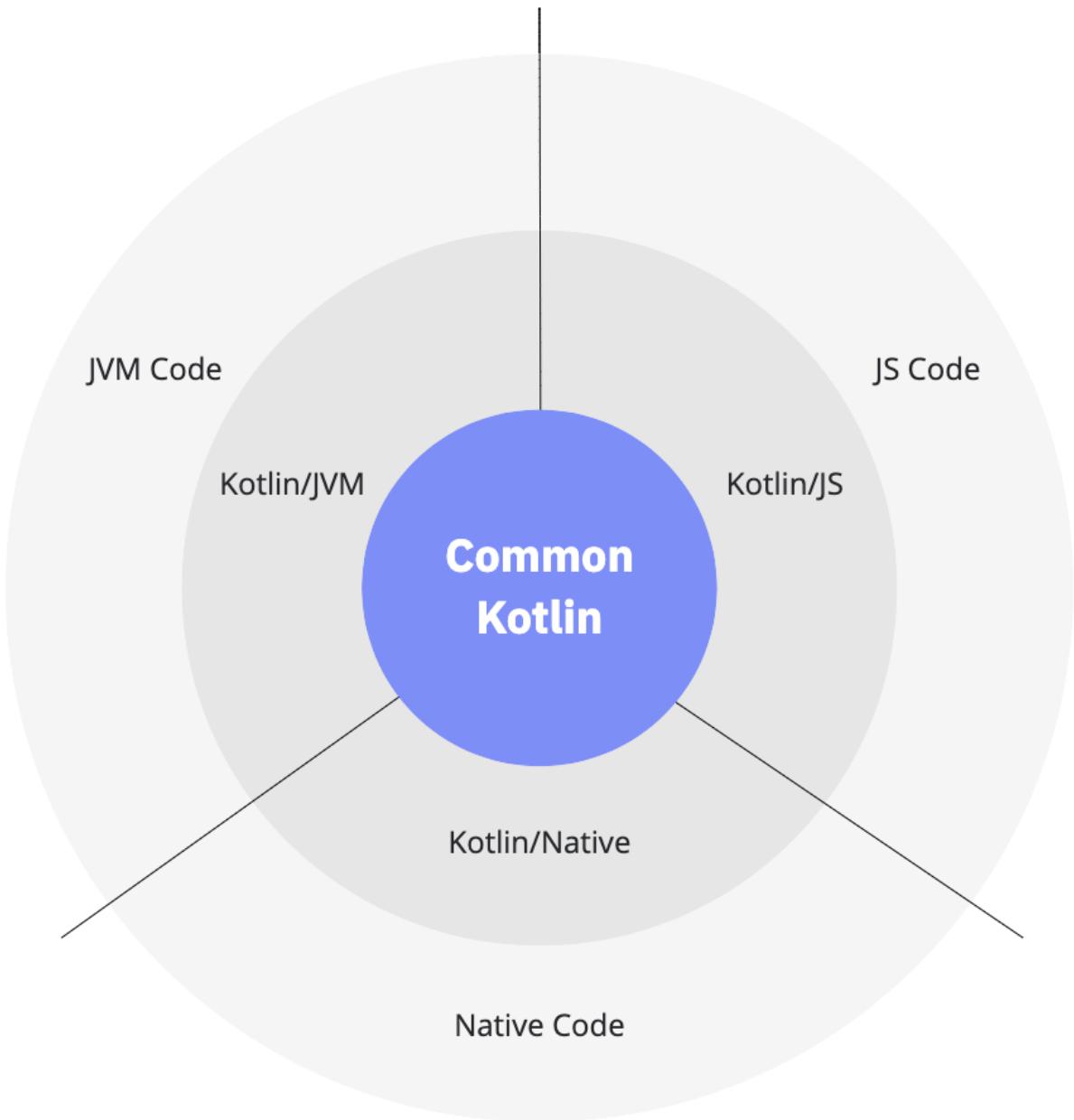
If anything is missing or seems confusing on this page, please [share your feedback](#).

Multiplatform programming

Multiplatform projects are in [Alpha](#). Language features and tooling may change in future Kotlin versions.

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.

This is how Kotlin Multiplatform works.



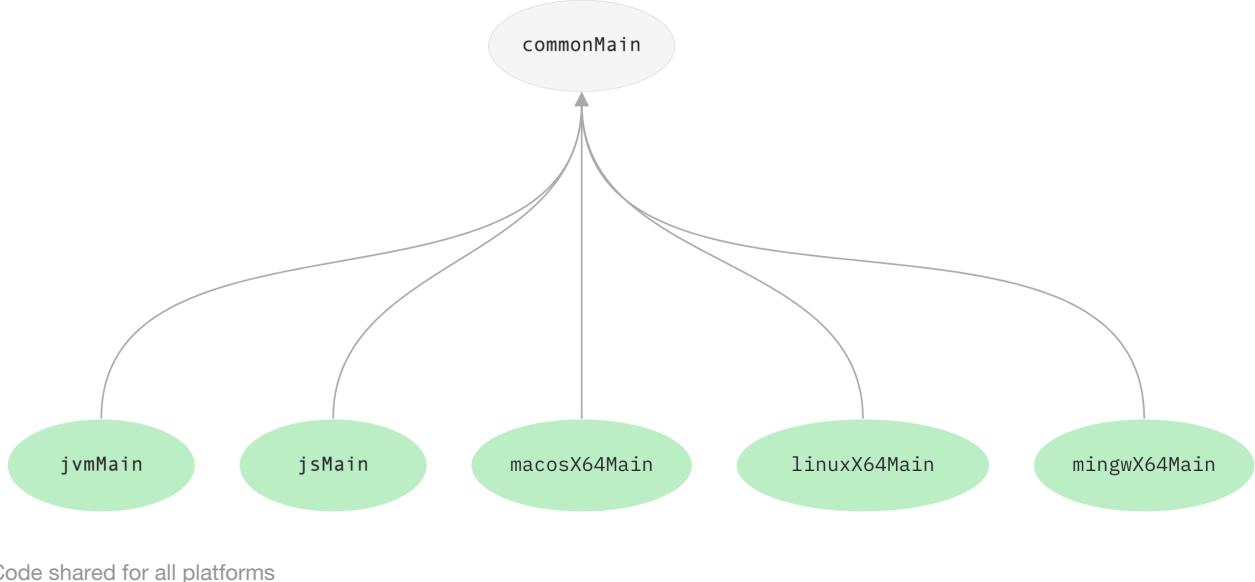
Kotlin Multiplatform

- Common Kotlin includes the language, core libraries, and basic tools. Code written in common Kotlin works everywhere on all platforms.
- With Kotlin Multiplatform libraries, you can reuse the multiplatform logic in common and platform-specific code. Common code can rely on a set of libraries that cover everyday tasks such as HTTP, serialization, and managing coroutines.
- To interop with platforms, use platform-specific versions of Kotlin. Platform-specific versions of Kotlin (Kotlin/JVM, Kotlin/JS, Kotlin/Native) include extensions to the Kotlin language, and platform-specific libraries and tools.

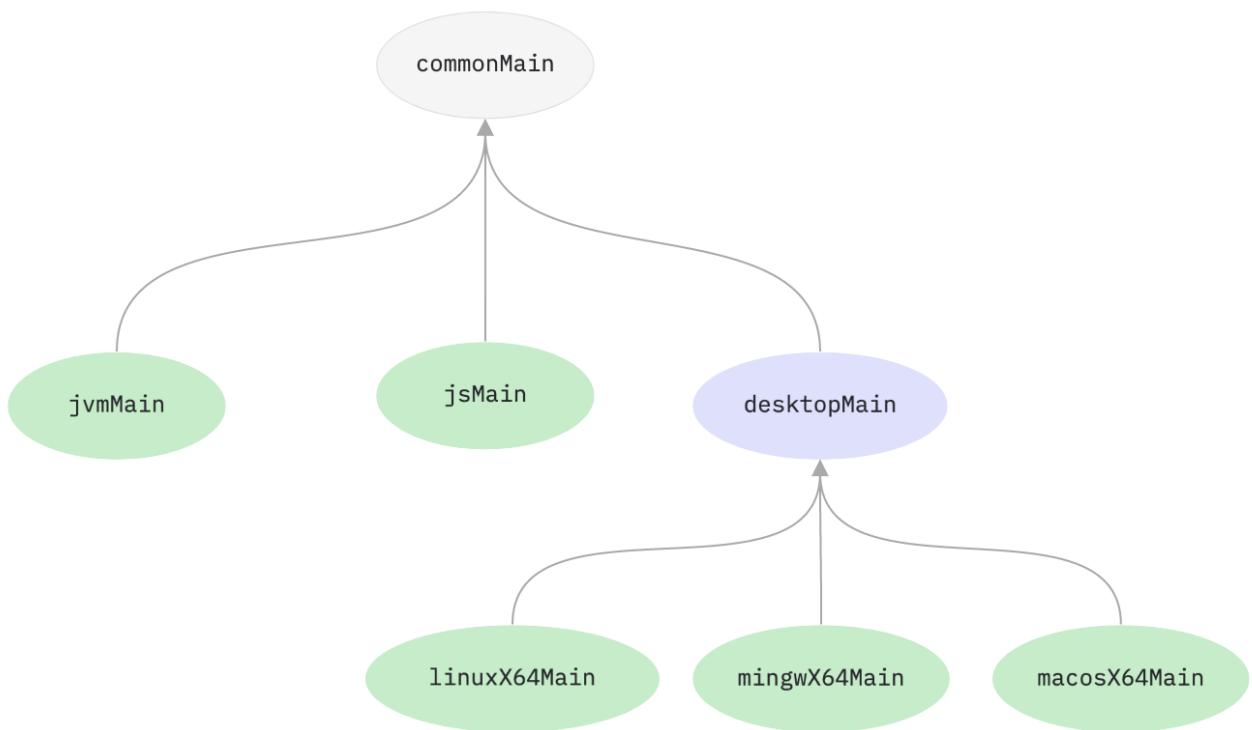
- Through these platforms you can access the platform native code (JVM, JS, and Native) and leverage all native capabilities.

With Kotlin Multiplatform, spend less time on writing and maintaining the same code for different platforms – just share it using the mechanisms Kotlin provides:

- Share code among all platforms used in your project. Use it for sharing the common business logic that applies to all platforms.



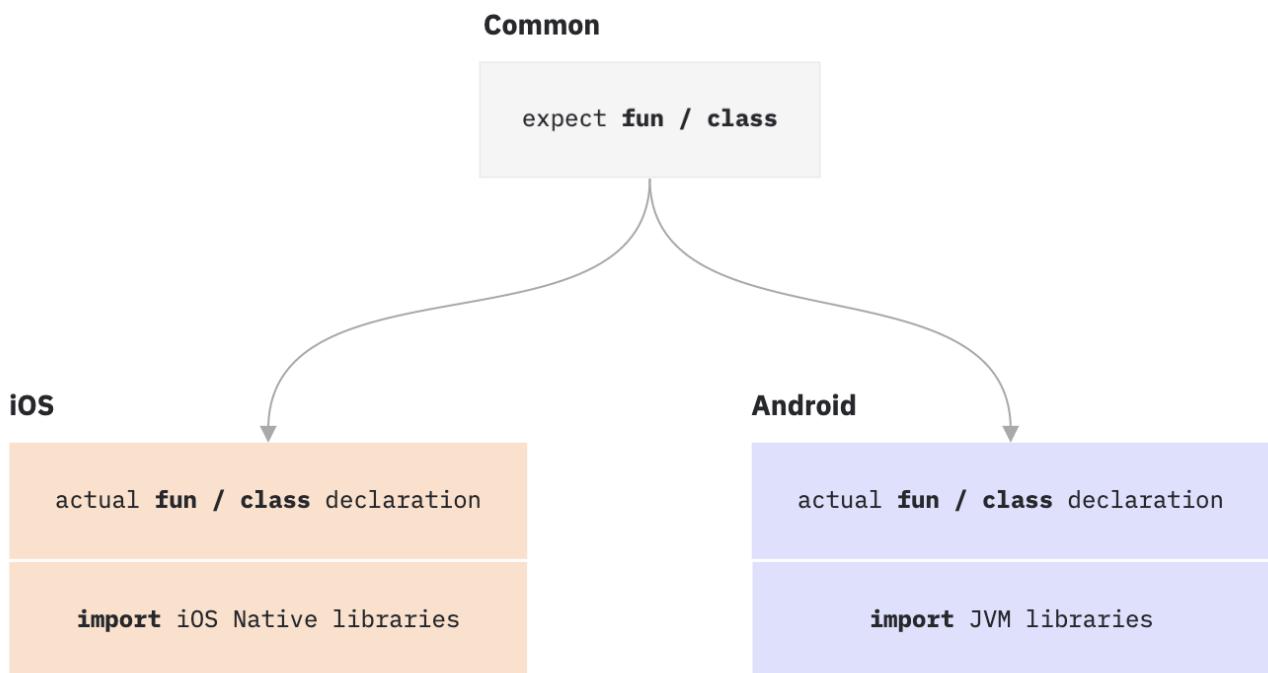
- Share code among some platforms included in your project but not all. Do this when you can reuse much of the code in similar platforms.



Hierarchical structure

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

With this mechanism, a common source set defines an expected declaration, and platform source sets must provide the actual declaration that corresponds to the expected declaration. This works for most Kotlin declarations, such as functions, classes, interfaces, enumerations, properties, and annotations.



Expect and actual declarations

```
//Common
expect fun randomUUID(): String
```

```
//Android
import java.util.*
actual fun randomUUID() = UUID.randomUUID().toString()
```

```
//iOS
import platform.Foundation.NSUUID
actual fun randomUUID(): String = NSUUID().UUIDString()
```

Use cases

Android – iOS

Sharing code between mobile platforms is one of the major Kotlin Multiplatform use cases. With Kotlin Multiplatform Mobile (KMM), you can build multiplatform mobile applications sharing code, such as business logic, connectivity, and more, between Android and iOS.

See [KMM features, case studies and examples](#)

Client – Server

Another scenario when code sharing may bring benefits is a connected application where the logic can be reused on both the server and the client side running in the browser. This is covered by Kotlin Multiplatform as well.

The [Ktor framework](#) is suitable for building asynchronous servers and clients in connected systems.

What's next?

New to Kotlin? Visit [Getting started with Kotlin](#).

Documentation

- [Get started with Kotlin Multiplatform Mobile \(KMM\)](#)
- [Create a multiplatform project](#)
- [Share code on multiple platforms](#)
- [Connect to platform-specific APIs](#)

Tutorials

- [Creating a KMM application](#) shows how to create a mobile application that works on Android and iOS with the help of the [KMM plugin for Android Studio](#). Create, run, and test your first multiplatform mobile application.
- [Creating a multiplatform Kotlin library](#) teaches how to create a multiplatform library available for JVM, JS, and Native and which can be used from any other common code (for example, shared with Android and iOS). It also shows how to write tests which will be executed on all platforms and use an efficient implementation provided by a specific platform.
- [Building a full stack web app with Kotlin Multiplatform](#) teaches the concepts behind building an application that targets Kotlin/JVM and Kotlin/JS by building a client-server application that makes use of shared code, serialization, and other multiplatform paradigms. It also provides a brief introduction to working with Ktor both as a server- and client-side framework.

Sample projects

- [Kotlin Multiplatform Mobile samples](#)
- [KotlinConf app](#)
- [KotlinConf Spinner app](#)

Kotlin for server side

Kotlin is a great fit for developing server-side applications, allowing you to write concise and expressive code while maintaining full compatibility with existing Java-based technology stacks and 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, which lets you stay on your familiar technology stack while reaping the benefits of a more modern language.
- Migration: Kotlin supports gradual, step by step 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](#) offer a guide through the key features of the language with a series of interactive exercises.

Frameworks for server-side development with 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.
- [Vert.x](#), a framework for building reactive Web applications on the JVM, offers [dedicated support](#) for Kotlin, including [full documentation](#).
- [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.
- [kotlinx.html](#) is a DSL that can be used to build HTML in a Web application. 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 microservice and serverless applications. It comes with a lot of built-in, handy 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, as well as using NoSQL databases through their Java drivers. For JPA, the [kotlin-jpa compiler plugin](#) adapts Kotlin-compiled classes to the requirements of the framework.

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.

Users of Kotlin on the server side

[Corda](#) is an open-source distributed ledger platform, 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](#).
- Micronaut also has a lot of well-detailed [guides](#), showing how you can build microservices in Kotlin.
- 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.

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.

- 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 in Android Studio and is actively used by many companies for developing Android applications.
- Kotlin support in Android Jetpack and other libraries. [KTX extensions](#) add Kotlin language features, such as coroutines, extension functions, lambdas, and named parameters, to existing Android libraries.
- 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.
- Support for multiplatform development. You can use Kotlin for developing not only Android but also [iOS](#), backend, and web applications. Enjoy the benefits of sharing the common code among the platforms.
- Code safety. Less code and better readability lead to fewer errors. The Kotlin compiler detects these remaining errors, making the code safe.
- 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. According to Google, over 60% of the top 1000 apps on the Play Store use Kotlin.

Many startups and Fortune 500 companies have already developed Android applications using Kotlin – see the list at [the Google website for Kotlin developers](#).

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](#).

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.js` and `kotlin.multiplatform` Gradle plugins. They provide a central and convenient way to set up and control Kotlin projects targeting JavaScript. 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 the [Kotlin/JS project setup](#) documentation.

Use cases for Kotlin/JS

There are numerous ways that Kotlin/JS can be used. To provide you some inspiration, here's a non-exhaustive list of scenarios in which you can use Kotlin/JS.

- Write frontend web applications using Kotlin/JS
 - Kotlin/JS allows you to leverage powerful browser and web APIs in a type-safe fashion. Create, modify and

interact with elements in the Document Object Model (DOM), use Kotlin code to control the rendering of canvas or WebGL components, and enjoy access to many more of the features supported in modern browsers.

- Write full, type-safe React applications with Kotlin/JS using the [kotlin-wrappers](#) provided by JetBrains, which provide convenient abstractions and deep integrations for one of the most popular JavaScript frameworks. kotlin-wrappers also provides support for a select number of adjacent technologies like react-redux, react-router, or styled-components. Interoperability with the JavaScript ecosystem also means that you can also use third-party React components and component libraries.
- Use the [Kotlin/JS frameworks](#) that take full advantage of Kotlin concepts, its expressive power and conciseness.
- Write server-side and serverless applications using Kotlin/JS
 - The Node.js target provided by Kotlin/JS enables you to create applications that run on a server or get executed on serverless infrastructure. You benefit from the same advantages as other applications executing in a JavaScript runtime, such as faster startup speed and a reduced memory footprint. With [kotlinx-nodejs](#), you have typesafe access to the [Node.js API](#) directly from your Kotlin code.
- Use Kotlin's [multiplatform](#) projects to share code with other Kotlin targets
 - All Kotlin/JS functionality can also be accessed when using the Kotlin multiplatform Gradle plugin.
 - If you have a backend written in Kotlin, you can share common code such as data models or validation logic with a frontend written in Kotlin/JS, allowing you to write and maintain full-stack web applications.
 - You could also share business logic between your web interface and mobile apps for Android and iOS, and avoid duplicating commonly used functionality like providing abstractions around REST API endpoints, user authentication, or your domain models.
- Create libraries for use with JavaScript and TypeScript
 - You don't have to write your whole application in Kotlin/JS, either – you can also generate libraries from your Kotlin code that can be consumed as modules from any code base written in JavaScript or TypeScript, regardless of other frameworks or technologies used. This approach of creating hybrid applications allows you to leverage the competencies that you and your team might already have around web development, while helping you reduce the amount of duplicated work, and making it easier to keep your web target consistent with other targets of your application.

Of course, this is not a complete list of how you can use Kotlin/JS to your advantage, but merely a selection of cherry-picked cases. We invite you to experiment with combinations of these use cases, and find out what works best for your project.

Regardless of your specific use case, Kotlin/JS projects can use compatible libraries from the Kotlin ecosystem, as well as third-party libraries from the JavaScript and TypeScript ecosystems. To use the latter from Kotlin code, you can either provide your own typesafe wrappers, use community-maintained wrappers, or let [Dukat](#) automatically generate Kotlin declarations for you. Using the Kotlin/JS-exclusive [dynamic type](#) allows you to loosen the constraints of Kotlin's type system, allowing you to skip creating detailed library wrappers - at the expense of type safety.

Kotlin/JS is also compatible with the most common module systems: UMD, CommonJS, and AMD. Being able to

produce and consume modules means that you can interact with the JavaScript ecosystem in a structured manner.

Kotlin/JS frameworks

Modern web development benefits significantly from frameworks that simplify building web applications. Here are 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 <https://kvision.io> for documentation, tutorials, and examples.

For updates and discussions about the framework, join [#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 <https://www.fritz2.dev> 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 <https://nacular.github.io/doodle/> for documentation, tutorials, and examples.

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

Compose for Web

Compose for Web brings [Google's Jetpack Compose UI toolkit](#) to your browser. It allows you to build reactive web user interfaces using the concepts introduced by Jetpack Compose. It provides a DOM API to describe your website, as well as an experimental set of multiplatform layout primitives. Compose for Web also gives you the option to share parts of your UI code and logic across Android, desktop, and web.

Compose for Web is in Technology Preview, which means it is ready for experiments but not yet ready for production use.

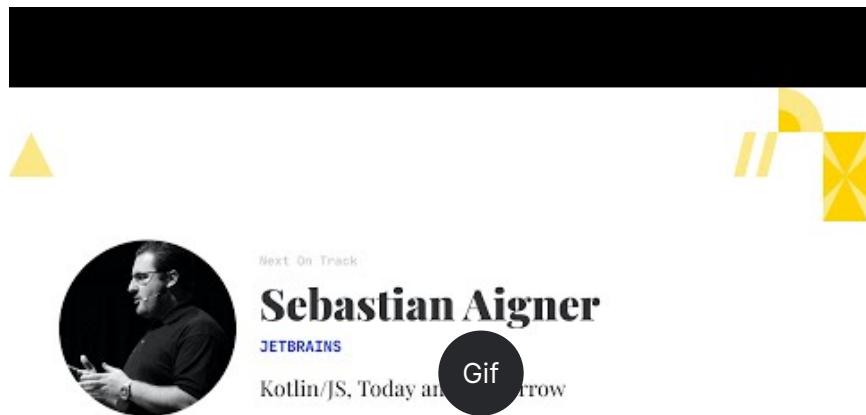
You can find more information about Compose for Web on its [landing page](#).

Join the `#compose-web` channel on [Kotlin Slack](#) to discuss Compose for Web, or `#compose` for general Compose discussions.

Kotlin/JS, Today and Tomorrow

Want to know more about Kotlin/JS?

In [this video](#), Kotlin Developer Advocate Sebastian Aigner will explain the main Kotlin/JS benefits to you, share some tips and use cases, and also tell you about the plans and upcoming features for Kotlin/JS.



[Watch video online.](#)

Get started with Kotlin/JS

If you're new to Kotlin, a good first step would be to familiarize yourself with the [basic syntax](#) of the language.

To start using Kotlin for JavaScript, please refer to the [Set up a Kotlin/JS project](#), or pick a hands-on lab from the next section to work through.

Hands-on labs for Kotlin/JS

Hands-on labs are long-form tutorials that help you get to know a technology by guiding you through a self-contained project related to a specific topic.

They include sample projects, which can serve as jumping-off points for your own projects, and contain useful snippets and patterns.

For Kotlin/JS, the following hands-on labs are currently available:

- [Building Web Applications with React and Kotlin/JS](#) guides you through the process of building a simple web application using the React framework, shows how a typesafe Kotlin DSL for HTML makes it convenient to build reactive DOM elements, and illustrates how to use third-party React components, and how to obtain information from APIs, while writing the whole application logic in pure Kotlin/JS.
- [Building a Full Stack Web App with Kotlin Multiplatform](#) teaches the concepts behind building an application that targets Kotlin/JVM and Kotlin/JS by building a client-server application that makes use of common code, serialization, and other multiplatform paradigms. It also provides a brief introduction into working with Ktor both as a server- and client-side framework.

New Kotlin/JS IR compiler

The [new Kotlin/JS IR compiler](#) (currently with [Alpha](#) stability) comes with a number of improvements over the current default compiler. For example, it improves the size of generated executables via dead code elimination and makes it smoother to interoperate with the JavaScript ecosystem and its tooling. By generating TypeScript declaration files (d.ts) from Kotlin code, the new compiler makes it easier to create “hybrid” applications that mix TypeScript and Kotlin code, and leverage code-sharing functionality using Kotlin Multiplatform.

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

Join the Kotlin/JS community

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

Kotlin Native

Kotlin/Native is a technology for compiling Kotlin code to native binaries, which can run without a virtual machine. It is an [LLVM](#) based backend for the Kotlin compiler and native implementation of the Kotlin standard library.

Why Kotlin/Native?

Kotlin/Native is primarily designed to allow compilation for platforms where virtual machines are not desirable or

possible, for example, embedded devices or iOS. It solves the 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

The full list of supported targets is available [here](#).

Interoperability

Kotlin/Native supports two-way interoperability with the Native world. On the one hand, 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

On the other hand, 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 a compiled Kotlin code into 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 to 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 into the compiler package.

Sharing code between platforms

[Multiplatform projects](#) allow sharing common Kotlin code between multiple platforms, including Android, iOS, JVM,

JavaScript, and native. Multiplatform libraries provide required APIs for the common Kotlin code and help develop shared parts of a project in Kotlin code in one place and share it with all or several target platforms.

You can use [Kotlin Multiplatform Mobile \(KMM\)](#) to create multiplatform mobile applications with code shared between Android and iOS.

How to start

Tutorials and documentation

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

Recommended documentation:

- [Kotlin Multiplatform Mobile documentation](#)
- [Multiplatform documentation](#)
- [C interop](#)
- [Swift/Objective-C interop](#)

Recommended tutorials:

- [Get started with Kotlin/Native](#)
- [Create your first KMM application](#)
- [Types mapping between C and Kotlin/Native](#)
- [Kotlin/Native as a Dynamic Library](#)
- [Kotlin/Native as an Apple Framework](#)

Sample projects

- [Kotlin Multiplatform Mobile samples](#)
- [Kotlin/Native sources and examples](#)
- [KotlinConf app](#)
- [KotlinConf Spinner app](#)
- [Kotlin/Native sources and examples \(.tgz\)](#)
- [Kotlin/Native sources and examples \(.zip\)](#)

Kotlin for data science

From building data pipelines to productionizing machine learning models, Kotlin can be a great choice for working with data:

- Kotlin is concise, readable, and easy to learn.
- Static typing and null safety help create reliable, maintainable code that is easy to troubleshoot.
- Being a JVM language, Kotlin gives you great performance and an ability to leverage an entire ecosystem of tried and true Java libraries.

Interactive editors

Notebooks such as [Jupyter Notebook](#) and [Apache Zeppelin](#) provide convenient tools for data visualization and exploratory research. Kotlin integrates with these tools to help you explore data, share your findings with colleagues, or build up your data science and machine learning skills.

Jupyter Kotlin kernel

The Jupyter Notebook is an open-source web application that allows you to create and share documents (aka "notebooks") that can contain code, visualizations, and markdown text. [Kotlin-jupyter](#) is an open source project that brings Kotlin support to Jupyter Notebook.

The screenshot shows a Jupyter Notebook interface with a Kotlin Kernel Examples notebook. The top navigation bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, and Logout. Below the menu bar are standard Jupyter notebook controls for file operations and cell execution.

In [1]:

```
%use lets-plot, krangl
```

In [2]:

```
val df = DataFrame.readCSV("ramen-ratings.csv")
val processedDF = df.filter{ it["Stars"].isMatching <String>{ !startsWith("Un")  }})
    .addColumn("StarsAsDouble") { it["Stars"].map <String> { it.toDouble()} }
```

In [3]:

```
val distinctBrandsPerCountry = processedDF.groupBy("Country").distinct("Brand").groupBy("Country").count()
val (xs, ys) = distinctBrandsPerCountry.rows.map { row -> (row["Country"] as String) to (row["n"] as Int) }.unzip()
val p = lets_plot(mapOf("country" to xs, "brands" to ys))

val layer = geom_bar(stat=Stat.identity, fill = "#78B3CA") {
    x = "country"
    y = "brands"
}

p + layer
```

Out[3]:

Country	Brands
Japan	58
Taiwan	47
USA	44
South Korea	32
Singapore	6
Hong Kong	10
Vietnam	14
Thailand	22
Malaysia	28
China	22
Indonesia	18
Germany	2
Mexico	1
Hungary	1
Fiji	1
Australia	4
Pakistan	2
Bangladesh	1
Canada	8
Nepal	1
UK	1
Netherlands	4
Finland	1
Sweden	2
Philippines	3
Estonia	1
Poland	1
Dubai	1
Slovakia	1

Kotlin in Jupyter notebook

Check out Kotlin kernel's [GitHub repo](#) for installation instructions, documentation, and examples.

Zeppelin Kotlin interpreter

Apache Zeppelin is a popular web-based solution for interactive data analytics. It provides strong support for the Apache Spark cluster computing system, which is particularly useful for data engineering. Starting from [version 0.9.0](#), Apache Zeppelin comes with bundled Kotlin interpreter.

```

fun square(n: Int): Int = n * n
val greeter = { s: String -> println("Hello $s!") }
val l = listOf("Drive", "to", "develop")

kc.showVars()
kc.showFunctions()

kc: KotlinContext! = org.apache.zeppelin.kotlin.repl.KotlinRepl$KotlinContext@304ec43a
x: Int = 1

```

FINISHED

Kotlin in Zeppelin notebook

Libraries

The ecosystem of libraries for data-related tasks created by the Kotlin community is rapidly expanding. Here are some libraries that you may find useful:

Kotlin libraries

- [Multik](#): multidimensional arrays in Kotlin. The library provides Kotlin-idiomatic, type- and dimension-safe API for mathematical operations over multidimensional arrays. Multik offers swappable JVM and native computational engines, and a combination of the two for optimal performance.
- [KotlinDL](#) is a high-level Deep Learning API written in Kotlin and inspired by Keras. It offers simple APIs for training deep learning models from scratch, importing existing Keras models for inference, and leveraging transfer learning for tweaking existing pre-trained models to your tasks.
- [Kotlin for Apache Spark](#) adds a missing layer of compatibility between Kotlin and Apache Spark. It allows Kotlin developers to use familiar language features such as data classes, and lambda expressions as simple expressions in curly braces or method references.
- [kotlin-statistics](#) is a library providing extension functions for exploratory and production statistics. It supports basic numeric list/sequence/array functions (from sum to skewness), slicing operators (such as countBy, simpleRegressionBy), binning operations, discrete PDF sampling, naive bayes classifier, clustering, linear regression, and much more.
- [kmath](#) is a library inspired by [NumPy](#). This library supports algebraic structures and operations, array-like structures, math expressions, histograms, streaming operations, a wrapper around [commons-math](#) and [koma](#), and more.
- [krangl](#) is a library inspired by R's [dplyr](#) and Python's [pandas](#). This library provides functionality for data manipulation using a functional-style API; it also includes functions for filtering, transforming, aggregating, and reshaping tabular data.
- [lets-plot](#) is a plotting library for statistical data written in Kotlin. Lets-Plot is multiplatform and can be used not only

with JVM, but also with JS and Python.

- [kravis](#) is another library for the visualization of tabular data inspired by R's [ggplot](#).
- [londogard-nlp-toolkit](#) is a library that provides utilities when working with natural language processing such as word/subword/sentence embeddings, word-frequencies, stopwords, stemming, and much more.

Java libraries

Since Kotlin provides first-class interop with Java, you can also use Java libraries for data science in your Kotlin code. Here are some examples of such libraries:

- [DeepLearning4J](#)- a deep learning library for Java
- [ND4J](#)- an efficient matrix math library for JVM
- [Dex](#)- a Java-based data visualization tool
- [Smile](#)- a comprehensive machine learning, natural language processing, linear algebra, graph, interpolation, and visualization system. Besides Java API, Smile also provides a functional [Kotlin API](#) along with Scala and Clojure API.
 - [Smile-NLP-kt](#)- a Kotlin rewrite of the Scala implicits for the natural language processing part of Smile in the format of extension functions and interfaces.
- [Apache Commons Math](#)- a general math, statistics, and machine learning library for Java
- [OptaPlanner](#)- a solver utility for optimization planning problems
- [Charts](#)- a scientific JavaFX charting library in development
- [CoreNLP](#)- a natural language processing toolkit
- [Apache Mahout](#)- a distributed framework for regression, clustering and recommendation
- [Weka](#)- a collection of machine learning algorithms for data mining tasks

If this list doesn't cover your needs, you can find more options in the [Kotlin Data Science Resources](#) digest from Thomas Nield.

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, we 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. We 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, we 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`), etc. 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:

```
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()` function throws an exception if the input string is not an integer.

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:

```
private fun readLn() = readLine()!!
private fun readInt() = readLn().toInt()
// etc 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:

```
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:

```

private fun readLn() = readLine()!! // string line
private fun readInt() = readLn().toInt() // single int
private fun readStrings() = readLn().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 = readLn()
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, etc for indices, r, and c for row and column numbers in tables, x and y for coordinates, etc. 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 10 5 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 10 5 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 web site 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 1.5.20

[Release date: 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:

Groovy

```
kotlin {  
    targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget) {  
        compilations.get("main").kotlinOptions.freeCompilerArgs += "-Xexport-kdoc"  
    }  
}
```

Kotlin

```
kotlin {  
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {  
        compilations.get("main").kotlinOptions.freeCompilerArgs += "-Xexport-kdoc"  
    }  
}
```

We'd 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

Release date: 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:

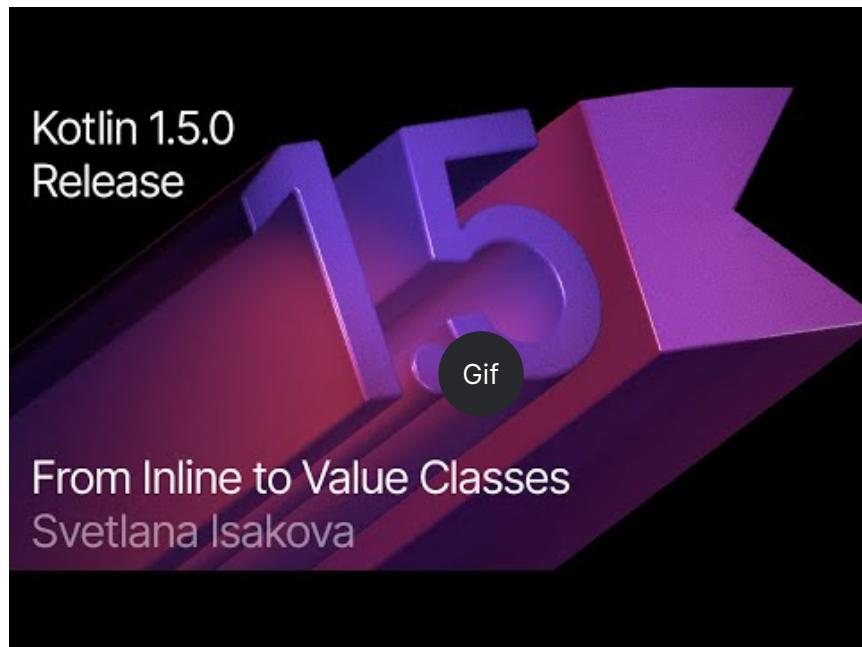
```
value class Password(val s: String)
```

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:

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {  
    kotlinOptions.useOldBackend = true  
}
```

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile> {  
    kotlinOptions.useOldBackend = true  
}
```

- In Maven:

```
<configuration>  
    <args>  
        <arg>-Xuse-old-backend</arg>  
    </args>  
</configuration>
```

New default JVM target: 1.8

The default target version for Kotlin/JVM compilations is now 1.8. The 1.6 target is deprecated.

If you need a build for JVM 1.6, you can still switch to this target. Learn how:

- [in Gradle](#)
- [in Maven](#)
- [in the command-line compiler](#)

SAM adapters via invokedynamic

Kotlin 1.5.0 now uses dynamic invocations (invokedynamic) for compiling SAM (Single Abstract Method) conversions:

- Over any expression if the SAM type is a [Java interface](#)
- Over lambda if the SAM type is a [Kotlin functional interface](#)

The new implementation uses [LambdaMetafactory.metafactory\(\)](#) and auxiliary wrapper classes are no longer generated

during compilation. This decreases the size of the application's JAR, which improves the JVM startup performance.

To roll back to the old implementation scheme based on anonymous class generation, add the compiler option `-Xsam-conversions=class`.

Learn how to add compiler options in [Gradle](#), [Maven](#), and the [command-line compiler](#).

Lambdas via invokedynamic

Compiling plain Kotlin lambdas into invokedynamic is [Experimental](#). It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate hearing your feedback on it in [YouTrack](#).

Kotlin 1.5.0 is introducing experimental support for compiling plain Kotlin lambdas (which are not converted to an instance of a functional interface) into dynamic invocations (invokedynamic). The implementation produces lighter binaries by using [LambdaMetafactory.metafactory\(\)](#), which effectively generates the necessary classes at runtime. Currently, it has three limitations compared to ordinary lambda compilation:

- A lambda compiled into invokedynamic is not serializable.
- Calling `toString()` on such a lambda produces a less readable string representation.
- Experimental [reflect](#) API does not support lambdas created with `LambdaMetafactory`.

To try this feature, add the `-Xlambdas=indy` compiler option. We'd 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.

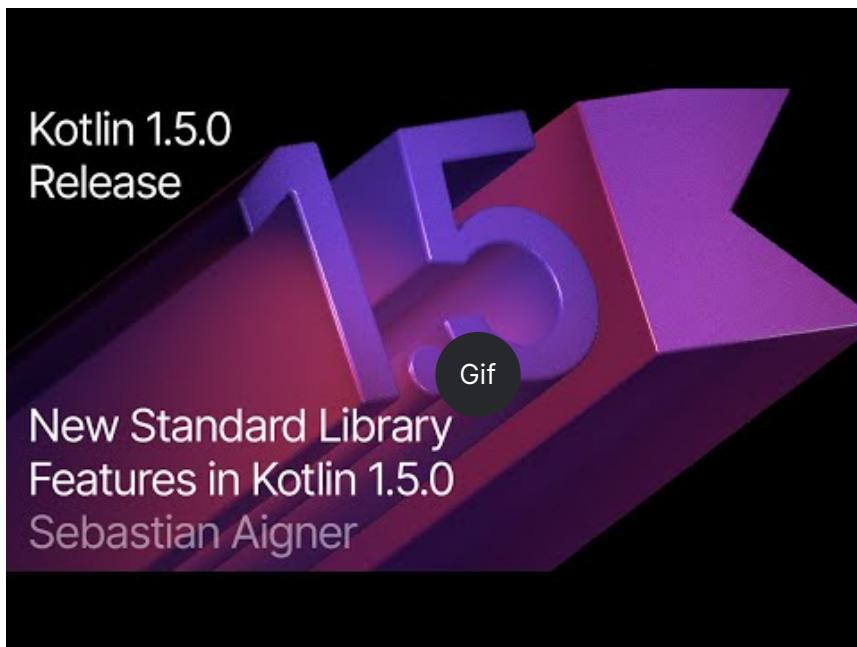
A new [API for getting a char category](#) is now available in multiplatform projects.

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() }`

Earlier versions **1.5.0 alternative**

`String.decapitalize()` `String.replaceFirstChar { it.lowercase() }`

- For Char functions:

Earlier versions **1.5.0 alternative**

`Char.toUpperCase()` `Char.uppercaseChar(): Char`
`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.

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 KEP](#).

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('ż', 'ł', 'ń', 'ż', 'ł', 'ą', 'a', '+')
    val (titleCases, notTitleCases) = chars.partition { it.isTitleCase() }
    println(titleCases) // [ż, ł, ń, ż]
    println(notTitleCases) // [ł, ą, a, +]
//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`:

Groovy

```
kotlin {  
    sourceSets {  
        commonTest {  
            dependencies {  
                implementation kotlin("test") // This brings the dependency  
                // on JUnit 4 transitively  
            }  
        }  
    }  
}
```

Kotlin

```
kotlin {  
    sourceSets {  
        val commonTest by getting {  
            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](#)

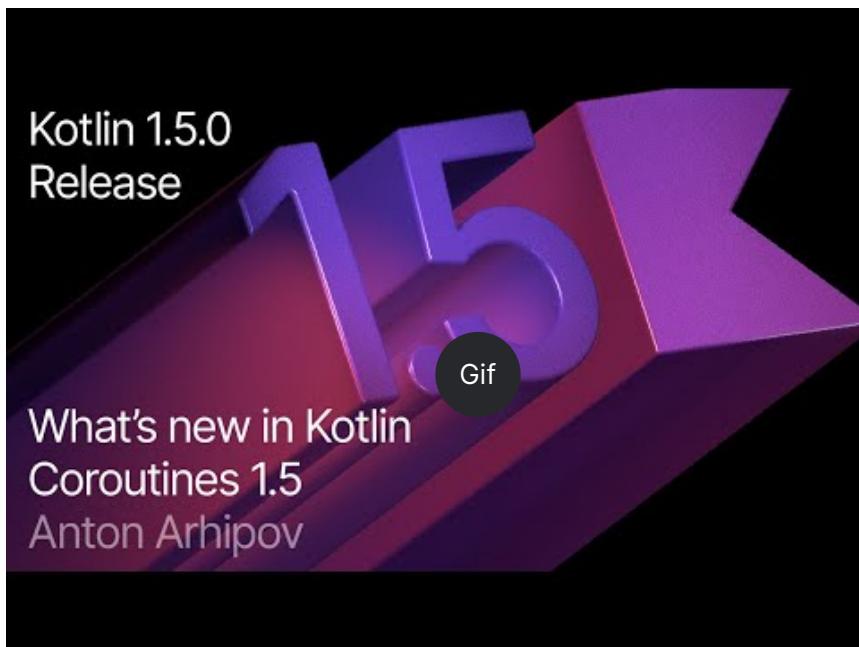
Coroutines 1.5.0-RC

`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](#).



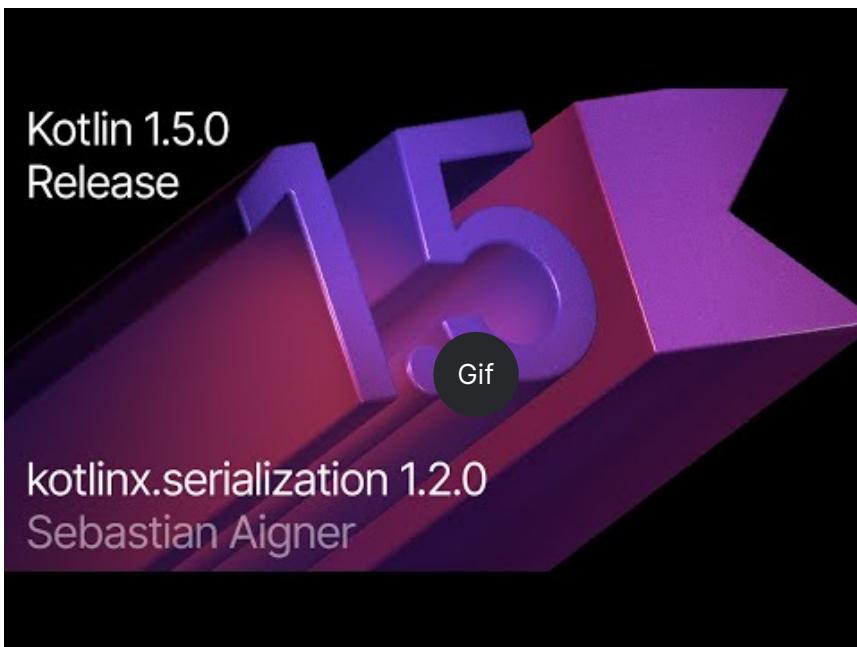
[Watch video online.](#)

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

Release date: 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'd 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'd 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'd 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'd 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:

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {  
    kotlinOptions.useIR = true  
}
```

Kotlin

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile::class) {  
    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 [KMM Networking and Data Storage sample](#) has decreased from 9.5 seconds (in 1.4.10) to 4.5 seconds (in 1.4.30).

Apple watchOS 64-bit simulator target

The x86 simulator target has been deprecated for watchOS since version 7.0. To keep up with the latest watchOS versions, Kotlin/Native has the new target watchosX64 for running the simulator on 64-bit architecture.

Support for Xcode 12.2 libraries

We have added support for the new libraries delivered with Xcode 12.2. You can now use them from Kotlin code.

Kotlin/JS

Lazy initialization of top-level properties

Lazy initialization of top-level properties is [Experimental](#). It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in [YouTrack](#).

The [IR backend](#) for Kotlin/JS is receiving a prototype implementation of lazy initialization for top-level properties. This reduces the need to initialize all top-level properties when the application starts, and it should significantly improve application start-up times.

We'll keep working on the lazy initialization, and we ask you to try the current prototype and share your thoughts and results in this [YouTrack ticket](#) or the [#javascript](#) channel in the official [Kotlin Slack](#) (get an invite [here](#)).

To use the lazy initialization, add the `-Xir-property-lazy-initialization` compiler option when compiling the code with the JS IR compiler.

Gradle project improvements

Support the Gradle configuration cache

Starting with 1.4.30, the Kotlin Gradle plugin supports the [configuration cache](#) feature. It speeds up the build process: once you run the command, Gradle executes the configuration phase and calculates the task graph. Gradle caches the result and reuses it for subsequent builds.

To start using this feature, you can [use the Gradle command](#) or [set up the IntelliJ based IDE](#).

Standard library

Locale-agnostic API for upper/lowercasing text

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

This release introduces the experimental locale-agnostic API for changing the case of strings and characters. The current `toLowerCase()`, `toUpperCase()`, `capitalize()`, `decapitalize()` API functions are locale-sensitive. This means that

different platform locale settings can affect code behavior. For example, in the Turkish locale, when the string “kotlin” is converted using `toUpperCase`, the result is “KOTLİN”, not “KOTLIN”.

```
// current API  
println("Needs to be capitalized".toUpperCase()) // NEEDS TO BE CAPITALIZED  
  
// new API  
println("Needs to be capitalized".uppercase()) // NEEDS TO BE CAPITALIZED
```

Kotlin 1.4.30 provides the following alternatives:

- For String functions:

Earlier versions	1.4.30 alternative
-------------------------	---------------------------

`String.toUpperCase()` `String.uppercase()`

`String.toLowerCase()` `String.lowercase()`

`String.capitalize()` `String.replaceFirstChar { it.uppercase() }`

`String.decapitalize()` `String.replaceFirstChar { it.lowercase() }`

- For Char functions:

Earlier versions	1.4.30 alternative
-------------------------	---------------------------

`Char.toUpperCase()` `Char.uppercaseChar(): Char`
 `Char`
 `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

[Release date: 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.

Learn how to [create Kotlin/JS applications from templates](#).

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 it when the whole application is not ready yet.

There are two tolerance policies for this mode:

- SEMANTIC: the compiler will accept code which is syntactically correct, but doesn't make sense semantically, such as `val x: String = 3`.
- SYNTAX: the compiler will accept any code, even if it contains syntax errors.

To allow compilation with errors, add the `-Xerror-tolerance-policy=` compiler option with one of the values listed above.

Learn more about [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

[Release date: 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:

Groovy

```
kotlin {  
    // for strict mode  
    explicitApi()  
    // or  
    explicitApi = 'strict'  
  
    // for warning mode  
    explicitApiWarning()  
    // or  
    explicitApi = 'warning'  
}
```

Kotlin

```
kotlin {  
    // for strict mode  
    explicitApi()  
    // or  
    explicitApi = ExplicitApiMode.Strict  
  
    // for warning mode  
    explicitApiWarning()  
    // or  
    explicitApi = ExplicitApiMode.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.

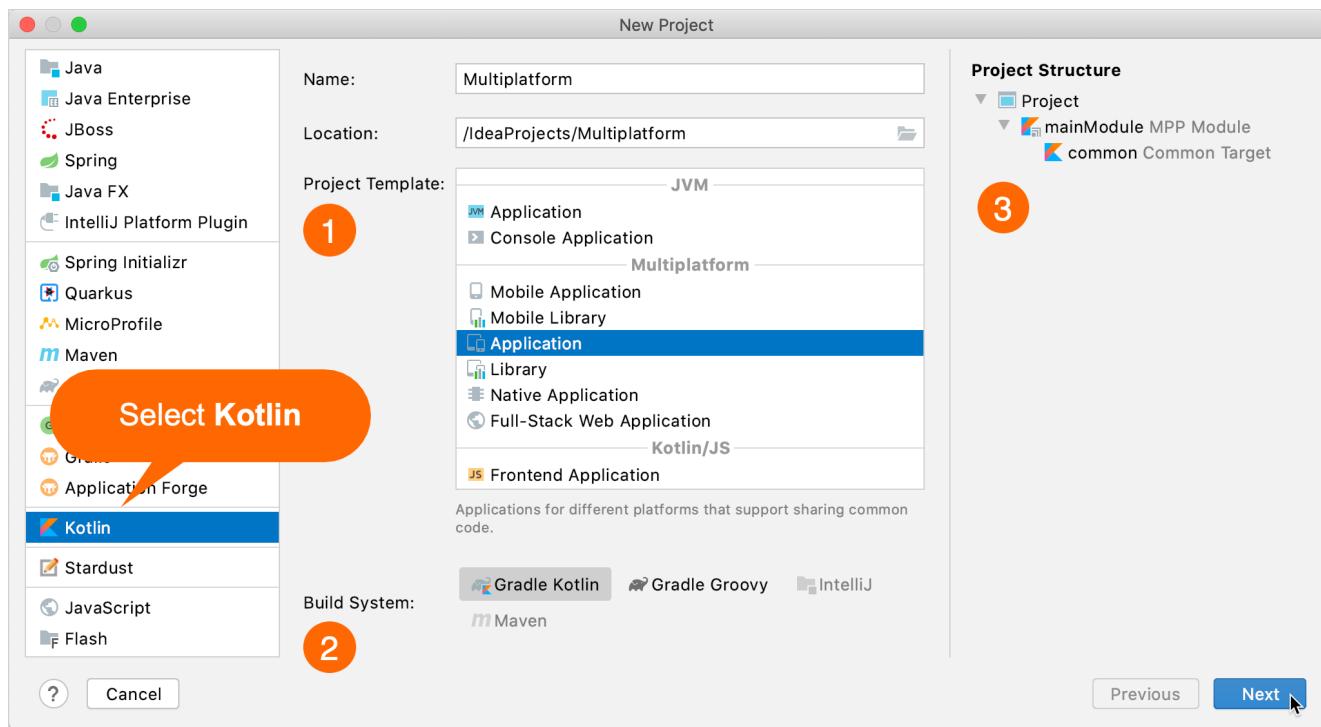
New tools in the IDE

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.

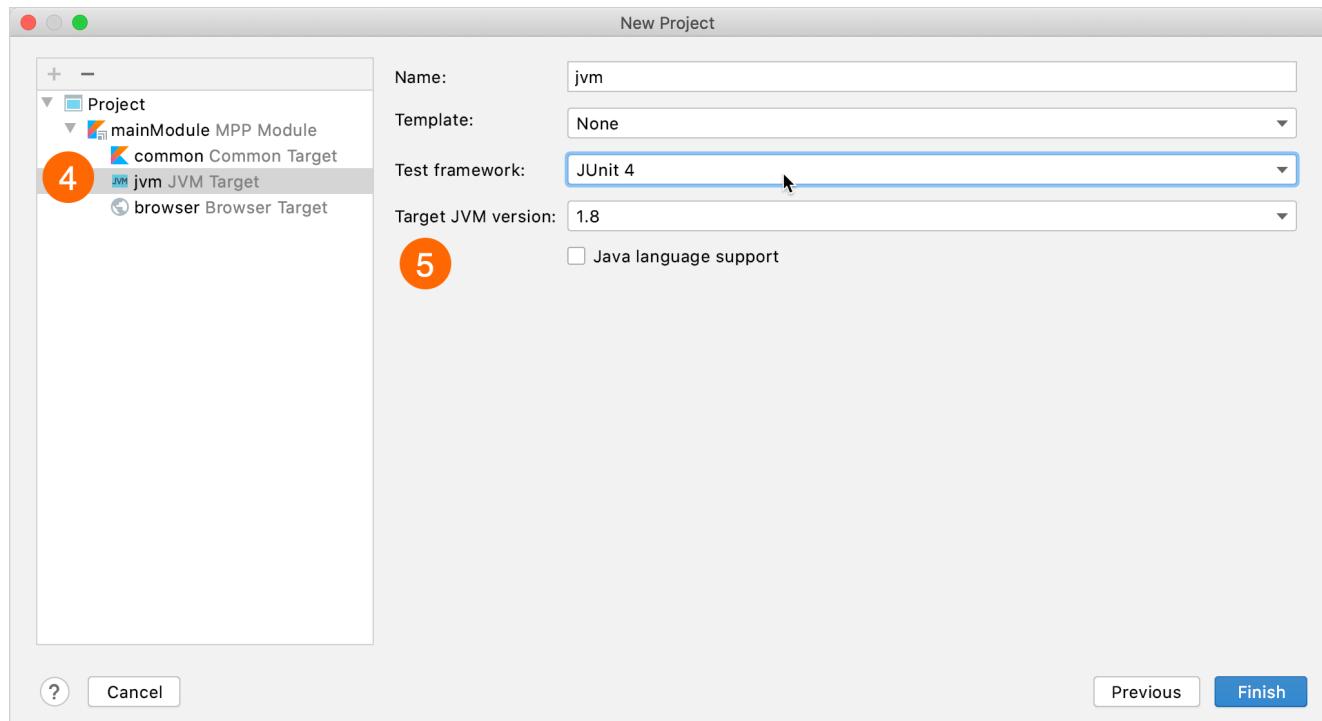
The Kotlin Project Wizard will only show the build systems supported on the selected project template.

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

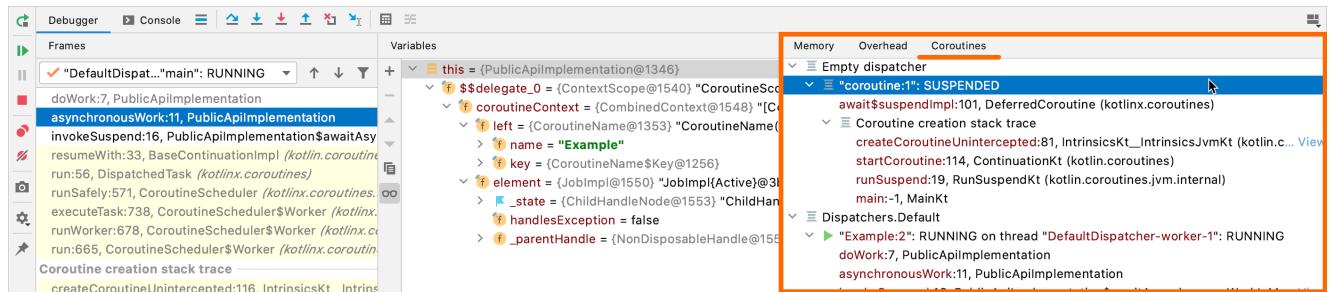
Many people already use [coroutines](#) for asynchronous programming. But when it came to debugging, working with coroutines before Kotlin 1.4, could be a real pain. Since coroutines jumped between threads, it was difficult to understand what a specific coroutine was doing and check its context. In some 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.

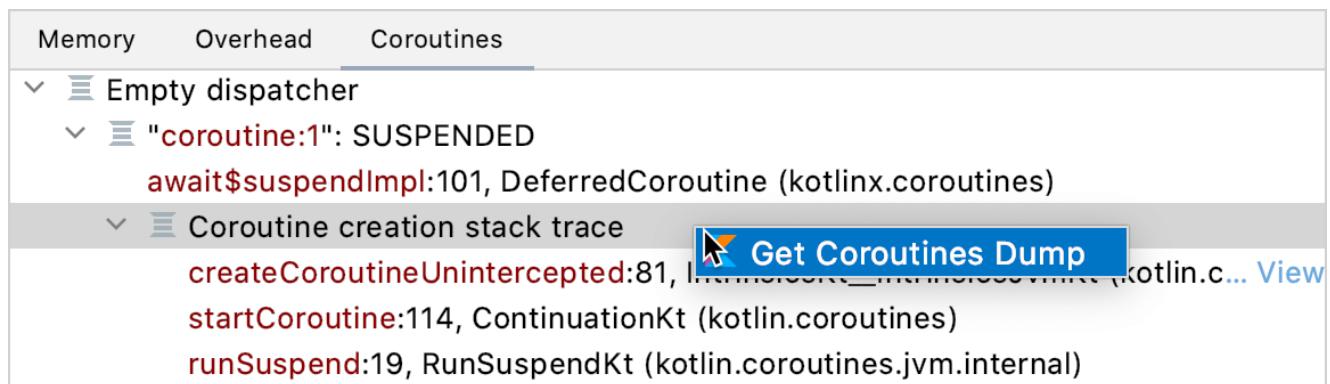


Debugging coroutines

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.



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`, `IllegalStateException`, `IllegalArgumentException`, and `TypeCastException`. This applies to: the `!!` operator, parameter null checks in the method preamble, platform-typed expression null checks, and the `as` operator with a non-null 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. [Read more about how to use the integration here](#).

New JS IR backend

The [IR backend for Kotlin/JS](#), which currently has [Alpha](#) stability, provides some new functionality specific to the Kotlin/JS target which is focused around the generated code size through dead code elimination, and improved interoperation with JavaScript and TypeScript, among others.

To enable the Kotlin/JS IR backend, set the key `kotlin.js.compiler=ir` in your `gradle.properties`, or pass the `IR` compiler type to the `js` function of your Gradle build script:

```
kotlin {
    js(IR) { // or: LEGACY, BOTH
        // ...
    }
    binaries.executable()
}
```

For more detailed information about how to configure the new backend, check out the [Kotlin/JS IR compiler documentation](#).

With the new `@JsExport` annotation and the ability to [generate TypeScript definitions](#) from Kotlin code, the Kotlin/JS IR compiler backend improves JavaScript & TypeScript interoperability. This also makes it easier to integrate Kotlin/JS code with existing tooling, to create hybrid applications and leverage code-sharing functionality in multiplatform projects.

[Learn more about the available features in the Kotlin/JS IR compiler backend](#).

Kotlin/Native

In 1.4.0, Kotlin/Native got a significant number of new features and improvements, including:

- [Support for suspending functions in Swift and Objective-C](#)
- [Objective-C generics support by default](#)
- [Exception handling in Objective-C/Swift interop](#)

- [Generate release .dSYMs on Apple targets by default](#)
- [Performance improvements](#)
- [Simplified management of CocoaPods dependencies](#)

Support for Kotlin's suspending functions in Swift and Objective-C

In 1.4.0, we add the basic support for suspending functions in Swift and Objective-C. Now, when you compile a Kotlin module into an Apple framework, suspending functions are available in it as functions with callbacks (completionHandler in the Swift/Objective-C terminology). When you have such functions in the generated framework's header, you can call them from your Swift or Objective-C code and even override them.

For example, if you write this Kotlin function:

```
suspend fun queryData(id: Int): String = ...
```

...then you can call it from Swift like so:

```
queryData(id: 17) { result, error in
    if let e = error {
        print("ERROR: \(e)")
    } else {
        print(result!)
    }
}
```

[Learn more about using suspending functions in Swift and Objective-C.](#)

Objective-C generics support by default

Previous versions of Kotlin provided experimental support for generics in Objective-C interop. Since 1.4.0, Kotlin/Native generates Apple frameworks with generics from Kotlin code by default. In some cases, this may break existing Objective-C or Swift code calling Kotlin frameworks. To have the framework header written without generics, add the `-Xno-objc-generics` compiler option.

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.all {
            freeCompilerArgs += "-Xno-objc-generics"
        }
    }
}
```

Please note that all specifics and limitations listed in the [documentation on interoperability with Objective-C](#) are still valid.

Exception handling in Objective-C/Swift interop

In 1.4.0, we slightly change the Swift API generated from Kotlin with respect to the way exceptions are translated.

There is a fundamental difference in error handling between Kotlin and Swift. All Kotlin exceptions are unchecked, while Swift has only checked errors. Thus, to make Swift code aware of expected exceptions, Kotlin functions should be marked with a `@Throws` annotation specifying a list of potential exception classes.

When compiling to Swift or the Objective-C framework, functions that have or are inheriting `@Throws` annotation are represented as `NSError*` -producing methods in Objective-C and as `throws` methods in Swift.

Previously, any exceptions other than `RuntimeException` and `Error` were propagated as `NSError`. Now this behavior changes: now `NSError` is thrown only for exceptions that are instances of classes specified as parameters of `@Throws` annotation (or their subclasses). Other Kotlin exceptions that reach Swift/Objective-C are considered unhandled and cause program termination.

Generate release .dSYMs on Apple targets by default

Starting with 1.4.0, the Kotlin/Native compiler produces debug symbol files (.dSYM s) for release binaries on Darwin platforms by default. This can be disabled with the `-Xadd-light-debug=disable` compiler option. On other platforms, this option is disabled by default. To toggle this option in Gradle, use:

```
kotlin {  
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {  
        binaries.all {  
            freeCompilerArgs += "-Xadd-light-debug={enable|disable}"  
        }  
    }  
}
```

[Learn more about crash report symbolication.](#)

Performance improvements

Kotlin/Native has received a number of performance improvements that speed up both the development process and execution. Here are some examples:

- To improve the speed of object allocation, we now offer the mimalloc memory allocator as an alternative to the system allocator. mimalloc works up to two times faster on some benchmarks. Currently, the usage of mimalloc in Kotlin/Native is experimental; you can switch to it using the `-Xallocator=mimalloc` compiler option.
- We've reworked how C interop libraries are built. With the new tooling, Kotlin/Native produces interop libraries up to 4 times as fast as before, and artifacts are 25% to 30% the size they used to be.
- Overall runtime performance has improved because of optimizations in GC. This improvement will be especially apparent in projects with a large number of long-lived objects. `HashMap` and `HashSet` collections now work faster by escaping redundant boxing.
- In 1.3.70 we introduced two new features for improving the performance of Kotlin/Native compilation: caching project dependencies and running the compiler from the Gradle daemon. Since that time, we've managed to fix

numerous issues and improve the overall stability of these features.

Simplified management of CocoaPods dependencies

Previously, once you integrated your project with the dependency manager CocoaPods, you could build an iOS, macOS, watchOS, or tvOS part of your project only in Xcode, separate from other parts of your multiplatform project. These other parts could be built in IntelliJ IDEA.

Moreover, every time you added a dependency on an Objective-C library stored in CocoaPods (Pod library), you had to switch from IntelliJ IDEA to Xcode, call pod install, and run the Xcode build there.

Now you can manage Pod dependencies right in IntelliJ IDEA while enjoying the benefits it provides for working with code, such as code highlighting and completion. You can also build the whole Kotlin project with Gradle, without having to switch to Xcode. This means you only have to go to Xcode when you need to write Swift/Objective-C code or run your application on a simulator or device.

Now you can also work with Pod libraries stored locally.

Depending on your needs, you can add dependencies between:

- A Kotlin project and Pod libraries stored remotely in the CocoaPods repository or stored locally on your machine.
- A Kotlin Pod (Kotlin project used as a CocoaPods dependency) and an Xcode project with one or more targets.

Complete the initial configuration, and when you add a new dependency to cocoapods, just re-import the project in IntelliJ IDEA. The new dependency will be added automatically. No additional steps are required.

[Learn how to add dependencies.](#)

Kotlin Multiplatform

Support for multiplatform projects is in [Alpha](#). It may change incompatibly and require manual migration in the future. We appreciate your feedback on it in [YouTrack](#).

[Kotlin Multiplatform](#) reduces time spent writing and maintaining the same code for [different platforms](#) while retaining the flexibility and benefits of native programming. We continue investing our effort in multiplatform features and improvements:

- [Sharing code in several targets with the hierarchical project structure](#)
- [Leveraging native libs in the hierarchical structure](#)
- [Specifying kotlinox dependencies only once](#)

Multiplatform projects require Gradle 6.0 or later.

Sharing code in several targets with the hierarchical project structure

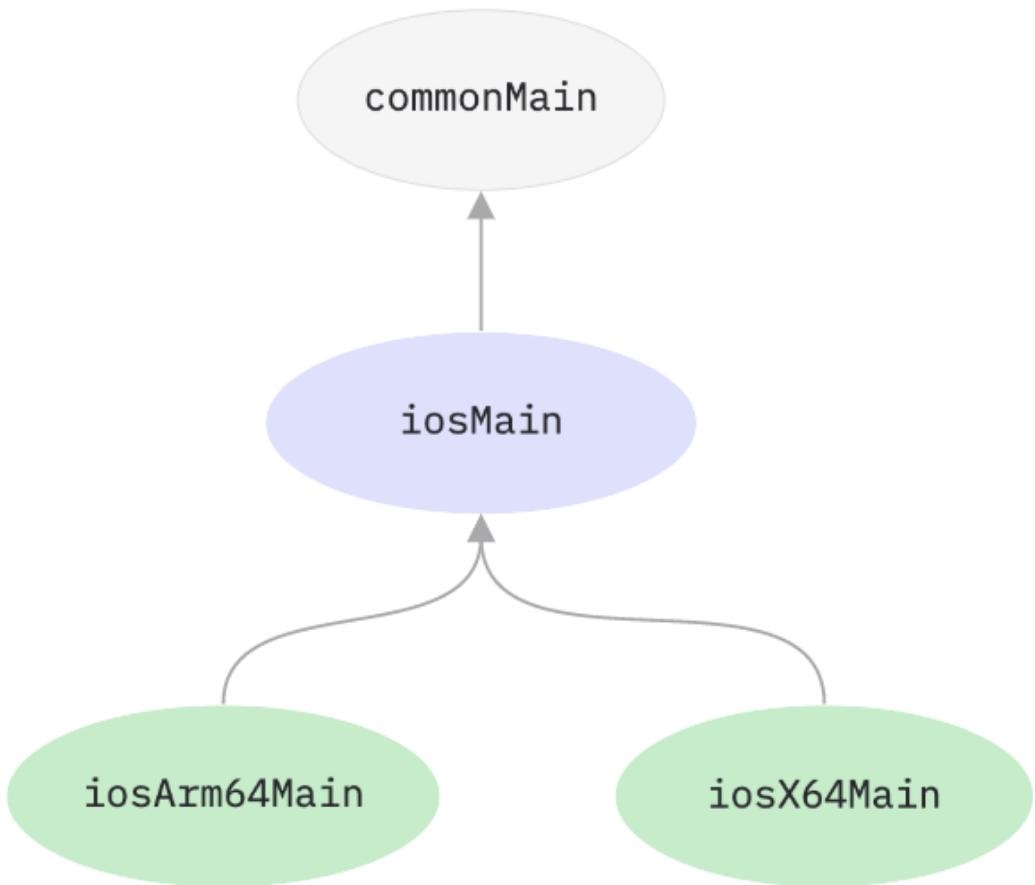
With the new hierarchical project structure support, you can share code among several platforms in a multiplatform project.

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](#).

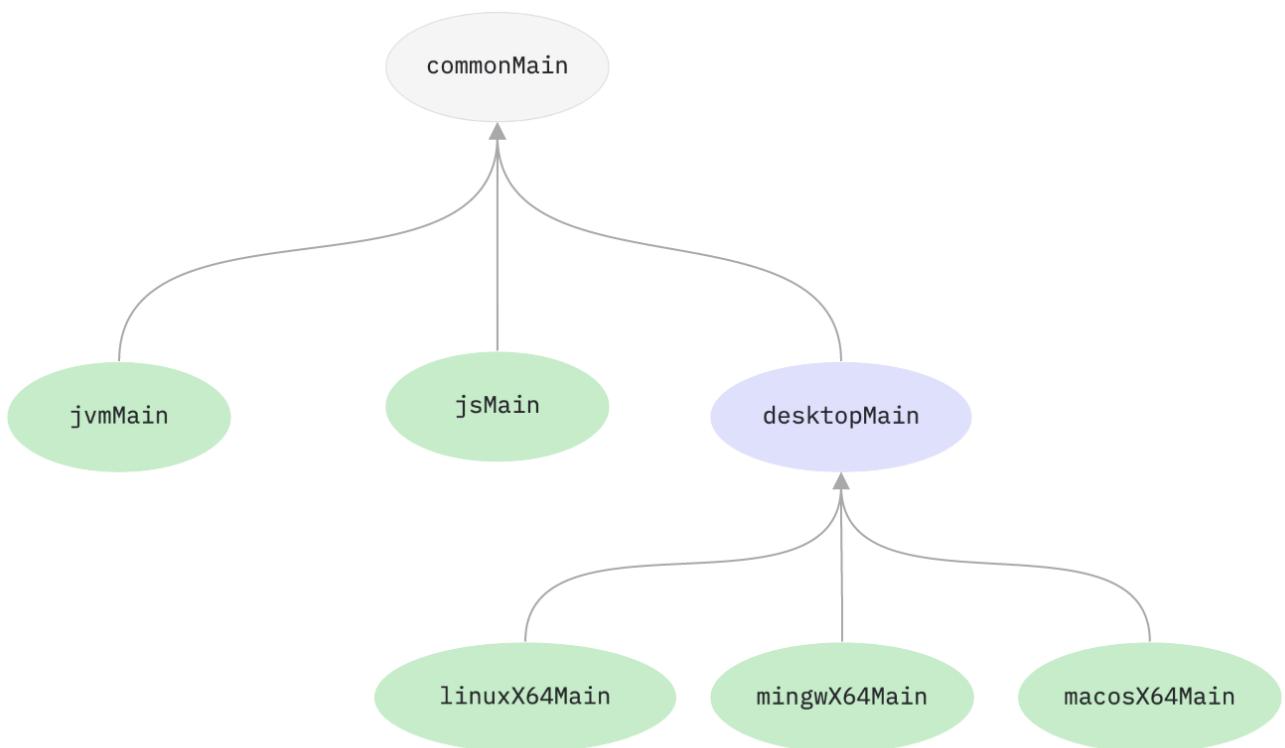
For example, create two iOS targets and the shared source set shown above with the `ios()` shortcut:

```

kotlin {
    ios() // iOS device and simulator targets; iosMain and iosTest source sets
}

```

For other combinations of targets, by connecting the source sets with the `dependsOn` relation.



Hierarchical structure

Groovy

```

kotlin {
    sourceSets {
        desktopMain {
            dependsOn(commonMain)
        }
        linuxX64Main {
            dependsOn(desktopMain)
        }
        mingwX64Main {
            dependsOn(desktopMain)
        }
        macosX64Main {
            dependsOn(desktopMain)
        }
    }
}

```

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)
        }
    }
}

```

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.

Groovy

```

kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.0'
            }
        }
    }
}

```

Kotlin

```

kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {

```

```
        implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.0")
    }
}
}
```

Don't use kotlinx library artifact names with suffixes specifying the platform, such as -common, -native, or similar, as they are NOT supported anymore. Instead, use the library base artifact name, which in the example above is kotlinx-coroutines-core.

However, the change doesn't currently affect:

- The stdlib library – starting from Kotlin 1.4.0, [the stdlib dependency is added automatically](#).
- The kotlin.test library – you should still use test-common and test-annotations-common. These dependencies will be addressed later.

If you need a dependency only for a specific platform, you can still use platform-specific variants of standard and kotlinx libraries with such suffixes as -jvm or -js, for example kotlinx-coroutines-core-jvm.

[Learn more about configuring dependencies](#).

Gradle project improvements

Besides Gradle project features and improvements that are specific to [Kotlin Multiplatform](#), [Kotlin/JVM](#), [Kotlin/Native](#), and [Kotlin/JS](#), there are several changes applicable to all Kotlin Gradle projects:

- [Dependency on the standard library is now added by default](#)
- [Kotlin projects require a recent version of Gradle](#)
- [Improved support for Kotlin Gradle DSL in the IDE](#)

Dependency on the standard library added by default

You no longer need to declare a dependency on the stdlib library in any Kotlin Gradle project, including a multiplatform one. The dependency is added by default.

The automatically added standard library will be the same version of the Kotlin Gradle plugin, since they have the same versioning.

For platform-specific source sets, the corresponding platform-specific variant of the library is used, while a common standard library is added to the rest. The Kotlin Gradle plugin will select the appropriate JVM standard library depending on the kotlinOptions.jvmTarget [compiler option](#) of your Gradle build script.

[Learn how to change the default behavior](#).

Minimum Gradle version for Kotlin projects

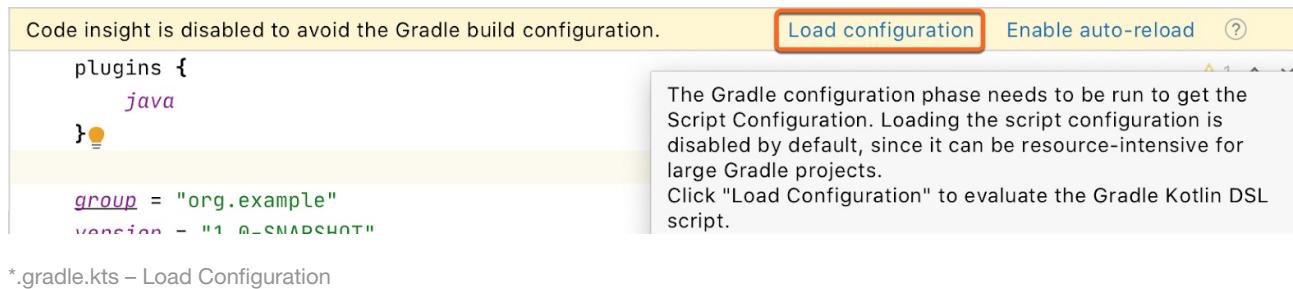
To enjoy the new features in your Kotlin projects, update Gradle to the [latest version](#). Multiplatform projects require Gradle 6.0 or later, while other Kotlin projects work with Gradle 5.4 or later.

Improved *.gradle.kts support in the IDE

In 1.4.0, we continued improving the IDE support for Gradle Kotlin DSL scripts (*.gradle.kts files). Here is what the new version brings:

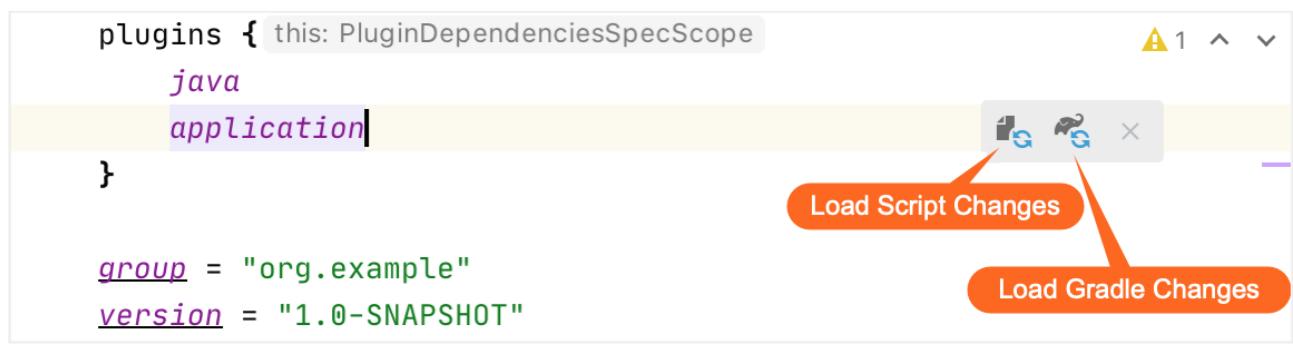
- Explicit loading of script configurations for better performance. Previously, the changes you make to the build script were loaded automatically in the background. To improve the performance, we've disabled the automatic loading of build script configuration in 1.4.0. Now the IDE loads the changes only when you explicitly apply them.

In Gradle versions earlier than 6.0, you need to manually load the script configuration by clicking Load Configuration in the editor.



In Gradle 6.0 and above, you can explicitly apply changes by clicking Load Gradle Changes or by reimporting the Gradle project.

We've added one more action in IntelliJ IDEA 2020.1 with Gradle 6.0 and above – Load Script Configurations, which loads changes to the script configurations without updating the whole project. This takes much less time than reimporting the whole project.

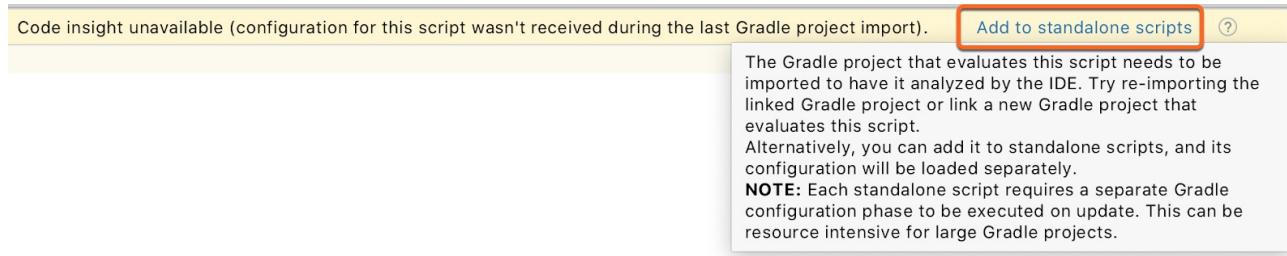


*.gradle.kts – Load Script Changes and Load Gradle Changes

You should also Load Script Configurations for newly created scripts or when you open a project with new Kotlin plugin for the first time.

With Gradle 6.0 and above, you are now able to load all scripts at once as opposed to the previous implementation where they were loaded individually. Since each request requires the Gradle configuration phase to be executed, this could be resource-intensive for large Gradle projects.

Currently, such loading is limited to build.gradle.kts and settings.gradle.kts files (please vote for the related [issue](#)). To enable highlighting for init.gradle.kts or applied [script plugins](#), use the old mechanism – adding them to standalone scripts. Configuration for that scripts will be loaded separately when you need it. You can also enable auto-reload for such scripts.



*.gradle.kts – Add to standalone scripts

- Better error reporting. Previously you could only see errors from the Gradle Daemon in separate log files. Now the Gradle Daemon returns all the information about errors directly and shows it in the Build tool window. This saves you both time and effort.

Standard library

Here is the list of the most significant changes to the Kotlin standard library in 1.4.0:

- [Common exception processing API](#)
- [New functions for arrays and collections](#)
- [Functions for string manipulations](#)
- [Bit operations](#)
- [Delegated properties improvements](#)
- [Converting from KType to Java Type](#)
- [Proguard configurations for Kotlin reflection](#)
- [Improving the existing API](#)
- [module-info descriptors for stdlib artifacts](#)
- [Deprecations](#)
- [Exclusion of the deprecated experimental coroutines](#)

Common exception processing API

The following API elements have been moved to the common library:

- `Throwable.stackTraceToString()` extension function, which returns the detailed description of this throwable with its stack trace, and `Throwable.printStackTrace()`, which prints this description to the standard error output.
- `Throwable.addSuppressed()` function, which lets you specify the exceptions that were suppressed in order to deliver the exception, and the `Throwable.suppressedExceptions` property, which returns a list of all the suppressed exceptions.
- `@Throws` annotation, which lists exception types that will be checked when the function is compiled to a platform method (on JVM or native platforms).

New functions for arrays and collections

Collections

In 1.4.0, the standard library includes a number of useful functions for working with collections:

- `setOfNotNull()`, which makes a set consisting of all the non-null items among the provided arguments.

```
fun main() {
    //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 "aeiou" }.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. Double-ended queue lets you can add or remove elements both at the beginning and the 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 `appendIn()` 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.



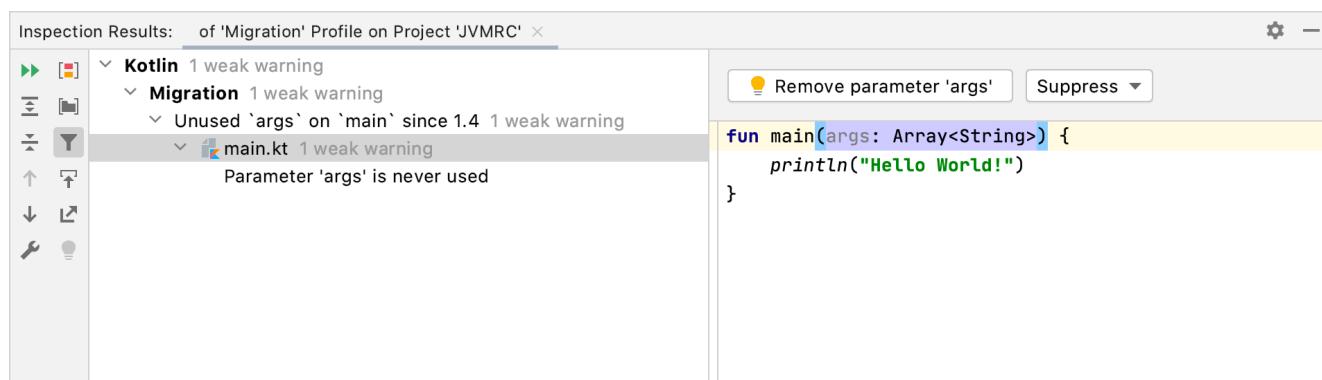
Kotlin Migration

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

Release date: 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 high-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
}
//sampleEnd
```

```
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() } }
        .isEmpty { 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 isEmpty 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

Release date: 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 typealias 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)
    }
}

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 = *intArrayOf(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
- Exponentation: pow (an extension function), sqrt, hypot, exp, expm1
- Logarithms: log, log2, log10, ln, ln1p
- Rounding:
 - ceil, floor, truncate, round (half to even) functions
 - roundToInt, roundToLong (half to integer) extension functions
- Sign and absolute value:
 - abs and sign functions
 - absoluteValue and sign extension properties
 - withSign extension function
- max and min of two values
- Binary representation:
 - ulp extension property
 - nextUp, nextDown, nextTowards extension functions
 - toBits, toRawBits, Double.fromBits (these are in the `kotlin` package)

The same set of functions (but without constants) is also available for `Float` arguments.

Operators and conversions for `BigInteger` and `BigDecimal`

Kotlin 1.2 introduces a set of functions for operating with `BigInteger` and `BigDecimal` and creating them from other numeric types. These are:

- `toBigInteger` for `Int` and `Long`
- `toBigDecimal` for `Int`, `Long`, `Float`, `Double`, and `BigInteger`
- Arithmetic and bitwise operator functions:
 - Binary operators `+`, `-`, `*`, `/`, `%` and infix functions `and`, `or`, `xor`, `shl`, `shr`
 - Unary operators `-`, `++`, `--`, and a function `inv`

Floating point to bits conversions

New functions were added for converting Double and Float to and from their bit representations:

- toBits and toRawBits returning Long for Double and Int for Float
- Double.fromBits and Float.fromBits for creating floating point numbers from the bit representation

Regex is now serializable

The kotlin.text.Regex class has become Serializable and can now be used in serializable hierarchies.

Closeable.use calls Throwable.addSuppressed if available

The Closeable.use function calls Throwable.addSuppressed when an exception is thrown during closing the resource after some other exception.

To enable this behavior you need to have kotlin-stdlib-jdk7 in your dependencies.

JVM backend

Constructor calls normalization

Ever since version 1.0, Kotlin supported expressions with complex control flow, such as try-catch expressions and inline function calls. Such code is valid according to the Java Virtual Machine specification. Unfortunately, some bytecode processing tools do not handle such code quite well when such expressions are present in the arguments of constructor calls.

To mitigate this problem for the users of such bytecode processing tools, we've added a command-line compiler option (-Xnormalize-constructor-calls=MODE) that tells the compiler to generate more Java-like bytecode for such constructs. Here MODE is one of:

- disable (default) – generate bytecode in the same way as in Kotlin 1.0 and 1.1.
- enable – generate Java-like bytecode for constructor calls. This can change the order in which the classes are loaded and initialized.
- preserve-class-initialization – generate Java-like bytecode for constructor calls, ensuring that the class initialization order is preserved. This can affect overall performance of your application; use it only if you have some complex state shared between multiple classes and updated on class initialization.

The “manual” workaround is to store the values of sub-expressions with control flow in variables, instead of evaluating them directly inside the call arguments. It's similar to -Xnormalize-constructor-calls=enable.

Java-default method calls

Before Kotlin 1.2, interface members overriding Java-default methods while targeting JVM 1.6 produced a warning on

super calls: Super calls to Java default methods are deprecated in JVM target 1.6. Recompile with '-jvm-target 1.8'. In Kotlin 1.2, there's an error instead, thus requiring any such code to be compiled with JVM target 1.8.

Breaking change: consistent behavior of x.equals(null) for platform types

Calling x.equals(null) on a platform type that is mapped to a Java primitive (Int!, Boolean!, Short !, Long!, Float!, Double!, Char!) incorrectly returned true when x was null. Starting with Kotlin 1.2, calling x.equals(...) on a null value of a platform type throws an NPE (but x == ... does not).

To return to the pre-1.2 behavior, pass the flag -Xno-exception-on-explicit-equals-for-boxed-null to the compiler.

Breaking change: fix for platform null escaping through an inlined extension receiver

Inline extension functions that were called on a null value of a platform type did not check the receiver for null and would thus allow null to escape into the other code. Kotlin 1.2 forces this check at the call sites, throwing an exception if the receiver is null.

To switch to the old behavior, pass the fallback flag -Xno-receiver-assertions to the compiler.

JavaScript backend

TypedArrays support enabled by default

The JS typed arrays support that translates Kotlin primitive arrays, such as IntArray, DoubleArray, into [JavaScript typed arrays](#), that was previously an opt-in feature, has been enabled by default.

Tools

Warnings as errors

The compiler now provides an option to treat all warnings as errors. Use -Werror on the command line, or the following Gradle snippet:

```
compileKotlin {  
    kotlinOptions.allWarningsAsErrors = true  
}
```

What's new in Kotlin 1.1

Release date: 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 KEPs 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)
    }
}

```

What's new in Kotlin plugin 2021.1

This release aims to increase productivity and improve the development experience. Version 2021.1 of the Kotlin plugin introduces the following major updates:

- [Performance improvements](#)

- [Evaluation of custom getters during debugging](#)
- [Improved Change Signature refactoring](#)
- [Code completion for type arguments](#)
- [UML diagrams for Kotlin classes](#)
- [Other platform enhancements](#)

You can also learn about new features in [this blog post](#).

Performance improvements

With 2021.1, the Kotlin plugin has received a number of performance improvements that speed up the development process.

Here are some major refinements:

- Faster syntax and error highlighting. The code highlighting API has been reworked, so you now get all the necessary diagnostic information faster. See the [benchmark results](#) and share your feedback in this [YouTrack issue](#).
- Code completion speed improvements. The Kotlin plugin provides faster code completion. Check out this [YouTrack issue](#) to see the benchmark results.

This version also introduces other improvements for code completion, particularly [code completion for type arguments](#).

- Improved IDE responsiveness. We've fixed numerous issues based on your feedback, and we've improved the overall stability of the plugin, as well as IDE responsiveness.

Evaluation of custom getters during debugging

This release expands the capabilities of the debugger. One of the most notable features is the ability to display Kotlin properties without a [backing field](#) in the Variables view.

Previously, during a debug session, you could only see the properties without a custom getter and those with a backing field. Properties with a custom getter didn't appear because they are represented as regular methods on the JVM. Starting with version 2021.1 of the Kotlin plugin, you can see all such properties and evaluate them on demand by clicking on `get()` near the property name.

For example, when debugging of the following code, you can execute the get() method to see the value:

```
class LanguageVersion(val major: Int, val minor: Int) {
    val isStable: Boolean
        get() = major <= 1 && minor <= 4
    val isExperimental: Boolean
```

```

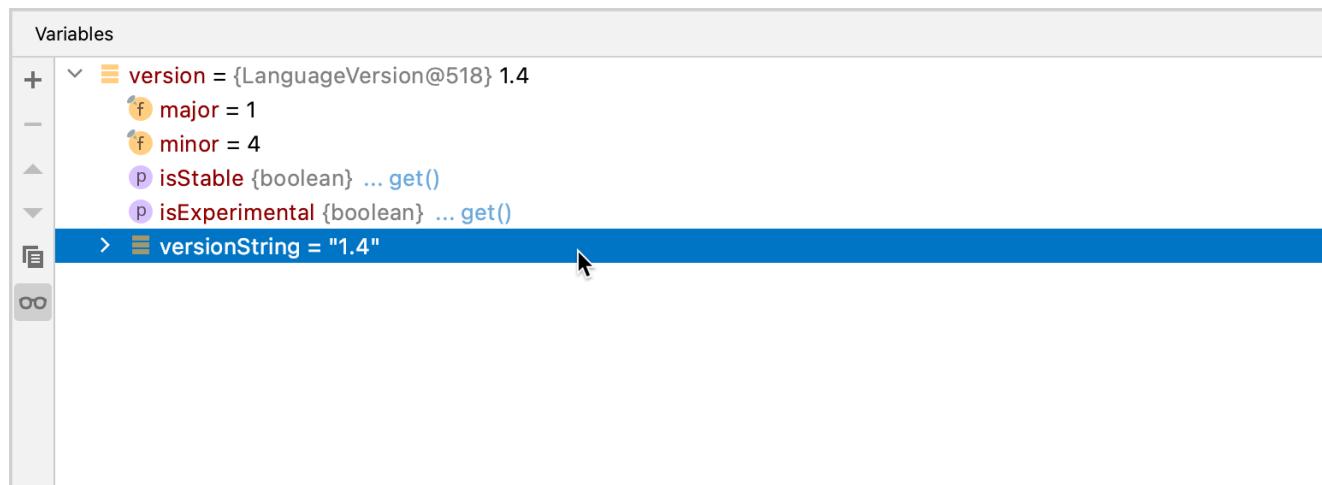
        get() = !isStable
    val versionString: String
        get() = "$major.$minor"

    override fun toString() = versionString
}

fun main() {
    val version = LanguageVersion(1, 4)
}

```

In the Debug window, you can see the values of the properties:



Variables view

Improved Change Signature refactoring

This release contains changes to the [Change Signature](#) refactoring. We've fixed more than [40 issues](#) to make the refactoring process better.

Here are some of the important improvements:

- Support for warning messages for more refactoring cases.
- Fixes to inheritance refactoring, including cross-language refactoring. See the [YouTrack issue](#) for details.* Improved UX of properties processing.
- Improved UX for processing properties.
- Addition of a way to declare the default parameter value. See the [YouTrack issue](#) for details.
- Correct backticks processing.

Code completion for type arguments

From now on, code completion will offer functions and properties after generic functions and will restore type arguments where needed. When you select such a function from the list, the IDE adds the correct type argument to the preceding code.

In the following example, there is a list of strings to be reversed:

```
fun typeArgumentsAtCodeCompletion() {  
    // Function definition from stdlib:  
    // public fun <T> emptyList(): List<T>  
  
    val list: List<String> = emptyList() // T is inferred from the context (explicit variable  
    type)  
}
```

To reverse this list, start typing the name of the reversed() function:



Now code completion suggests functions

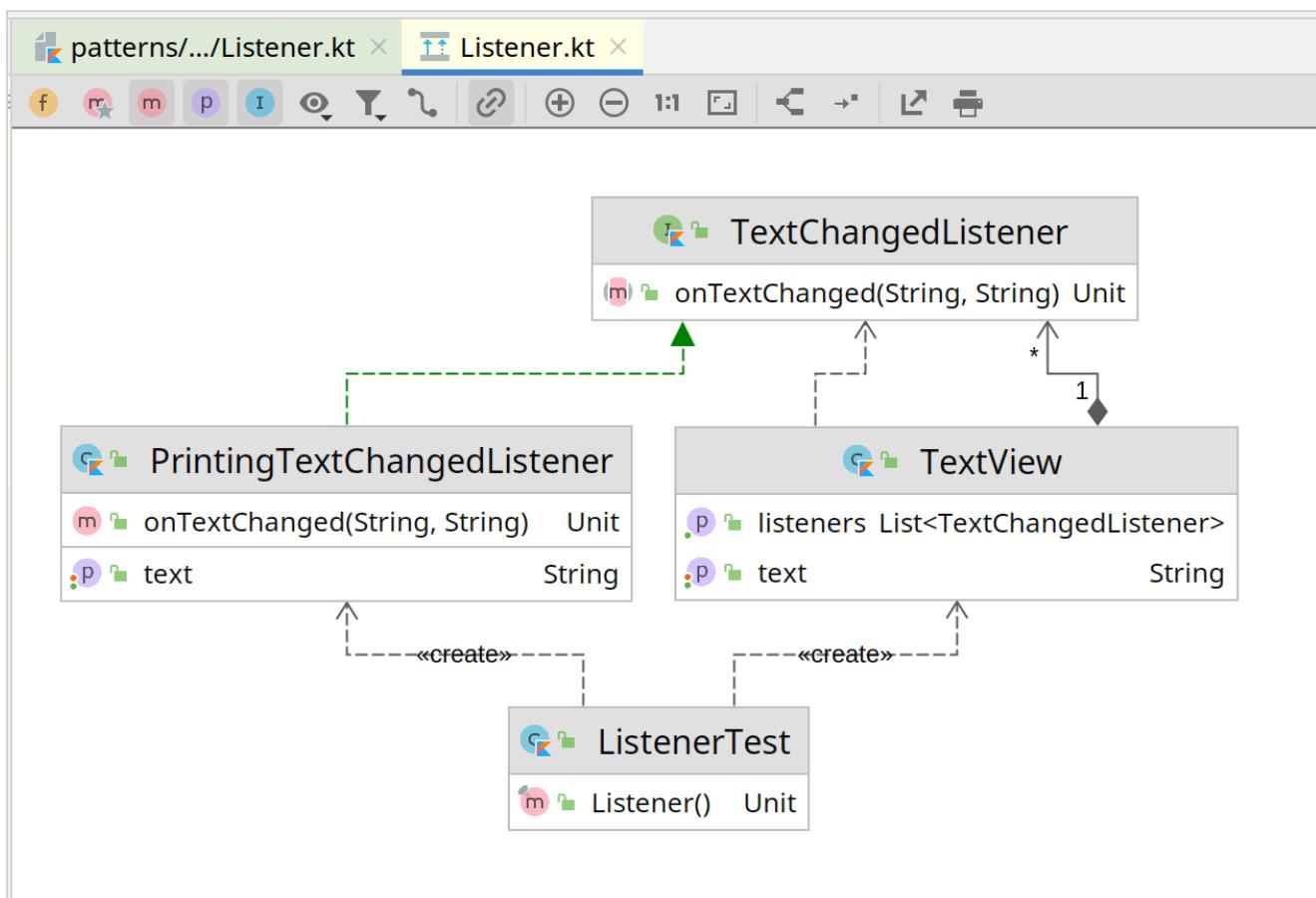
After you apply the code completion suggestion, the IDE automatically adds the <String> type:

```
fun typeArgumentsAtCodeCompletion() {  
    // Function definition from stdlib:  
    // public fun <T> emptyList(): List<T>  
  
    val list: List<String> = emptyList<String>().reversed()  
}
```

The type argument for emptyList() is required to evaluate the expression type. Otherwise, the code will not compile.

UML diagrams for Kotlin classes

With this release, you can test Kotlin code visualization via UML Class diagrams. To build a diagram, select Diagrams | Show Diagram... | Kotlin Classes in the Project View.



UML diagram for Kotlin classes

Currently, the diagrams only show inheritance and nesting relationships. All other more detailed association connections, like aggregation, composition, dependency, and others will be available in the future releases.

Other platform enhancements

Since the plugin and the platform have been moved to the same codebase and now ship simultaneously, this release also brings the following features that improve the Kotlin experience:

- You can now launch the basic memory profiler that was announced in the [IntelliJ IDEA 2020.3 release](#) via Gradle run.
- To improve the experience of working with coroutines, this release provides [better thread-blocking call detection](#). The inspection now correctly warns you about inappropriate blocking method calls.
- We've fixed some of the language injection issues that received the most votes. Language injection now works correctly for the following cases:
 - When you use triple quotes to inject multi-line strings and add the `.trimIndent()`, or `.trimMargin()` functions at the end. See the [YouTrack issue](#) for details.
 - When you concatenate strings. See the [YouTrack issue](#) for details.

See the [What's new in IntelliJ IDEA 2021.1 blog post](#) or watch [this video](#) to learn more about the platform enhancements.

What's new in Kotlin plugin 2020.3

This plugin release introduces the following features:

- [New types of inline refactorings](#)
- [Structural search and replace](#)
- [EditorConfig support](#)
- [Project templates for Jetpack Compose for Desktop](#)

New types of inline refactorings

Cross-language conversion is possible starting with version 2020.3 of the Kotlin plugin. You can now use the inline refactoring actions for Kotlin elements defined in Java.

The Kotlin plugin can inline code from libraries with attached sources. That means you can [inline refactor](#) all the [Kotlin scope functions](#): also, let, run, apply, and with.

Also, this release brings refactoring improvements for lambda expressions. Now the IDE analyzes their syntax more thoroughly and formats them correctly.

Structural search and replace

[Structural search and replace \(SSR\)](#) actions are now available for Kotlin. The SSR feature allows you to find and replace code patterns, taking the syntax and semantics of the source code into account.

To start using the feature, open the Structural Search dialog for your .kt file and select one of the [templates](#) or create your own.

EditorConfig support

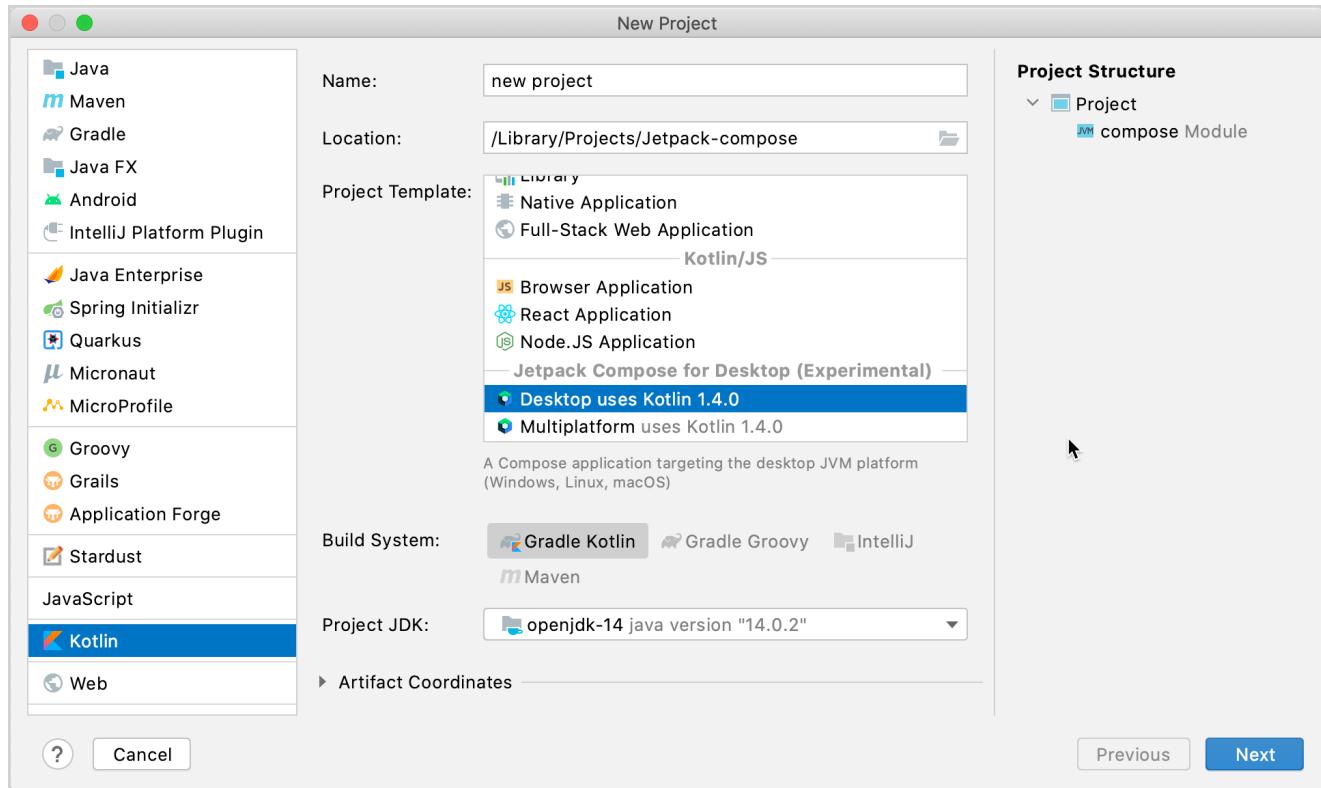
From 2020.3, the Kotlin API has the full support of [.editorconfig](#) files for [managing code style on a directory level](#).

Project templates for Jetpack Compose for Desktop

The new experimental Jetpack Compose for Desktop templates are now available in the Kotlin Project Wizard. You can

create a project using the following templates:

- Desktop – a Compose project targeting the desktop JVM platform: Windows, Linux, or macOS.
- Multiplatform – a Multiplatform Compose project targeting the desktop JVM platform (Windows, Linux, macOS) and Android with shared code in common modules.



Kotlin Project Wizard – Jetpack Compose project

To create a project, select one of the templates while creating a new project and specify the Gradle build system. The Kotlin plugin creates all the configuration files automatically. You can try out this experimental feature by working through [Getting Started with Compose for Desktop tutorial](#).

Read more about Jetpack Compose features in [this blog post](#) and look through the [examples of Compose applications](#).

Basic syntax

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

Read-only local variables are defined using the keyword `val`. They can be assigned a value only once.

```
fun main() {
//sampleStart
    val a: Int = 1 // immediate assignment
    val b = 2 // `Int` type is inferred
    val c: Int // Type required when no initializer is provided
    c = 3 // deferred assignment
//sampleEnd
    println("a = $a, b = $b, c = $c")
}
```

Variables that can be reassigned use the `var` keyword.

```
fun main() {
//sampleStart
    var x = 5 // `Int` type is inferred
    x += 1
//sampleEnd
    println("x = $x")
}
```

You can declare variables at the top level.

```
//sampleStart
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
//sampleEnd

fun main() {
    println("x = $x; PI = $PI")
    incrementX()
    println("incrementX()")
    println("x = $x; PI = $PI")
}
```

See also [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(var height: Double, var length: Double) {  
    var perimeter = (height + length) * 2  
}
```

The default constructor with parameters listed in the class declaration is available automatically.

```
class Rectangle(var height: Double, var length: Double) {  
    var perimeter = (height + length) * 2  
}  
fun main() {  
//sampleStart  
    val rectangle = Rectangle(5.0, 2.0)  
    println("The perimeter is ${rectangle.perimeter}")  
//sampleEnd  
}
```

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(var height: Double, var length: Double): Shape() {  
    var perimeter = (height + length) * 2  
}
```

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
}

```

Using 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 var s) for all properties
- equals()
- hashCode()
- toString()
- copy()
- component1(), component2(), ..., for all properties (see [Data classes](#))

Default values for function parameters

```
fun foo(a: Int = 0, b: String = "") { ... }
```

Filter a list

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

Or alternatively, even shorter:

```
val positives = list.filter { it > 0 }
```

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

Check the presence of an element in a collection

```
if ("john@example.com" in emailsList) { ... }  
if ("jane@example.com" !in emailsList) { ... }
```

String interpolation

```
println("Name $name")
```

Learn the difference between [Java and Kotlin string concatenation](#).

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 range: includes 100
for (i in 1 until 100) { ... } // half-open range: does not include 100
for (x in 2..10 step 2) { ... }
for (x in 10 downTo 1) { ... }
if (x in 1..10) { ... }
```

Lazy property

```
val p: String by lazy {
    // compute the string
}
```

Extension functions

```
fun String.spaceToCamelCase() { ... }

"Convert this to camelcase".spaceToCamelCase()
```

Create a singleton

```
object Resource {
    val name = "Name"
}
```

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()  
  
println(files?.size ?: "empty") // if files is null, this prints "empty"
```

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() ?: ""
```

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

```
fun foo(param: Int) {
    val result = if (param == 1) {
        "one"
    } else if (param == 2) {
        "two"
    } else {
        "three"
    }
}
```

Builder-style usage of methods that return Unit

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

Single-expression functions

```
fun theAnswer() = 42
```

This is equivalent to

```
fun theAnswer(): Int {
    return 42
}
```

This can be effectively combined with other idioms, leading to shorter code. For example, with the when expression:

```
fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param value")
}
```

Call multiple methods on an object instance (with)

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
    fun forward(pixels: Double)
}

val myTurtle = Turtle()
with(myTurtle) { //draw a 100 pix square
    penDown()
    for (i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

Configure properties of an object (apply)

```
val myRectangle = Rectangle().apply {
    length = 4
    breadth = 5
    color = 0xFAFAFA
}
```

This is useful for configuring properties that aren't present in the object constructor.

Java 7's try-with-resources

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader ->
    println(reader.readText())
}
```

Generic function that requires the generic type information

```
// public final class Gson {
//     ...
//     public <T> T fromJson(JsonElement json, Class<T> classOfT) throws JsonSyntaxException {
//         ...

inline fun <reified T: Any> Gson.fromJson(json: JsonElement): T = this.fromJson(json,
T::class.java)
```

Nullable Boolean

```
val b: Boolean? = ...
if (b == true) {
    ...
} else {
    // `b` is false or null
}
```

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?

Learn how to perform [typical tasks with strings 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 | 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 | Editor | Inspections | Kotlin.
2. Open Kotlin | Style issues.
3. Switch on File is not formatted according to project settings 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 (potentially with related top-level declarations), its name should be the same as the name of the class, with the .kt extension appended. If a file contains multiple classes, or only top-level declarations, choose a name describing what the file contains, and name the file accordingly. Use upper camel case 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.

Source file organization

Placing multiple declarations (classes, top-level functions or properties) in the same Kotlin source file is encouraged as long as these declarations are closely related to each other semantically, and the file size remains reasonable (not exceeding a few hundred lines).

In particular, when defining extension functions for a class which are relevant for all clients of this class, put them in the same file with the class itself. When defining extension functions that make sense only for a specific client, put them next to the code of that client. Avoid creating files just to hold all extensions of some class.

Class layout

The contents of a class should go in the following order:

1. Property declarations and initializer blocks
2. Secondary constructors
3. Method declarations
4. Companion object

Do not sort the method declarations alphabetically or by visibility, and do not separate regular methods from extension methods. Instead, put related stuff together, so that someone reading the class from top to bottom can follow the logic of what's happening. Choose an order (either higher-level stuff first, or vice versa) and stick to it.

Put nested classes next to the code that uses those classes. If the classes are intended to be used externally and aren't referenced inside the class, put them in the end, after the companion object.

Interface implementation layout

When implementing an interface, keep the implementing members in the same order as members of the interface (if

necessary, interspersed with additional private methods used for the implementation).

Overload layout

Always put overloads next to each other in a class.

Naming rules

Package and class naming rules in Kotlin are quite simple:

- Names of packages are always lowercase and do not use underscores (org.example.project). Using multi-word names is generally discouraged, but if you do need to use multiple words, you can either just concatenate them together or use camel case (org.example.myProject).
- Names of classes and objects start with an uppercase letter and use camel case:

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

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

Function names

Names of functions, properties and local variables start with a lowercase letter and use camel case 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 currently not supported by the Android runtime. 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 names ([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 in 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().getDefaultCharsetForPropertiesFiles(file)
```

Control flow statements

If the condition of an if or when statement is multiline, always use curly braces around the body of the statement.

Indent each subsequent line of the condition by four spaces relative to statement begin. 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 usually should 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 of 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 of 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 | 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, // trailing comma  
    ) in cars) {  
        meanValue += year  
    }  
    println(meanValue/cars.size)  
}  
printMeanValue()
```

Documentation comments

For longer documentation comments, place the opening `/**` on a separate line and begin each subsequent line with an asterisk:

```
/**  
 * This is a documentation comment  
 * on multiple lines.  
 */
```

Short comments can be placed on a single line:

```
/** This is a short documentation comment. */
```

Generally, avoid using `@param` and `@return` tags. Instead, incorporate the description of parameters and return values directly into the documentation comment, and add links to parameters wherever they are mentioned. Use `@param` and `@return` only when a lengthy description is required which doesn't fit into the flow of the main text.

```
// Avoid doing this:

/**
 * Returns the absolute value of the given number.
 * @param number The number to return the absolute value for.
 * @return The absolute value.
 */
fun abs(number: Int) { /*...*/ }

// Do this instead:

/**
 * Returns the absolute value of the given [number].
 */
fun abs(number: 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 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 until function to loop over an open range:

```
for (i in 0..n - 1) { /*...*/ } // bad  
for (i in 0 until n) { /*...*/ } // good
```

Strings

Prefer string templates to string concatenation.

Prefer multiline strings to embedding \n escape sequences into regular string literals.

To maintain indentation in multiline strings, use trimIndent when the resulting string does not require any internal indentation, or trimMargin when internal indentation is required:

```
fun main() {  
    //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, with the exception of overrides that do not require any new documentation (to support generating documentation for the library)

Basic types

In Kotlin, everything is an object in the sense that we can call member functions and properties on any variable. Some types can have a special internal representation - for example, numbers, characters and booleans can be represented as primitive values at runtime - but to the user they look like ordinary classes. In this section we describe the basic types used in Kotlin: [numbers](#), [booleans](#), [characters](#), [strings](#), and [arrays](#).

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)

All variables initialized with integer values not exceeding the maximum value of Int have the inferred type Int. If the initial value exceeds this value, then the type is Long. To specify the Long value explicitly, append the suffix L to the value.

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

Floating-point types

For real numbers, Kotlin provides floating-point types Float and Double. According to the [IEEE 754 standard](#), floating point types differ by their decimal place, that is, how many decimal digits they can store. Float reflects the IEEE 754 single precision, while Double provides double precision.

Type	Size (bits)	Significant bits	Exponent bits	Decimal digits
Float	32	24	8	6-7

Type	Size (bits)	Significant bits	Exponent bits	Decimal digits
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
```

Note that 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

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.

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
```

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.

Note that nullable references to the same number can be 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 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.

```
fun main() {  
    //sampleStart  
    val b: Byte = 1 // OK, literals are checked statically  
    // val i: Int = b // ERROR  
    val i1: Int = b.toInt()  
    //sampleEnd  
}
```

All number types support conversions to other types:

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

In many cases, there is no need in 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

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, when the operands are not statically typed as floating point numbers (e.g. `Any`, `Comparable<...>`, a type parameter), the operations use the `equals` and `compareTo` implementations for `Float` and `Double`, which disagree with the standard, so that:

- `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`

Unsigned integers

In addition to [integer types](#), Kotlin provides the following types for unsigned integer numbers:

- `UByte`: an unsigned 8-bit integer, ranges from 0 to 255
- `UShort`: an unsigned 16-bit integer, ranges from 0 to 65535
- `UInt`: an unsigned 32-bit integer, ranges from 0 to $2^{32} - 1$
- `ULong`: an unsigned 64-bit integer, ranges from 0 to $2^{64} - 1$

Unsigned types support most of the operations of their signed counterparts.

Changing type from unsigned type to signed counterpart (and vice versa) is a binary incompatible change.

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 of unsigned type has 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 similar API to `Array` class without boxing overhead.

When you use unsigned arrays, you'll get a warning that indicates that this feature is not stable yet. To remove the warning, opt in using 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 array are not a stable feature, so API which 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.

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 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 // ULong, even though no expected type provided and constant fits into UInt
```

Further discussion

See [language proposal for unsigned types](#) for technical details and further discussion.

Booleans

The type Boolean represents boolean objects that can have two values: true and false.

Boolean has a nullable counterpart Boolean? that also has the null value.

Built-in operations on booleans include:

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

|| and && work lazily.

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

    println(myTrue || myFalse)
    println(myTrue && myFalse)
    println(!myTrue)
//sampleEnd
}
```

On JVM: nullable references to boolean objects are boxed similarly to [numbers](#).

Characters

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

Special characters start from an escaping backslash \. The following escape sequences are supported: \t, \b, \n, \r, \', \" , \\ and \\$.

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 JVM: Like [numbers](#), characters are boxed when a nullable reference is needed. Identity is not preserved by the boxing operation.

Strings

Strings in Kotlin are represented by the type String. 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"
    println(str.uppercase()) // Create and print a new String object
    println(str) // the original string remains the same
//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")  
    //sampleEnd  
}
```

Note that in most cases using [string templates](#) or raw strings is preferable to string concatenation.

String literals

Kotlin has two types of string literals:

- escaped strings that may contain escaped characters
- raw strings that can contain newlines and arbitrary text

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](#) above for the list of supported escape sequences.

A raw string 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 raw 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, | 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 the string. A template expression starts with a dollar sign (\$) and consists of either a name:

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

or an expression in curly braces:

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

You can use templates both in raw and escaped strings. To insert the \$ character in a raw 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  
"""
```

Arrays

Arrays in Kotlin are represented by the `Array` class. It has get and set functions that turn into [] by operator overloading conventions, and the `size` property, along with other useful member functions:

```
class Array<T> private constructor() {  
    val size: Int  
    operator fun get(index: Int): T  
    operator fun set(index: Int, value: T): Unit  
  
    operator fun iterator(): Iterator<T>  
    // ...  
}
```

To create an array, use the function `arrayOf()` and pass the item values to it, so that `arrayOf(1, 2, 3)` creates an array [1, 2, 3]. Alternatively, the `arrayOfNulls()` function can be used to create an array of a given size filled with null elements.

Another option is to use the `Array` constructor that takes the array size and the function that returns values of array elements given its index:

```

fun main() {
//sampleStart
    // Creates an Array<String> with values ["0", "1", "4", "9", "16"]
    val asc = Array(5) { i -> (i * i).toString() }
    asc.forEach { println(it) }
//sampleEnd
}

```

As we said above, the [] operation stands for calls to member functions get() and set().

Arrays in Kotlin are invariant. This means that Kotlin does not let us assign an Array<String> to an Array<Any>, which prevents a possible runtime failure (but you can use Array<out Any>, see [Type Projections](#)).

Primitive type arrays

Kotlin also has classes that represent arrays of primitive types without boxing overhead: ByteArray, ShortArray, IntArray, and so on. These classes have no inheritance relation to the Array class, but they have the same set of methods and properties. Each of them also has a corresponding factory function:

```

val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]

```

```

// Array of int of size 5 with values [0, 0, 0, 0, 0]
val arr = IntArray(5)

// e.g. initialise the values in the array with a constant
// Array of int of size 5 with values [42, 42, 42, 42, 42]
val arr = IntArray(5) { 42 }

// e.g. initialise the values in the array using a lambda
// Array of int of size 5 with values [0, 1, 2, 3, 4] (values initialised to their index value)
var arr = IntArray(5) { it * 1 }

```

Type checks and casts

is and !is operators

To perform a runtime check 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 in Kotlin because the compiler tracks the `is`-checks and explicit casts for immutable values and inserts (safe) casts automatically when needed:

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x is automatically cast to String
    }
}
```

The compiler is smart enough to know a cast to be safe if a negative check leads to a return:

```
if (x !is String) return

print(x.length) // x is automatically cast to String
```

or in the right-hand side of `&&` and `||`:

```
// 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
}
```

Such smart casts work for when expressions and while loops as well:

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is IntArray -> print(x.sum())
}
```

Note that smart casts work only when the compiler can guarantee that the variable won't change between the check and the usage. More specifically, smart casts are applicable according to the following rules:

- val local variables - always except for local delegated properties.
- val properties - if the property is private or internal or the check is performed in the same module where the property is declared. Smart casts aren't applicable to open properties or properties that have custom getters.
- var local variables - if the variable is not modified between the check and the usage, is not captured in a lambda

that modifies it, and is not a local delegated property.

- var properties - never because the variable can be modified at any time by other code.

"Unsafe" cast operator

Usually, the cast operator throws an exception if the cast isn't possible. Thus, it's called unsafe. The unsafe cast in Kotlin is done by the infix operator as.

```
val x: String = y as String
```

Note that null cannot be cast to String as this type is not nullable. If y is null, the code above throws an exception. To make such code correct for null values, use the 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? that 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-null type String, the result of the cast is nullable.

Type erasure and generic type checks

Kotlin ensures type safety of operations involving generics at compile time, while, at runtime, instances of generic types don't hold information about their actual type arguments. For example, List<Foo> is erased to just List<*>. In general, there is no way to check whether an instance belongs to a generic type with certain type arguments at runtime.

Given that, the compiler prohibits is -checks that cannot be performed at runtime due to type erasure, such as ints is List<Int> or list is T (type parameter). You can, however, 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: List<String>) {
    if (list is ArrayList) {
        // `list` is smart-cast to `ArrayList<String>`
    }
}

```

The same syntax with omitted type arguments can be used for casts that do not take type arguments into account: list as ArrayList.

Inline functions with [reified type parameters](#) have their actual type arguments inlined at each call site. This enables arg is T checks for the type parameters, but 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

As said above, type erasure makes checking actual type arguments of a generic type instance impossible at runtime. Additionally, generic types in the code might be connected to each other not closely enough for the compiler to ensure type safety.

Even so, sometimes we have high-level program logic that implies type safety instead. For example:

```

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 that 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, there you could use interfaces DictionaryReader<T> and DictionaryWriter<T> with type-safe implementations for different types. You can introduce reasonable abstractions to move unchecked casts from calling code to the implementation details. Proper use of generic variance can also help.

For generic functions, using reified type parameters makes the casts such as 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 JVM: the array types (`Array<Foo>`) retain the information about the erased type of their elements, and the type casts to an array type are partially checked: the nullability and actual type arguments of the elements type are still erased. For example, the cast `foo as Array<List<String>?>` will succeed if `foo` is an array holding any `List<*>`, nullable or not.

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.

```
var max = a
if (a < b) max = b

// With else
var max: Int
if (a > b) {
```

```

        max = a
    } else {
        max = b
    }

// As expression
val max = if (a > b) a else b

```

Branches of if branches 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 -> { // Note the block
        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).

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) {
    parseInt(s) -> 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 of 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<>:
 - 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, for example: abc@, 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 at labels

With function literals, local functions and object expressions, functions can be nested in Kotlin. 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 this:

```
//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()  
}
```

The return -expression returns from the nearest enclosing function - 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)
    }
    print(" done with explicit label")
}
//sampleEnd

fun main() {
    foo()
}
```

Now, it returns only from the lambda expression. Oftentimes it is more convenient to use implicit labels: 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)
    }
    print(" 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(fun(value: Int) {
        if (value == 3) return // local return to the caller of the anonymous function - the
forEach loop
        print(value)
    })
    print(" done with anonymous function")
}
//sampleEnd

fun main() {
    foo()
}
```

Note that the use of local returns in 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" and not "return a labeled expression (@a 1)".

Exceptions

Exception classes

All exception classes in Kotlin inherit the class `Throwable`. 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. finally block may be omitted. However, at least one catch or finally block should be present.

Try is an expression

try is an expression; thus, it can have a return value:

```
val a: Int? = try { parseInt(input) } 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). Contents of the finally block do not 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.

The following is an example interface of the JDK implemented by StringBuilder class:

```
Appendable append(CharSequence csq) throws IOException;
```

What does this signature say? It says that every time I append a string to something (a StringBuilder, some kind of a log, a console, etc.) I have to catch those IOExceptions. Why? Because it might be performing IO (Writer also implements Appendable)... So it results in this kind of code all over the place:

```
try {
    Log.append(message)
} catch (IOException e) {
    // Must be safe
}
```

And this is no good, see [Effective Java, 3rd Edition](#), Item 77: Don't ignore exceptions.

Bruce Eckel says 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.

Other citations of this sort:

- [Java's checked exceptions were a mistake](#) (Rod Waldhoff)
- [The Trouble with Checked Exceptions](#) (Anders Hejlsberg)

If you want to alert callers of 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](#) as well as [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 type of the `throw` expression is the special type `Nothing`. The 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
```

Another case where you may encounter this type is 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](#) section 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 etc.) and the class body, surrounded by curly braces. Both the header and the body are optional; if the class has no body, curly braces can be omitted.

```
class Empty
```

Constructors

A class in Kotlin can have a primary constructor and one or more secondary constructors. The primary constructor is part of the class header: 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 cannot contain any code. Initialization code can be placed in initializer blocks, which are prefixed with the init keyword.

During an instance initialization, 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")
}
```

Parameters of the primary constructor 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 the same way as regular properties, the 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

The 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) {  
    var 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 as the first statement of a secondary constructor, so the code in all initializer blocks and property initializers is executed before the secondary constructor body.

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 parameters of the primary constructor 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:

```

val invoice = Invoice()

val customer = Customer("Joe Smith")

```

Kotlin does not have a new keyword.

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 and some of its members may be declared abstract. An abstract member does not have an implementation in its class. You don't need to annotate an abstract class or function with `open`.

You can override a non-abstract `open` member with an abstract one.

```
open class Polygon {  
    open fun draw() {}  
}  
  
abstract class Rectangle : Polygon() {  
    abstract override fun draw()  
}
```

Companion objects

If you need to write a function that can be called without having a class instance but needs access to the internals of a class (for example, 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`, that 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, they 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 right there, using the parameters of the primary constructor.

If the derived class has no primary constructor, then each secondary constructor has to initialize the base type using the `super` keyword, or to delegate to another constructor which does that. 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 were missing, the compiler would complain. If there is no `open` modifier on a function, like `Shape.fill()`, declaring a method with the same signature in a subclass is illegal, either with

override or without it. The open modifier has no effect when added on members of a final class (i.e.. a class with no open modifier).

A member marked override is itself open, i.e. 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

Overriding properties works in a similar way to overriding 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 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) and thus 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")
}

```

It means that, by the time of the base class constructor execution, the properties declared or overridden in the derived class are not yet initialized. If any of those properties are used in the base class initialization logic (either directly or indirectly, through another overridden open member implementation), it 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, and init blocks.

Calling the superclass implementation

Code in a derived class can call its superclass functions and property accessors 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 with 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 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, e.g. 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 its own implementation that eliminates 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 the following.

```

var <propertyName>[: <.PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]

```

The initializer, getter and setter are optional. Property type is optional if it can be inferred from the initializer (or from the getter 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:

```
val isEmpty: Boolean
    get() = this.size == 0
```

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.

You can omit the property type if it can be inferred from the getter:

```
val isEmpty get() = this.size == 0 // has type Boolean
```

If you need to change the visibility of an accessor or to annotate it, but don't need 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 can not 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, in the following case there will be no backing field:

```

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 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 the compile time, mark it as a compile time constant using the `const` modifier. Such properties need to fulfil the following requirements:

- Top-level, or member of an [object declaration or a companion object](#).
- Initialized with a value of type String or a primitive type
- No custom getter

Such properties can 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-null type must be initialized in the constructor. However, fairly often this is not convenient. For example, properties can be initialized through dependency injection, or in the setup method of a unit test. In this case, you cannot supply a non-null initializer in the constructor, but you still want to avoid null checks when referencing the property inside the body of a class.

To handle this case, 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  
    }  
}
```

The 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-null, 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 the properties that are lexically accessible, when declared in the same type or 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 properties simply reads from (and maybe writes to) a backing field. On the other hand, with custom getters and setters one can implement any behaviour of a property. Somewhere in between, there are certain common patterns of how a property may work. A few examples: lazy values, reading from a map by a given key, accessing a database, notifying listener on access, etc.

Such common behaviours 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 to 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 it can 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 and thus 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. See the following example.

```
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 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, obviously, have to override bar() and provide an implementation.

However, if you derive D from A and B, you need to implement all the methods which you have inherited from multiple interfaces, and to specify how exactly D should implement them. This rule applies both to methods for which you've inherited a single implementation (bar()) and 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](#).

Functional interfaces vs. type aliases

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, functions, properties and their setters can have visibility modifiers. Getters always have the same visibility as the property.

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 and 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 do not specify any visibility modifier, public is used by default, which means that your declarations will be visible everywhere.
- If you mark a declaration private, it will only be visible inside the file containing the declaration.
- If you mark it internal, it is visible everywhere in the same [module](#).
- protected is not available for top-level declarations.

To use a visible top-level declaration from another package, you should still [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

```

Classes and interfaces

For members declared inside a class:

- `private` means visible inside this class only (including all its members).
- `protected` is the same as `private` but 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, outer class does not see private members of its inner classes.

If you override a protected member and do not specify the visibility explicitly, the overriding member will also have protected visibility.

Examples:

```

open class Outer {
    private val a = 1
    protected open val b = 2
    internal 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
}

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

To specify a visibility of the primary constructor of a class, use the following syntax:

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 where the class is visible (this means that a constructor of an internal class is only visible within the same module).

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:

- 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 an ability to extend a class 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 from a third-party library that you can't modify. Such functions are available for calling in the usual way as if they were methods of the original class. This mechanism is called extension functions. 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, that means 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 declare the generic type parameter before the function name to make it available in the receiver type expression. See [Generic functions](#).

Extensions are resolved statically

Extensions do not actually modify classes they extend. By defining an extension, you do not insert new members into a class, but merely make new functions callable with the dot-notation on variables of this type.

Extension functions are dispatched statically, that means that they are not virtual by receiver type. An extension function being called is determined by the type of the expression on which the function is invoked, not by the type of the result of evaluating that expression at runtime. 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 called extension function 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 which 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") }

    Example().printFunctionType(1)
//sampleEnd
}

```

Nullable receiver

Note that extensions can be defined with a nullable receiver type. Such extensions can be called on an object variable even if its value is null, and can check for this == null inside the body.

This way you can call `toString()` in Kotlin without checking for null: the check happens inside the extension function.

```
fun Any?.toString(): String {
    if (this == null) return "null"
    // after the null check, 'this' is autocast to a non-null type, so the toString() below
    // resolves to the member function of the Any class
    return toString()
}
```

Extension properties

Similarly to functions, Kotlin supports extension properties:

```
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 such 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

Inside a class, you can declare extensions for another class. Inside such an extension, there are multiple implicit receivers- objects members of which can be accessed without a qualifier. The instance of the class in which the extension is declared is called dispatch receiver, and the instance of the receiver type of the extension method is called 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(443) // error, the extension function is
    unavailable outside Connection
}
```

In case of a name conflict between the members of the dispatch receiver and the 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 of other entities](#) 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, such an extension cannot access the receiver's private members.

Data classes

You often create classes whose main purpose is to hold data. In such classes, some standard functionality and utility functions are often mechanically derivable from the data. In Kotlin, these are called data classes and 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 needs to have at least one parameter.
- All primary constructor parameters need to be marked as `val` or `var`.
- Data classes cannot be abstract, open, sealed or inner.

Additionally, the members generation 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 the `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 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 all 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  
}
```

Only the property name will be used inside the `toString()`, `equals()`, `hashCode()`, and `copy()` implementations, and there will only be one component function `component1()`. While two `Person` objects can have different ages, they will be treated as equal.

```
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  
    //sampleEnd  
    println("person1 == person2: ${person1 == person2}")  
    println("person1 with age ${person1.age}: ${person1}")  
    println("person2 with age ${person2.age}: ${person2}")  
}
```

Copying

To copy an object for changing some of its properties, but keeping the rest unchanged, use the `copy()` function. For the `User` class above, its implementation would be as follows:

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

You can 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 enable their use in [destructuring declarations](#):

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // prints "Jane, 35 years of age"
```

Standard data classes

The standard library provides `Pair` and `Triple`. In most cases, though, named data classes are a better design choice, because they make the code more readable by providing meaningful names for properties.

Sealed classes

Sealed classes and interfaces represent restricted class hierarchies that provide more control over inheritance. All subclasses of a sealed class are known at compile time. No other subclasses may appear after a module with the sealed class is compiled. For example, third-party clients can't extend your sealed class in their code. Thus, each instance of a sealed class has a type from a limited set that is known when this class is compiled.

The same works for sealed interfaces and their implementations: once a module with a sealed interface is compiled, no new implementations can appear.

In some sense, sealed classes are similar to [enum](#) classes: the set of values for an enum type is also restricted, but each enum constant exists only as a single instance, whereas a subclass of a sealed class can have multiple instances, each with its own state.

To declare a sealed class or interface, put the `sealed` modifier before its name:

```
sealed interface Expr

sealed class MathExpr(): Expr

data class Const(val number: Double) : MathExpr()
data class Sum(val e1: Expr, val e2: Expr) : MathExpr()
object NotANumber : Expr
```

A sealed class is [abstract](#) by itself, it cannot be instantiated directly and can have abstract members.

Constructors of sealed classes can have one of two visibilities: protected (by default) or private:

```
sealed class MathExpr {  
    constructor() { /*...*/ } // protected by default  
    private constructor(vararg operands: Number): this() { /*...*/ } // private is OK  
    // public constructor(s: String): this() {} // Error: public and internal are not allowed  
}
```

Location of direct subclasses

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 proper qualified name. They can't be local nor anonymous objects.

enum classes can't extend a sealed class (as well as any other class), but they can implement sealed interfaces.

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 expect 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.](#)

Sealed classes and when expression

The key benefit of using sealed classes comes into play when you use them in a when expression. If it's possible to verify that the statement covers all cases, you don't need to add an else clause to the statement. However, this works only if you use when as an expression (using the result) and not as a statement:

```
fun eval(expr: Expr): Double = when(expr) {  
    is Const -> expr.number  
    is Sum -> eval(expr.e1) + eval(expr.e2)  
    NotANumber -> Double.NaN  
    // the `else` clause is not required because we've covered all the cases  
}
```

when expressions on `expect` sealed classes in the common code of multiplatform projects still require an else branch. This happens because subclasses of actual platform implementations aren't known in the common code.

Generics: in, out, where

As in Java, classes in Kotlin can have type parameters:

```
class Box<T>(t: T) {  
    var value = t  
}
```

In general, to create an instance of such a class, provide the type arguments:

```
val box: Box<Int> = Box<Int>(1)
```

But if the parameters may be inferred, for example, from the constructor arguments or by some other means, 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 most tricky parts of Java's type system is wildcard types (see [Java Generics FAQ](#)). And Kotlin doesn't have any. Instead, it has two other things: declaration-site variance and type projections.

First, let's think about why Java needs those mysterious wildcards. The problem is explained in [Effective Java, 3rd Edition](#), Item 31: Use bounded wildcards to increase API flexibility. First, generic types in Java are invariant, meaning that `List<String>` is not a subtype of `List<Object>`. Why so? If `List` was not invariant, it would have been no better than Java's arrays, since the following code would have compiled and caused an exception at runtime:

```
// Java  
List<String> strs = new ArrayList<String>();  
List<Object> objs = strs; // !!! A compile-time error here saves us from a runtime exception  
later.  
objs.add(1); // Put an Integer into a list of Strings  
String s = strs.get(0); // !!! ClassCastException: Cannot cast Integer to String
```

So, Java prohibits such things in order to guarantee run-time safety. But this has some implications. For example, consider the `addAll()` method from `Collection` interface. What's the signature of this method? Intuitively, you'd put it this way:

```
// Java
interface Collection<E> ... {
    void addAll(Collection<E> items);
}
```

But then, you are not able to do the following simple thing (which is perfectly safe):

```
// Java
void copyAll(Collection<Object> to, Collection<String> from) {
    to.addAll(from);
    // !!! Would not compile with the naive declaration of addAll:
    // Collection<String> is not a subtype of Collection<Object>
}
```

(In Java, you learned this lesson the hard way, see [Effective Java, 3rd Edition, Item 28: Prefer lists to arrays](#))

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 some 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 since you don't know what objects comply to that unknown subtype of `E`. In return for this limitation, you have the desired behaviour: `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 trick works is rather simple: if you can only take items from a collection, then using a collection of `String`s and reading `Object`s from it is fine. Conversely, if you can only put items into the collection, it's OK to take a collection of `Object`s and put `String`s into it: there is `List<? super String>` in Java, a supertype of `List<Object>`.

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 an `Object`.

Joshua Bloch calls those objects you only read from Producers, and those you only write to Consumers. He recommends:

"For maximum flexibility, use wildcard types on input parameters that represent producers or consumers", and 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 this object is immutable: for example, nothing prevents you from calling `clear()` to remove all 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>`, which 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: when a type parameter `T` of a class `C` is declared `out`, it may occur only in `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: 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-explaining (as they were successfully used in C# for quite some time already), thus the mnemonic mentioned above is not really needed, and one can rephrase it for a higher purpose:

The Existential Transformation: Consumer in, Producer out! :-)

Type projections

Use-site variance: type projections

It is very convenient 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 cannot be either co- or 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`, thus neither of `Array<Int>` and `Array<Any>` is a subtype of the other. Why? Again, because `copy` may 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 sometime later.

To prohibit the `copy` function from writing to `from`, do the following:

```
fun copy(from: Array<out Any>, to: Array<Any>) { ... }
```

This is type projection, that means that `from` is not simply an array, but a restricted (projected) one. You can only call those methods that return the type parameter `T`, in this case it means that you can only call `get()`. This is our approach to use-site variance, and corresponds to Java's `Array<? extends Object>`, but in a slightly simpler way.

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 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 would 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>`. It 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>`. It 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 can imagine 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

Not only classes 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 that corresponds to Java's extends keyword:

```
fun <T : Comparable<T>> sort(list: List<T>) { ... }
```

The type specified after a colon is the upper bound: 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 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.

Type erasure

The type safety checks that Kotlin performs for generic declaration usages are only 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<*>.

Therefore, 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.

Type casts to generic types with concrete type arguments, for example, 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. The compiler issues a warning on unchecked casts, and at runtime, only the non-generic part is checked (equivalent to foo as List<*>).

The type arguments of a 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.

However, reified type parameters of inline functions are substituted by the actual type arguments in the inlined function body at the call sites and thus can be used for type checks and casts, with the same restrictions for instances of generic types as described above.

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 usage of enum classes is implementing type-safe enums:

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

Each enum constant is an object. Enum constants are separated with 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 also declare their own anonymous classes with their corresponding methods, as well as 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 enum constant definitions from the member definitions with a semicolon.

Implementing interfaces in enum classes

An enum class can implement an interface (but not derive from a class), providing either a single interface members implementation for all of the entries, or separate ones for each entry within its anonymous class. This is done by adding the interfaces 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.values()) {
        println("$f($a, $b) = ${f.apply(a, b)}")
    }
}
```

Working with enum constants

Enum classes in Kotlin have synthetic 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.values(): Array<EnumClass>
```

The `valueOf()` method throws an `IllegalArgumentException` if the specified name does not match any of the enum constants defined in the class.

You can access the constants in an enum class in a generic way, using the `enumValues<T>()` and `enumValueOf<T>()` functions:

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

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

```
printAllValues<RGB>() // prints RED, GREEN, BLUE
```

Every enum constant has properties to obtain its name and position in the enum class declaration:

```
val name: String  
val ordinal: Int
```

The enum constants also implement the [Comparable](#) interface, with the natural order being the order in which they are defined in the enum class.

Inline classes

Sometimes it is necessary for business logic to create a wrapper around some type. However, it introduces runtime overhead due to additional heap allocations. Moreover, if the wrapped type is primitive, the performance hit is terrible, 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)
```

The inline modifier for inline classes is deprecated.

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, and have the init block:

```
@JvmInline
value class Name(val s: String) {
    init {
        require(s.length > 0) { }
    }

    val length: Int
        get() = s.length

    fun greet() {
        println("Hello, $s")
    }
}

fun main() {
    val name = Name("Kotlin")
    name.greet() // method `greet` is called as a static method
    println(name.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 must be 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.

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.

The mangling scheme has been changed in Kotlin 1.4.30. 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.

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
}

```

Object expressions and declarations

Sometimes you need to create an object of a slight modification of some class, without explicitly declaring a new subclass for it. Kotlin handles this case 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 handy 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 need just an object with no 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"
    }
//sampleEnd
    print(helloWorld)
}

```

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 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, appropriate constructor parameters must be passed 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 object 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

Singleton can be useful in several cases, and Kotlin (after Scala) 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 cannot be used on the right hand side of an assignment statement.

Object declaration's initialization is thread-safe and done at 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 (i.e. be nested directly inside a function), but they can be nested into other object declarations or non-inner classes.

Companion objects

An object declaration inside a class can be marked with the companion keyword:

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

Members of the companion object can be called by using simply 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 fields of a 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 details.

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), matching 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

There are certain common kinds of properties, that, though you can implement them manually every time you need them, it would be helpful to implement them once and add to a library. Examples include:

- Lazy properties: the value gets computed only upon first access.
- Observable properties: listeners get 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 `get()` (and `set()`) corresponding to the property will be delegated to its `getValue()` and `setValue()` methods. Property delegates don't have to implement any interface, but they have to provide a `getValue()` function (and `setValue()` --- for `var`s).

For example:

```

import kotlin.reflect.KProperty

class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, thank you for delegating '${property.name}' to me!"
    }
}

```

```

    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
        println("$value has been assigned to '${property.name}' in $thisRef.")
    }
}

```

When you read from p that delegates to an instance of Delegate, the getValue() function from Delegate is called, so that 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](#).

Note that you can declare a delegated property inside a function or code block, it shouldn't necessarily 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, and all threads will see the same value. If the synchronization of initialization delegate is not required, so that multiple threads can execute it simultaneously, pass `LazyThreadSafetyMode.PUBLICATION` as a parameter to the `lazy()` function.

And if you're sure that the initialization will always happen on 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 the 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: a property being assigned to, the old value and the new one:

```
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 the `vetoable` is 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 proper `::` 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: you 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 like parsing JSON or doing other “dynamic” things. 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 (by the string keys – 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 works also for var's properties if you use a MutableMap instead of 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 the first access only. If someCondition fails, the variable won't be computed at all.

Property delegate requirements

Here are requirements for delegating properties.

For a read-only property (val), a delegate should provide an operator function getValue() with the following parameters:

- thisRef must be the same 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 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 extension functions. The latter is handy when you need to delegate property to an object which 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 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()`. Thus, you can pass a `ReadWriteProperty` whenever a `ReadOnlyProperty` is expected.

```
fun resourceDelegate(): ReadWriteProperty<Any?, Int> =
    object : ReadWriteProperty<Any?, Int> {
```

```

        var curValue = 0
        override fun getValue(thisRef: Any?, property: KProperty<*>): Int = curValue
        override fun setValue(thisRef: Any?, property: KProperty<*>, value: Int) {
            curValue = value
        }
    }

val readOnly: Int by resourceDelegate() // ReadWriteProperty as val
var readWrite: Int by resourceDelegate()

```

Translation rules

Under the hood for every delegated property the Kotlin compiler generates an auxiliary property and delegates to it. For example, for the property prop the hidden property prop\$delegate is generated, 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.

Providing a delegate

By defining the provideDelegate operator you can extend the logic of 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, if you want 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 for getValue:

- thisRef must be the same 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 getter or 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

Functions in Kotlin are declared using the fun keyword:

```
fun double(x: Int): Int {
    return 2 * x
}
```

Function usage

Calling functions uses the traditional approach:

```
val result = double(2)
```

Calling member functions uses the 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. 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 a number of overloads compared to other languages:

```
fun read(  
    b: ByteArray,  
    off: Int = 0,  
    len: Int = b.size,  
) { /*...*/ }
```

A default value is defined using the `=` after the type.

Overriding methods always use the same default parameter values as the base method. When overriding a method with 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

When calling a function, you can name one or more of its arguments. This may be helpful when a function has a large number of 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 they are listed in, and if you want to use their default values you can just leave them out altogether.

Consider the following function `reformat()` that 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 arguments with default values:

```
reformat("This is a long String!")
```

You can skip some arguments with default values. 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"))
```

On the JVM: You can't use the named argument syntax when calling Java functions because Java bytecode does not always preserve names of function parameters.

Unit-returning functions

If a function does not return any 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 a function returns a single expression, the curly braces can be omitted and the body is specified after a = 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 it 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, i.e. the ts variable in the example above 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 following parameters can be passed using the named argument syntax, or, if the parameter has a function type, by passing a lambda outside parentheses.

When you call a vararg -function, you can pass arguments one-by-one, 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)
```

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 the 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, infix function call's precedence is higher than that of the boolean operators && and ||, is-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# or Scala. 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 another function:

```
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, the 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 a risk of stack overflow. When a function is marked with the

tailrec modifier and meets the required form, 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 doesn't change anymore, 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, and you cannot use it within `try/ catch/ finally` blocks. Currently, tail recursion is supported by Kotlin for JVM and Kotlin/Native.

See also:

- [Inline functions](#)
- [Extension functions](#)
- [Higher-order functions and lambdas](#)

High-order functions and lambdas

Kotlin functions are [first-class](#), which means that they can be stored in variables and data structures, passed as arguments to and returned from other [higher-order functions](#). You can operate with functions in any way that is 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 is the [functional programming idiom](#) `fold` for collections, which takes an initial accumulator value and a combining function and builds its return value by consecutively combining current accumulator value with each collection element, replacing the accumulator:

```
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 parameter `combine` has a [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 it an [instance of the function type](#) 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")
}
```

The following sections explain in more detail the concepts mentioned so far.

Function types

Kotlin uses a family of function types like `(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 parameter types list and a return type: `(A, B) -> C` denotes a type that represents functions taking two arguments of types A and B and returning a value of type C. The parameter types list 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 a dot in the notation: the type `A.(B) -> C` represents functions that can be called on a receiver object of A with a parameter of B and return a value of C. Function literals with receiver are often used along with these types.
- Suspending functions belong to function types of a special kind, which have a suspend modifier in the 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: `((Int, Int) -> Int)?`.

Function types can 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:

- Using a code block within a function literal, in one of the 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.

- Using 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.

- Using 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 receiver are interchangeable, so that 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 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 passed immediately as an expression. Consider the following example:

```
max(strings, { a, b -> a.length < b.length })
```

Function max is a higher-order function, it takes a function value as the 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 an -> sign.
- 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

In Kotlin, there is a 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 to that call, the parentheses can be omitted entirely:

```
run { println("...") }
```

it: implicit name of a single parameter

It's very common that a lambda expression has only one parameter.

If the compiler can figure the signature out itself, it is allowed not to declare the only parameter and omit `->`. 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 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

One thing missing from the lambda expression syntax presented above is the ability to specify the return type of the function. 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 that 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 that the parameter types can be omitted if they can be inferred from 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 and has to be specified explicitly (or is assumed to be Unit) for anonymous functions with a block body.

Anonymous function parameters are always passed inside the parentheses. The shorthand syntax allowing to leave the function outside the parentheses works only for lambda expressions.

One other 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 said above, Kotlin provides the ability to [call an instance](#) of a function type with receiver 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 [extension functions](#), which also allow you to access the members of the receiver object inside the body of the function.

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 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 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 means those variables that are 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, you need to 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

In case you want only some of the lambdas passed to an inline function to be inlined, you can 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 the inline functions or passed as inlinable arguments, but `noinline` ones can be manipulated in any way you like such as 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 suppress the warning if you are sure the inlining is needed using the annotation `@Suppress("NOTHING_TO_INLINE")`).

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 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 needs to be marked 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 if a node has a certain type. It's all fine, but the call site is not very pretty:

```
treeNode.findParentOfType(MyTreeNode::class.java)
```

What you actually want is simply pass a type to this function. You can call it like this:

```
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?
}
```

You qualified 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, normal operators like `!is` and `as` are working now. Also, you can call it as mentioned 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 a backing field. 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 and 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 non-public API of a module, the 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.

Null safety

Nullable types and non-null types

Kotlin's type system is aimed at eliminating the danger of null references from code, also known as the [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 NPE for short.

Kotlin's type system is aimed to eliminate `NullPointerException`'s from our code. The only possible causes of NPE's may be:

- An explicit call to throw `NullPointerException()`.
- Usage of the `!!` operator that is described below.
- Some data inconsistency with regard to initialization, such as when:
 - An uninitialized this available in a constructor is passed and used somewhere ("leaking this").
 - [A superclass constructor calls an open member](#) whose implementation in the derived class uses uninitialized state.
- Java interoperation:
 - Attempts to access a member on a null reference of a [platform type](#);

- Generic types used for Java interoperation with incorrect nullability, for example, a piece of Java code might add null into a Kotlin MutableList<String>, meaning that MutableList<String?> should be used 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-null references). For example, a regular variable of type String cannot hold null:

```
fun main() {
//sampleStart
    var a: String = "abc" // Regular initialization means non-null by default
    a = null // compilation error
//sampleEnd
}
```

To allow nulls, you can declare a variable as nullable string, written String?:

```
fun main() {
//sampleStart
    var b: String? = "abc" // can be set 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 of doing that.

Checking for null in conditions

First, you can explicitly check if 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 (that means a local variable which is not modified between the check and the usage or a member val which has a backing field and is not overridable), because otherwise it might happen that b changes to null after the check.

Safe calls

Your second option is the safe call operator, written ?:.

```
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, if Bob, an Employee, may be assigned to a Department (or not), that in turn may have another Employee as a department head, then to obtain the name of Bob's department head (if any), 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()
```

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
```

Along with the complete if -expression, this can be expressed with the Elvis operator, written ?::

```
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 right-hand side expression 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 very handy, for example, for 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-null 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 does not appear out of the blue.

Safe casts

Regular casts may result into 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-null elements, you can do so by using `filterNotNull`:

```
val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()
```

Equality

In Kotlin there are two types of equality:

- Structural equality (== - a check for `equals()`)
- Referential equality (==== - two references point to the same object)

Structural equality

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.

Note that there's no point in optimizing your code when comparing to null explicitly: `a == null` will be automatically translated to `a === null`.

To provide a custom `equals` check implementation, override the `equals(other: Any?): Boolean` function. 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 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. For values represented by primitive types at runtime (for example, Int), the === equality check is equivalent to the == check.

Floating-point numbers equality

When an equality check operands are statically known to be Float or Double (nullable or not), the check follows the [IEEE 754 Standard for Floating-Point Arithmetic](#).

Otherwise, the structural equality is used, which disagrees with the standard so that NaN is equal to itself, and -0.0 is not equal to 0.0.

See: [Floating-point numbers comparison](#).

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
            }

            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 printLine() { println("Top-level function") }

    class A {
        fun printLine() { println("Member function") }

        fun invokePrintLine(omitThis: Boolean = false) {
            if (omitThis) printLine()
            else this.printLine()
        }
    }

    A().invokePrintLine() // Member function
    A().invokePrintLine(omitThis = true) // Top-level function
//sampleEnd()
}
```

Coroutines

Asynchronous or non-blocking programming is the new reality. 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 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.

As a bonus, coroutines not only open the doors to asynchronous programming, but 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](#)

Sample projects

- [kotlinx.coroutines examples and sources](#)
- [KotlinConf app](#)

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
- Programmatically laying out UI components: [Anko](#)
- 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: @DsIMarker

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
}

```

Operator overloading

Kotlin allows you to provide custom implementations for the predefined set of operators on types. These operators have predefined symbolic representation (like `+` or `*`) and precedence. To implement an operator, provide a [member function](#) or an [extension function](#) with a specific name for the corresponding type. This type becomes the left-hand side type for binary operations and the argument type for the unary ones.

To overload an operator, mark the corresponding function with the `operator` modifier:

```

interface IndexedContainer {
    operator fun get(index: Int)
}

```

When [overriding](#) your operator overloads, you can omit operator:

```

class OrdersList: IndexedContainer {

```

```
override fun get(index: Int) { /*...*/ }
```

Unary operations

Unary prefix operators

Expression	Translated to
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()

This table says that when the compiler processes, for example, an expression +a, it performs the following steps:

- Determines the type of a, let it be T.
- Looks up a function unaryPlus() with the operator modifier and no parameters for the receiver T, that means a member function or an extension function.
- If the function is absent or ambiguous, it is a compilation error.
- If the function is present and its return type is R, the expression +a has type R.

These operations, as well as all the others, are optimized for basic types and do not introduce overhead of function calls for them.

As an example, here's how you can overload the unary minus operator:

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

val point = Point(10, 20)

fun main() {
    println(-point) // prints "Point(x=-10, y=-20)"
}
```

Increments and decrements

Expression Translated to

a++ a.inc() + see below

a-- a.dec() + see below

The inc() and dec() functions must return a value, which will be assigned to the variable on which the ++ or -- operation was used. They shouldn't mutate the object on which the inc or dec was invoked.

The compiler performs the following steps for resolution of an operator in the postfix form, for example a++:

- Determines the type of a, let it be T.
- Looks up a function inc() with the operator modifier and no parameters, applicable to the receiver of type T.
- Checks that the return type of the function is a subtype of T.

The effect of computing the expression is:

- Store the initial value of a to a temporary storage a0.
- Assign the result of a0.inc() to a.
- Return a0 as the result of the expression.

For a-- the steps are completely analogous.

For the prefix forms ++a and --a resolution works the same way, and the effect is:

- Assign the result of a.inc() to a.
- Return the new value of a as a result of the expression.

Binary operations

Arithmetic operators

Expression Translated to

a + b a.plus(b)

a - b a.minus(b)

a * b a.times(b)

Expression Translated to

a / b a.div(b)

a % b a.rem(b)

a..b a.rangeTo(b)

For the operations in this table, the compiler just resolves the expression in the Translated to column.

Below is an example Counter class that starts at a given value and can be incremented using the overloaded + operator:

```
data class Counter(val dayIndex: Int) {  
    operator fun plus(increment: Int): Counter {  
        return Counter(dayIndex + increment)  
    }  
}
```

in operator

Expression Translated to

a in b b.contains(a)

a !in b !b.contains(a)

For in and !in the procedure is the same, but the order of arguments is reversed.

Indexed access operator

Expression Translated to

a[i] a.get(i)

a[i, j] a.get(i, j)

a[i_1, ..., i_n] a.get(i_1, ..., i_n)

a[i] = b a.set(i, b)

a[i, j] = b a.set(i, j, b)

Expression	Translated to
------------	---------------

a[i_1, ..., i_n] = b a.set(i_1, ..., i_n, b)

Square brackets are translated to calls to get and set with appropriate numbers of arguments.

invoke operator

Expression	Translated to
------------	---------------

a() a.invoke()

a(i) a.invoke(i)

a(i, j) a.invoke(i, j)

a(i_1, ..., i_n) a.invoke(i_1, ..., i_n)

Parentheses are translated to calls to invoke with appropriate number of arguments.

Augmented assignments

Expression	Translated to
------------	---------------

a += b a.plusAssign(b)

a -= b a.minusAssign(b)

a *= b a.timesAssign(b)

a /= b a.divAssign(b)

a %= b a.remAssign(b)

For the assignment operations, for example a += b, the compiler performs the following steps:

- If the function from the right column is available:
 - If the corresponding binary function (that means plus() for plusAssign()) is available too, report 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](#).

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.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
```

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(ann: Ann) {  
    val i = ann.value  
}
```

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 for introspecting the structure of your own program at runtime. Kotlin makes functions and properties first-class citizens in the language, and introspecting them (for example, learning a name or a type of a property or function at runtime) is closely intertwined with simply using a functional or reactive style.

Kotlin/JS provides the limited support for reflection features. [Learn more about reflection in Kotlin/JS.](#)

JVM dependency

On the JVM platform, the runtime component required for using the reflection features is distributed as a separate artifact kotlin-reflect.jar in the Kotlin compiler distribution. 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:

Groovy

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-reflect:1.5.20"  
}
```

Kotlin

```
dependencies {  
    implementation("org.jetbrains.kotlin:kotlin-reflect:1.5.20")  
}
```

- In Maven:

```
<dependencies>  
    <dependency>  
        <groupId>org.jetbrains.kotlin</groupId>  
        <artifactId>kotlin-reflect</artifactId>  
    </dependency>  
</dependencies>
```

If you don't use Gradle or Maven, make sure you have kotlin-reflect.jar in the classpath of your project. In other supported cases (IntelliJ IDEA projects, using command-line compiler or Ant), it is added by default. In command-line compiler and Ant, you can use -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 value of type [KClass](#).

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 a 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 obtain the reference to an exact class of an object, for instance `GoodWidget` or `BadWidget`, despite the type of the receiver expression (`Widget`).

Callable references

References to functions, properties, and constructors, apart from introspecting the program structure, 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, which is the property type for properties, and the constructed type for constructors.

Function references

When you have a named function declared like this:

```
fun isOdd(x: Int) = x % 2 != 0
```

You can call it directly (`isOdd(5)`). Alternatively, you can use it as a function type value: pass it to another function. To do this, 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 example, `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`.

Note that even if you initialize a variable with a reference to an extension function, the inferred function type will have no receiver (it will have an additional parameter accepting a receiver object). To have a function type with 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(*)).` Now, you can apply it 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 property object of type `KProperty<Int>`. 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 of type `KMutableProperty<Int>`, 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:

```
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, 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 corresponding 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 an object of function type is expected 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 reference is bound to its receiver. It can be called directly (like in the example above) or used whenever an expression of function type 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 bound and the corresponding unbound references. 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
```

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
```

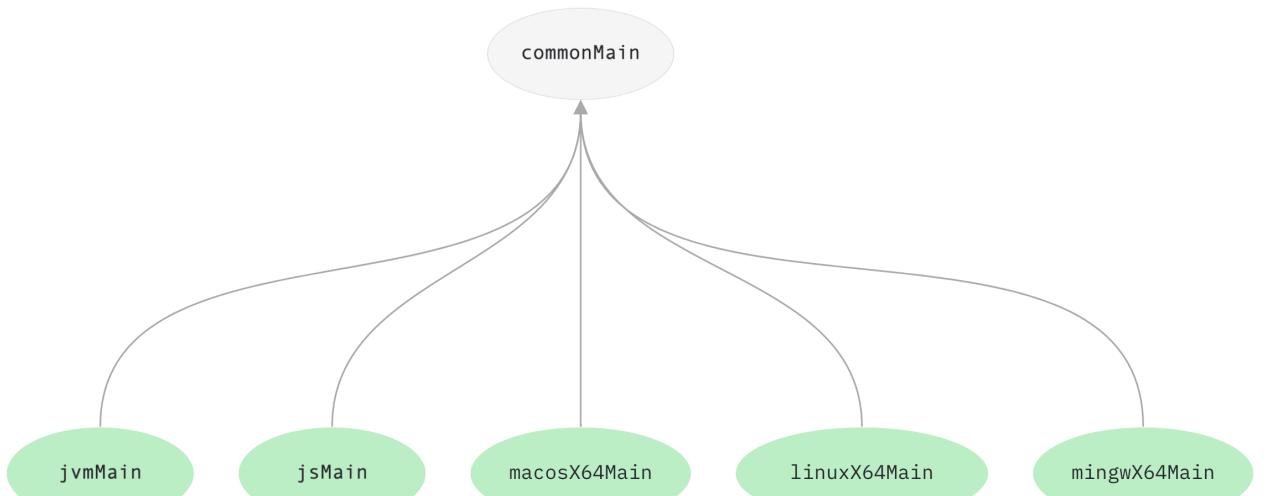
Kotlin Multiplatform

Multiplatform projects are in [Alpha](#). Language features and tooling may change in future Kotlin versions.

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](#).

With Kotlin Multiplatform, share the code using the mechanisms Kotlin provides:

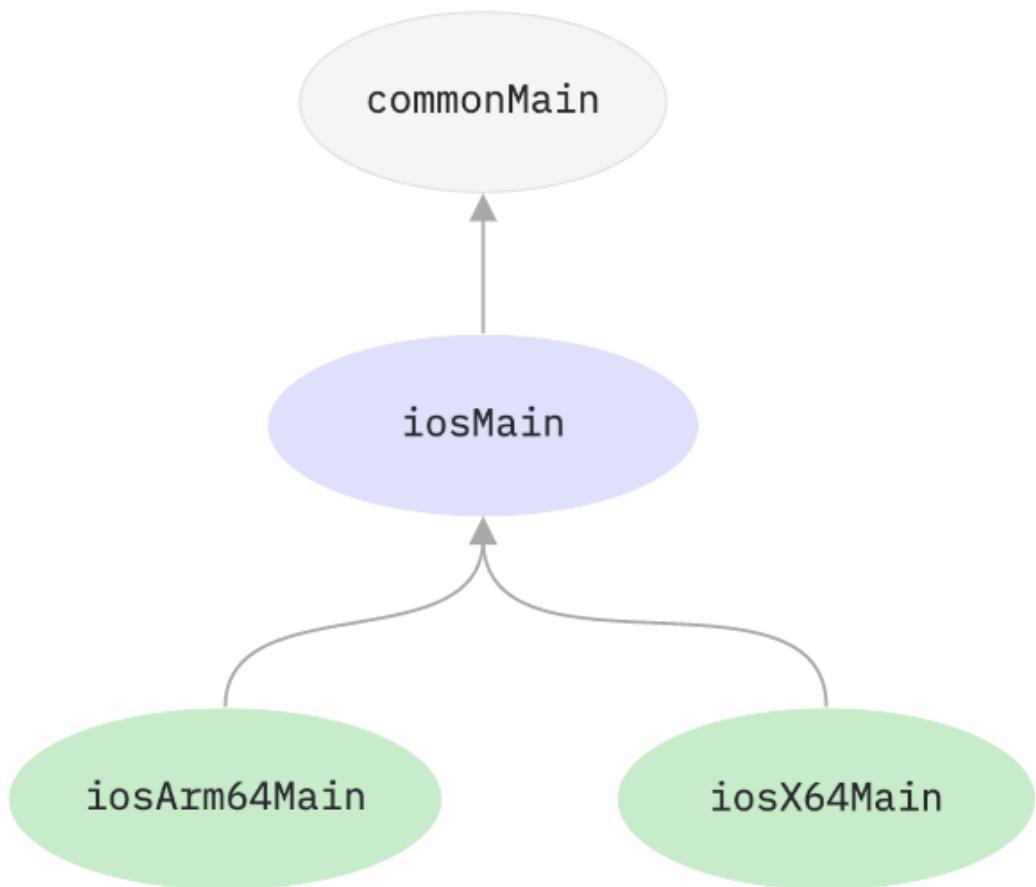
- [Share code among all platforms used in your project](#). Use it for sharing the common business logic that applies to all platforms.



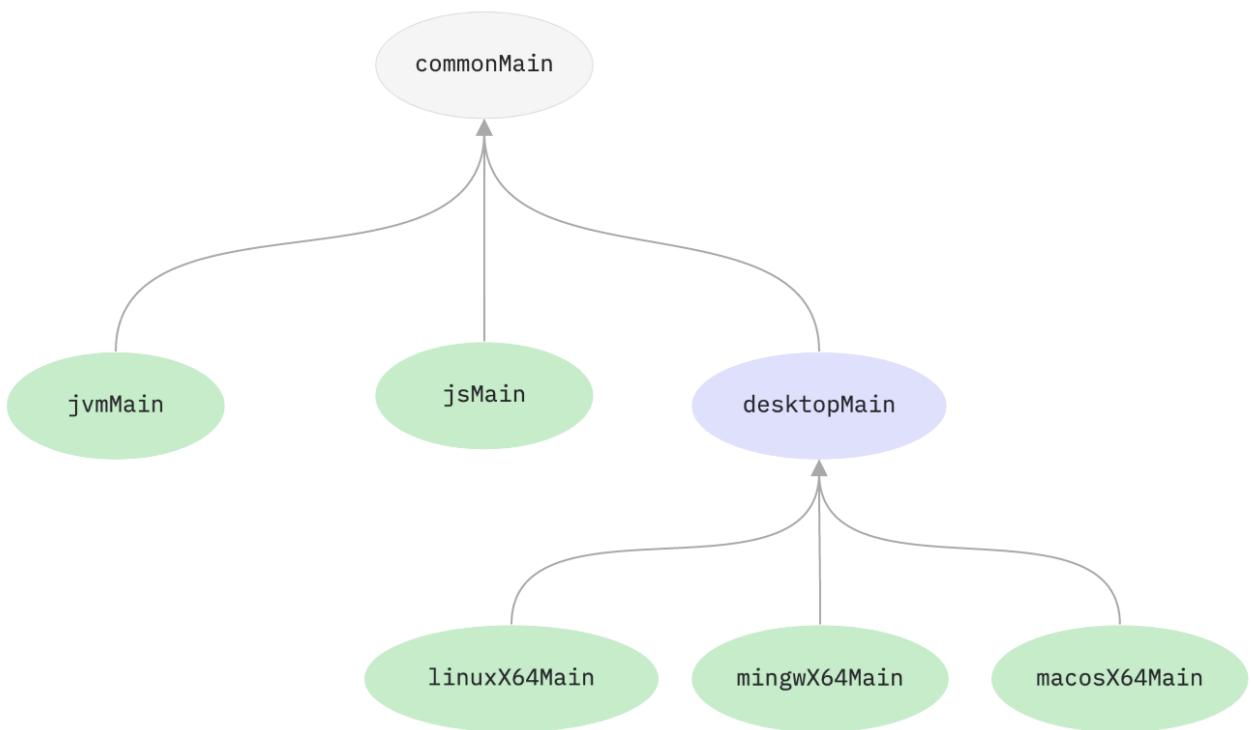
Code shared for all platforms

- [Share code among some platforms included in your project but not all](#). You can reuse much of the code in similar platforms using a hierarchical structure. You can use [target shortcuts](#) for common combinations of targets or [create](#)

the hierarchical structure manually.



Code shared for iOS targets



Hierarchical structure

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

Tutorials

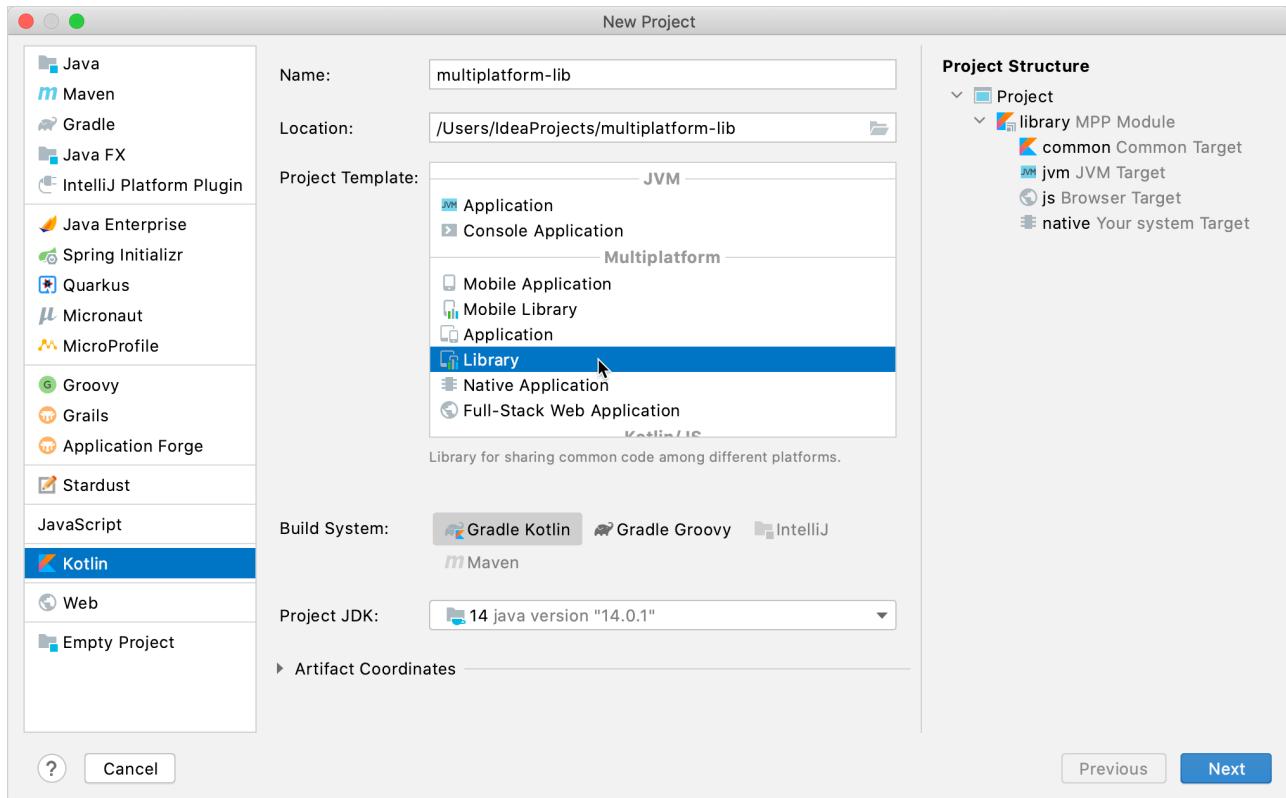
- [Creating a multiplatform Kotlin library](#) teaches how to create a multiplatform library available for JVM, JS, and Native and which can be used from any other common code (for example, shared with Android and iOS). It also shows how to write tests which will be executed on all platforms and use an efficient implementation provided by a specific platform.
- [Building a Full Stack Web App with Kotlin Multiplatform](#) teaches the concepts behind building an application that targets Kotlin/JVM and Kotlin/JS by building a client-server application that makes use of shared code, serialization, and other multiplatform paradigms. It also provides a brief introduction to working with Ktor both as a server- and client-side framework.

Create a multiplatform library

This section provides steps for creating a multiplatform library. You can also complete the [tutorial](#) where you will create a multiplatform library, test it, and publish it to Maven.

1. In IntelliJ IDEA, select File | New | Project.

2. In the panel on the left, select Kotlin.
3. Enter a project name, then in the Multiplatform section select Library as the project template.



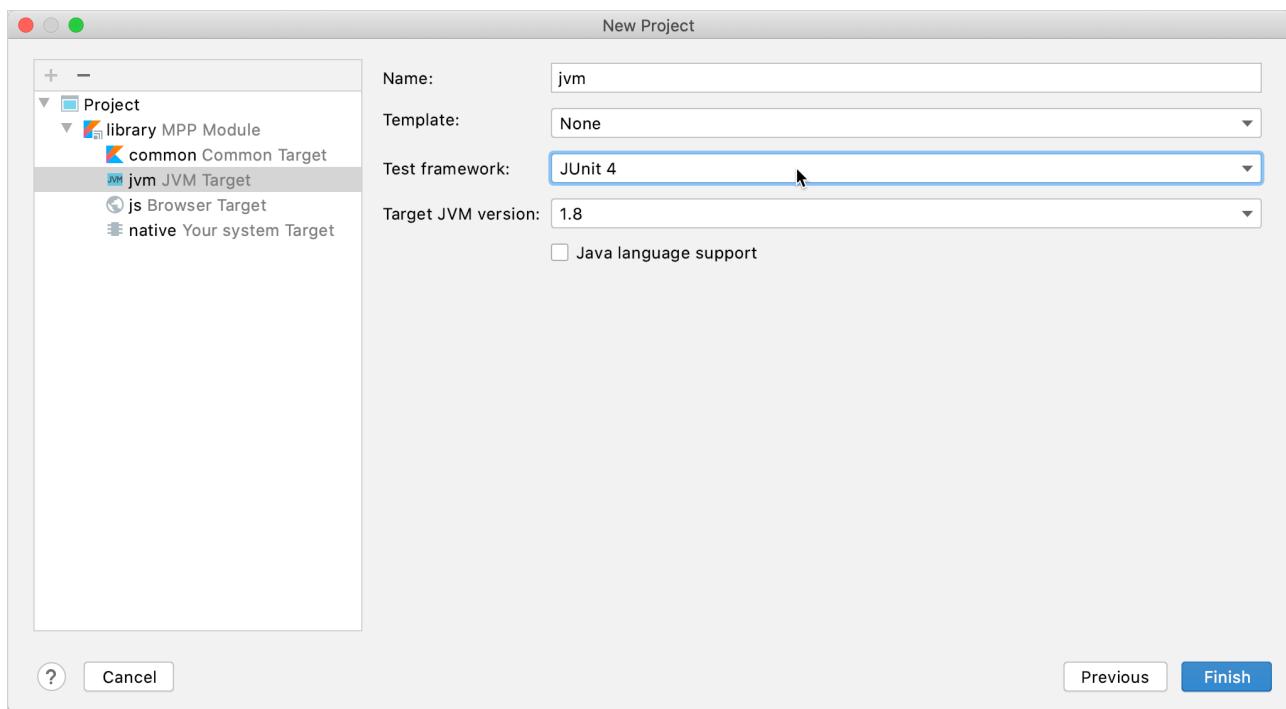
Select a project template

4. Select the Gradle DSL – Kotlin or Groovy.

5. Click Next.

You can finish creating the project by clicking Finish on the next screen or configure it if necessary:

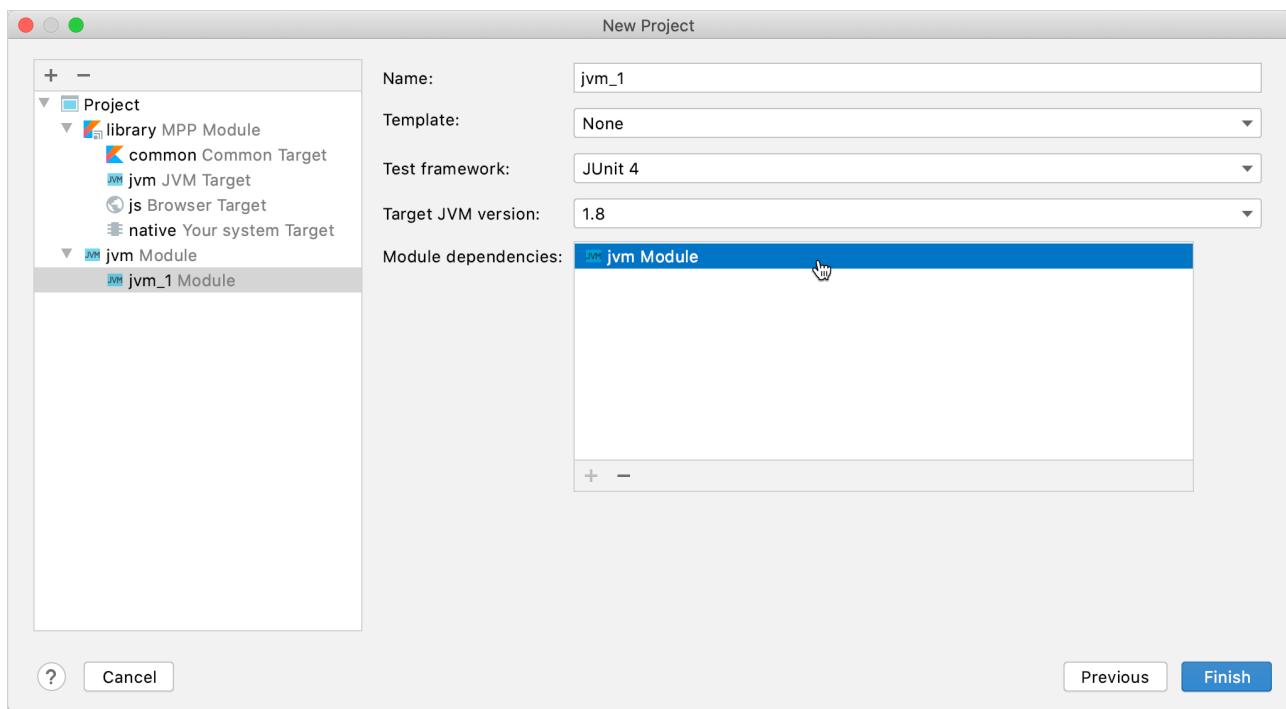
6. Add the target platforms and modules by clicking the + icon.
7. Configure target settings, such as the target template, JVM target version, and test framework.



Configure the project

8. If necessary, specify dependencies between modules:

- Multiplatform and Android modules
- Multiplatform and iOS modules
- JVM modules



Add module dependencies

9. Click Finish.

The new project opens.

What's next?

- [Understand the multiplatform project structure.](#)
- [Create and publish a multiplatform library – tutorial.](#)
- [Create your first KMM application for Android and iOS – tutorial.](#)
- [Create a full-stack web app with Kotlin Multiplatform – hands-on tutorial.](#)

Discover your project

Discover main parts of your multiplatform project:

- [Multiplatform plugin](#)
- [Targets](#)
- [Source sets](#)

- [Compilations](#)

Multiplatform plugin

When you [create a multiplatform project](#), the Project Wizard automatically applies the kotlin-multiplatform Gradle plugin in the file build.gradle (.kts).

You can also apply it manually.

The kotlin-multiplatform plugin works with Gradle 6.1 or later.

Groovy

```
plugins {  
    id 'org.jetbrains.kotlin.multiplatform' version '1.5.20'  
}
```

Kotlin

```
plugins {  
    kotlin("multiplatform") version "1.5.20"  
}
```

The kotlin-multiplatform plugin configures the project for creating an application or library to work on multiple platforms and prepares it for building on these platforms.

In the file build.gradle (.kts), it creates the kotlin extension at the top level, which includes configuration for [targets](#), [source sets](#), and dependencies.

Targets

A multiplatform project is aimed at multiple platforms that are represented by different targets. A target is part of the build that is responsible for building, testing, and packaging the application for a specific platform, such as macOS, iOS, or Android. See the list of [supported platforms](#).

When you create a multiplatform project, targets are added to the kotlin block in the file build.gradle (build.gradle.kts).

```
kotlin {  
    jvm()  
    js {  
        browser {}  
    }  
}
```

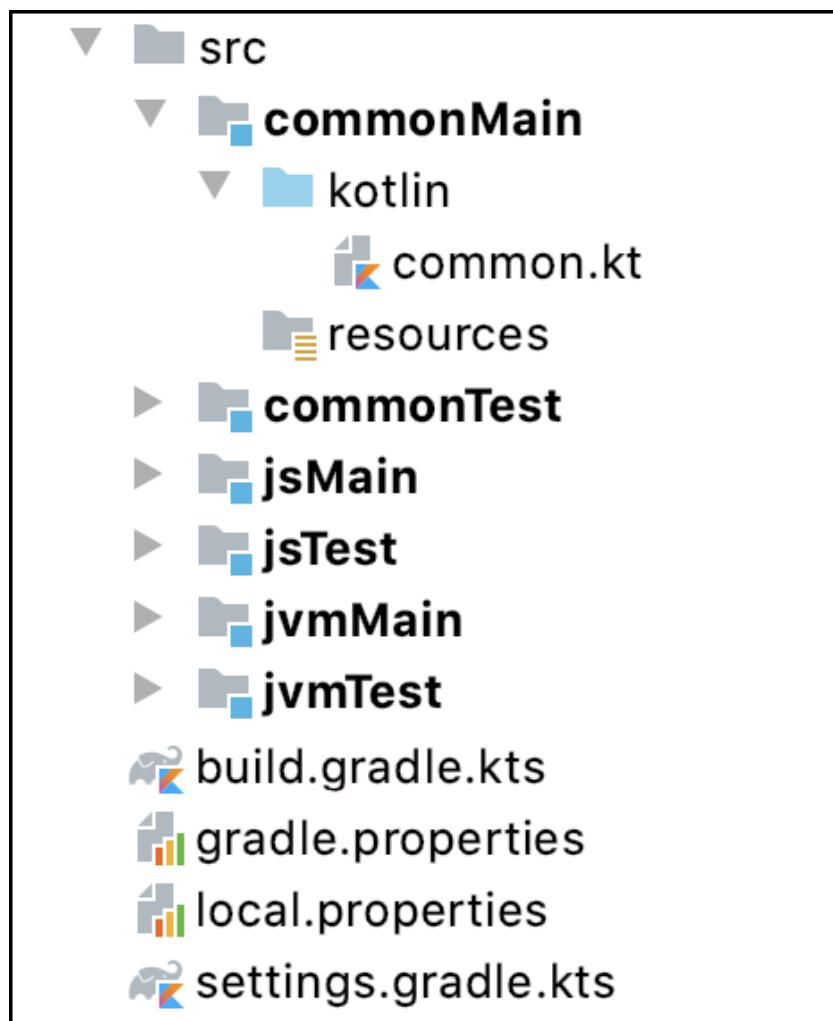
```
}
```

Learn how to [set up targets manually](#).

Source sets

The project includes the directory `src` with Kotlin source sets, which are collections of Kotlin code files, along with their resources, dependencies, and language settings. A source set can be used in Kotlin compilations for one or more target platforms.

Each source set directory includes Kotlin code files (the `kotlin` directory) and resources. The Project Wizard creates default source sets for the main and test compilations of the common code and all added targets.



Source sets

Source set names are case sensitive.

Source sets are added to the sourceSets block of the top-level kotlin block.

Groovy

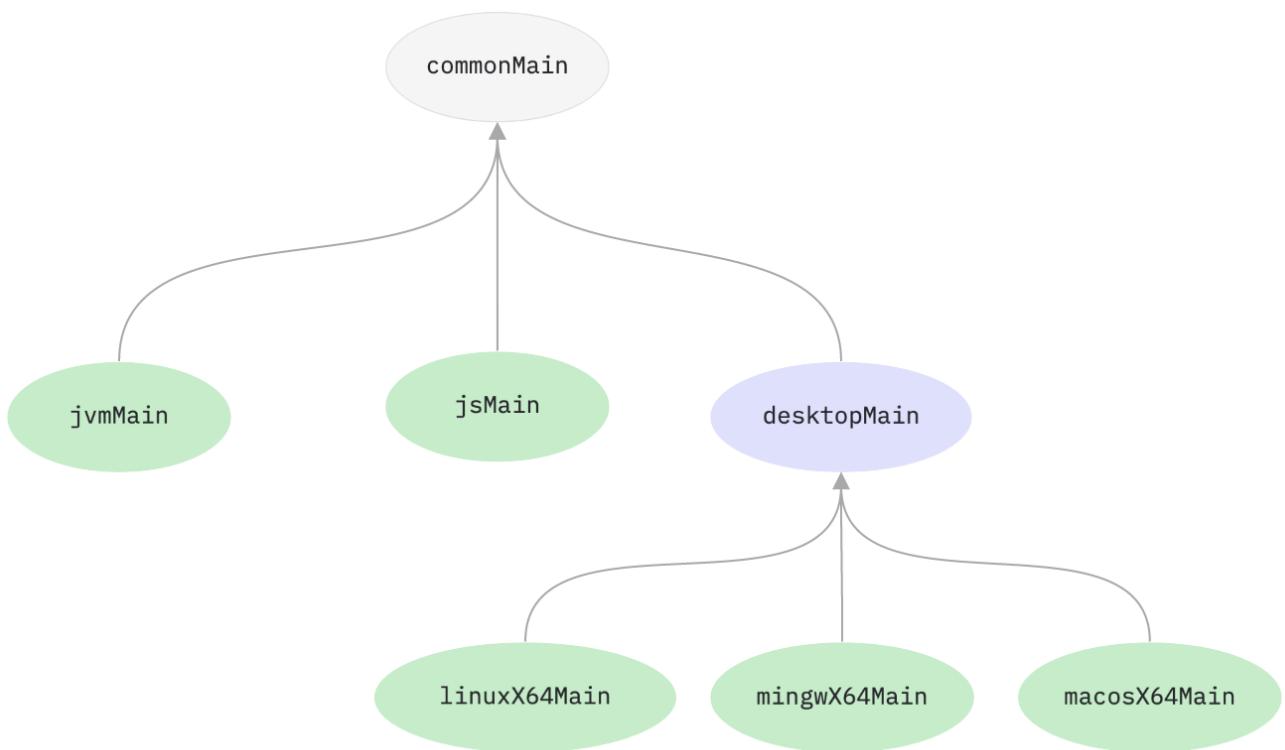
```
kotlin {  
    sourceSets {  
        commonMain { /* ... */}  
        commonTest { /* ... */}  
        jvmMain { /* ... */}  
        jvmTest { /* ... */}  
        jsMain { /* ... */}  
        jsTest { /* ... */}  
    }  
}
```

Kotlin

```
kotlin {  
    sourceSets {  
        val commonMain by getting { /* ... */}  
        val commonTest by getting { /* ... */}  
        val jvmMain by getting { /* ... */}  
        val jvmTest by getting { /* ... */}  
        val jsMain by getting { /* ... */}  
        val jsTest by getting { /* ... */}  
    }  
}
```

Source sets form a hierarchy, which is used for sharing the common code. In a source set shared among several targets, you can use the platform-specific language features and dependencies that are available for all these targets.

For example, all Kotlin/Native features are available in the desktopMain source set, which targets the Linux (linuxX64), Windows (mingwX64), and macOS (macosX64) platforms.



Hierarchical structure

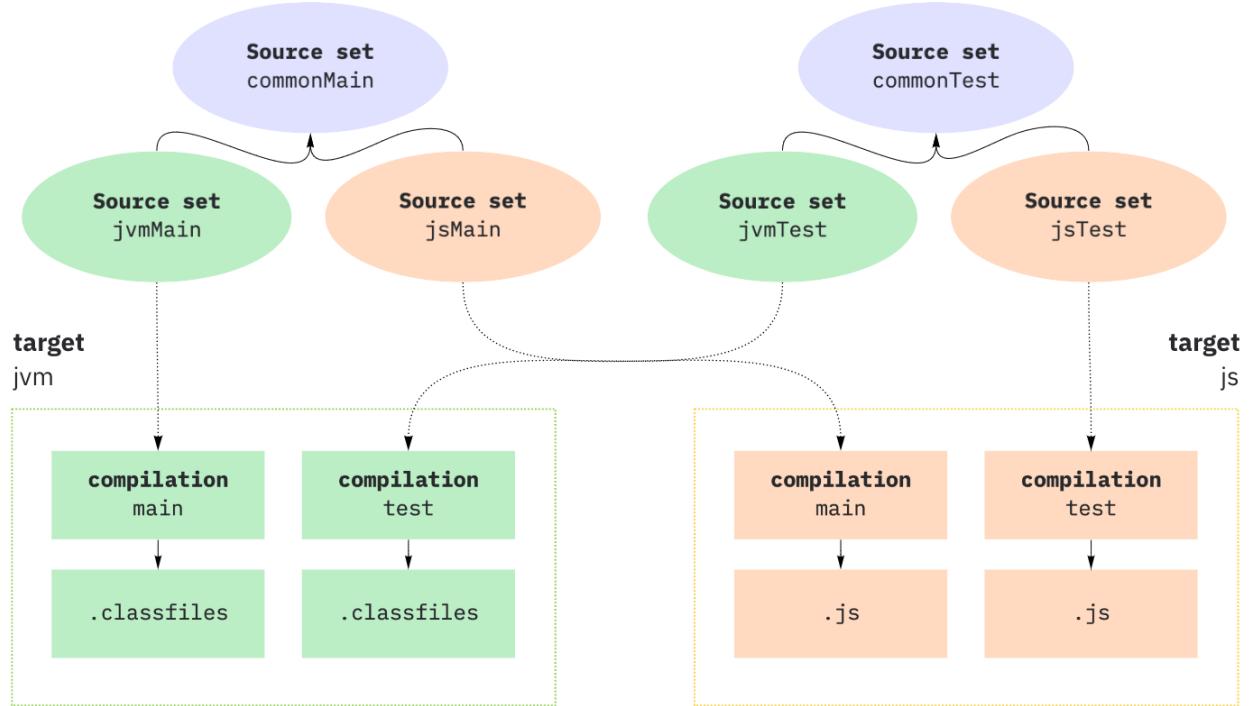
Learn how to [build the hierarchy of source sets](#).

Compilations

Each target can have one or more compilations, for example, for production and test purposes.

For each target, default compilations include:

- main and test compilations for JVM, JS, and Native targets.
- A compilation per [Android build variant](#), for Android targets.



Compilations

Each compilation has a default source set, which contains sources and dependencies specific to that compilation.

Learn how to [configure compilations](#).

Share code on platforms

With Kotlin Multiplatform, you can share the code using the mechanisms Kotlin provides:

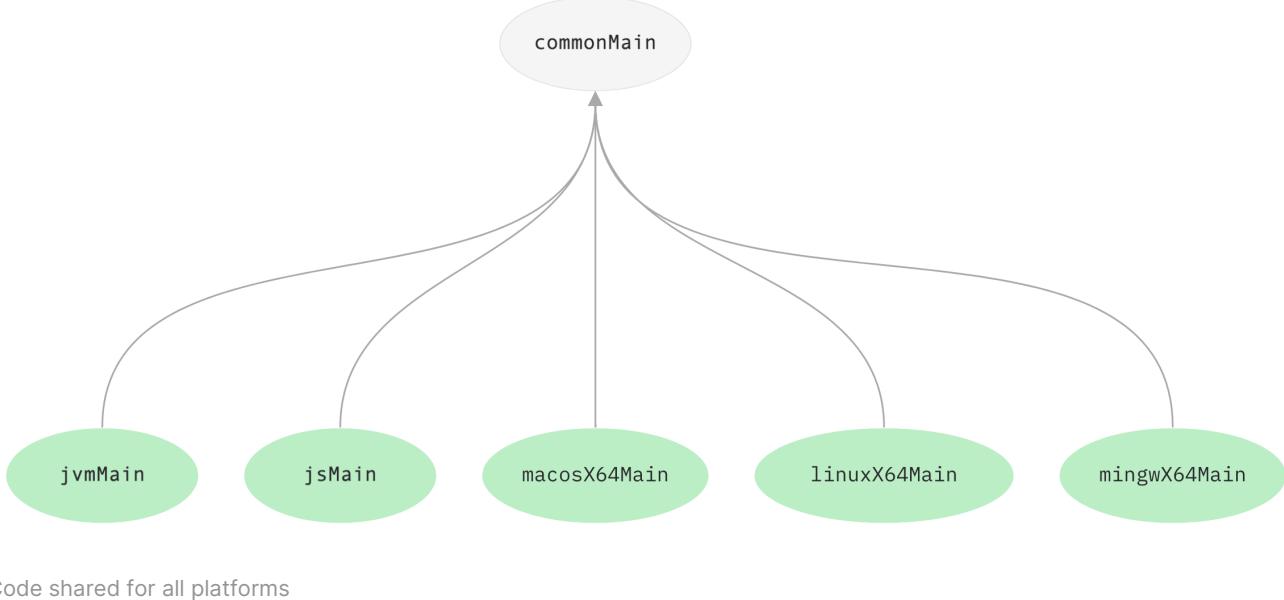
- [Share code among all platforms used in your project](#). Use it for sharing the common business logic that applies to all platforms.
- [Share code among some platforms included in your project but not all](#). You can reuse much of the code in similar platforms using a hierarchical structure. You can use [target shortcuts](#) for common combinations of targets or [create the hierarchical structure manually](#).

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

Share code on all platforms

If you have business logic that is common for all platforms, you don't need to write the same code for each platform –

just share it in the common source set.



All platform-specific source sets depend on the common source set by default. You don't need to specify any dependsOn relations manually for default source sets, such as jvmMain, macosX64Main, and others.

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

Share code on similar platforms

You often need to create several native targets 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 is for iOS ARM64 devices, the other is 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 the same. So iOS-specific code could be shared between them.

Evidently, 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.

In this case, you can share code across native targets in your project using the hierarchical structure.

To enable the hierarchy structure support, add the following flag to your gradle.properties.

```
kotlin.mpp.enableGranularSourceSetsMetadata=true
```

There are two ways you can create the hierarchical structure:

- [Use target shortcuts](#) to easily create the hierarchy structure for common combinations of native targets.
- [Configure the hierarchical structure manually](#).

Learn more about [sharing code in libraries](#) and [using Native libraries in the hierarchical structure](#).

Due to a [known issue](#), you won't be able to use IDE features, such as code completion and highlighting, for the shared native source set in a multiplatform project with hierarchical structure support if your project depends on:

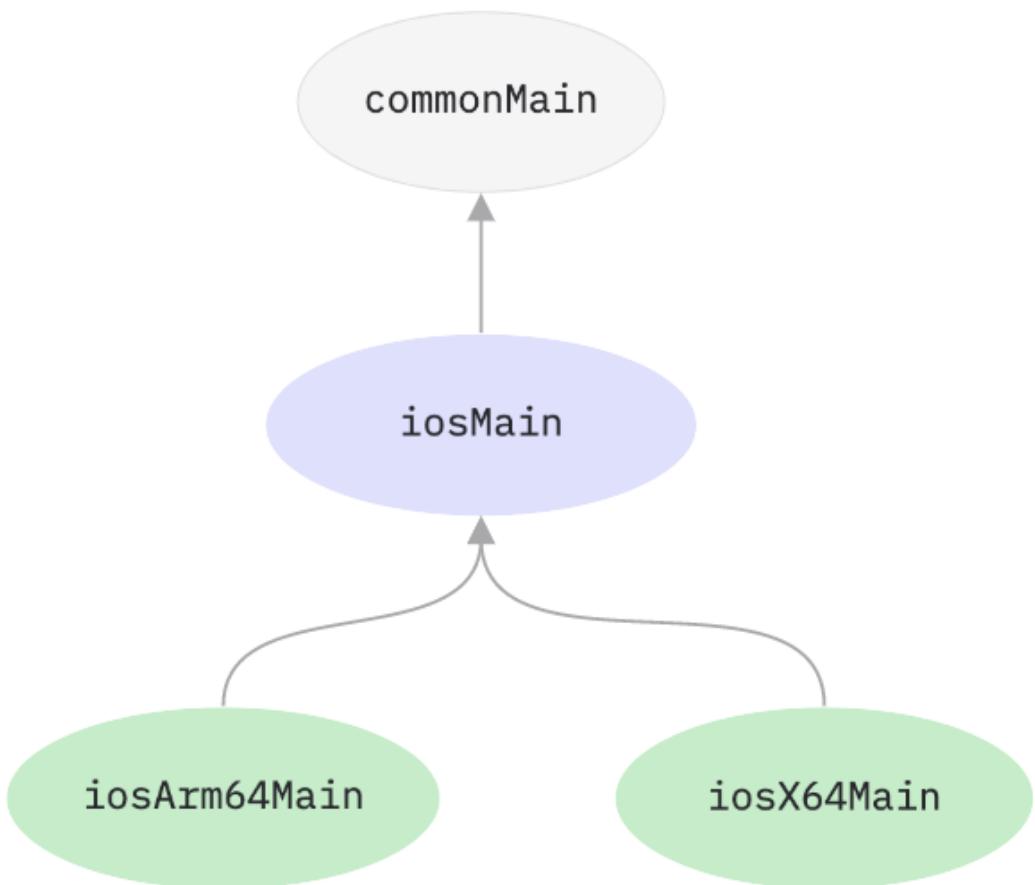
- Multiplatform libraries that don't support the hierarchical structure.
- Third-party native libraries, with the exception of [platform libraries](#) supported out of the box.

This issue applies only to the shared native source set. The IDE will correctly support the rest of the code.

Learn how to [work around this issue](#) for similar source sets, such as iosArm64 and iosX64.

Use target shortcuts

In a typical multiplatform project with two iOS-related targets – iosArm64 and iosX64, the hierarchical structure includes an intermediate source set (iosMain), which is used by the platform-specific source sets.



Code shared for iOS targets

The kotlin-multiplatform plugin provides target shortcuts for creating structures for common combinations of targets.

Target shortcut Targets

ios iosArm64, iosX64

watchos watchosArm32, watchosArm64, watchosX64

tvos tvosArm64, tvosX64

All shortcuts create similar hierarchical structures in the code. For example, the ios shortcut creates the following hierarchical structure:

Groovy

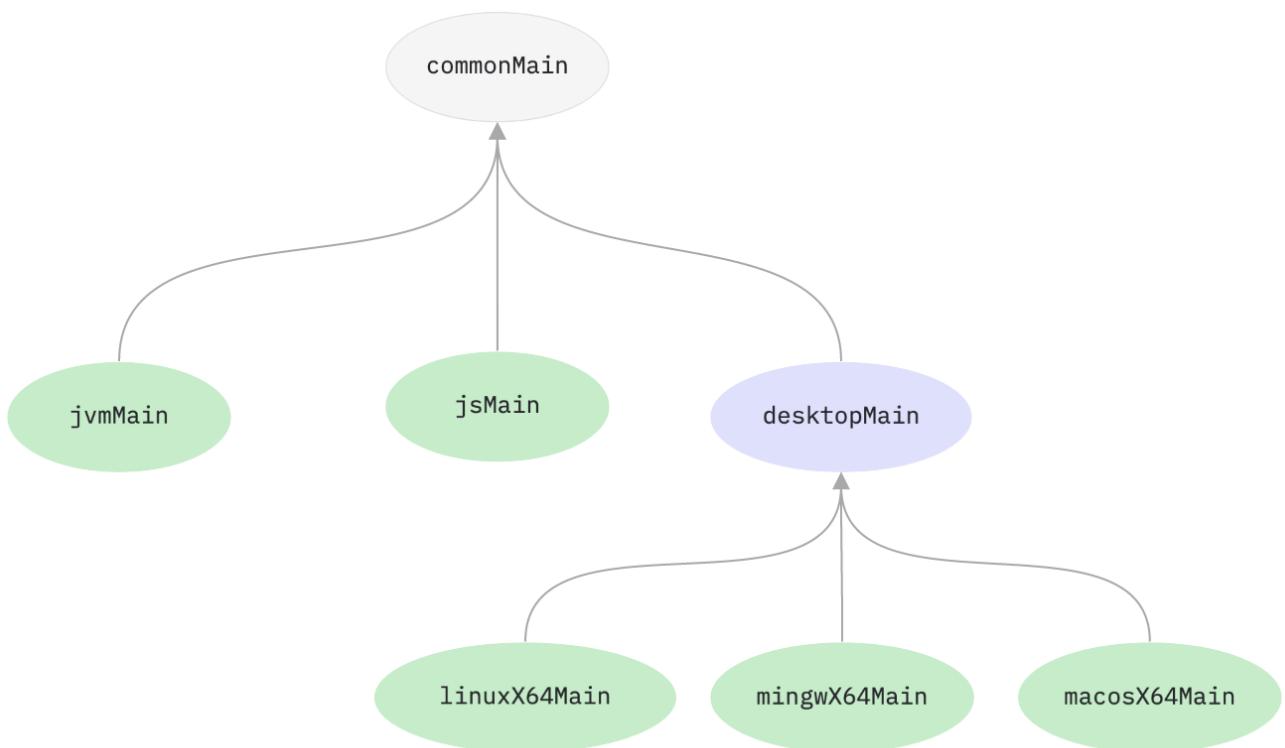
```
kotlin {  
    sourceSets{  
        iosMain {  
            dependsOn(commonMain)  
            iosX64Main.dependsOn(it)  
            iosArm64Main.dependsOn(it)  
        }  
    }  
}
```

Kotlin

```
kotlin {  
    sourceSets{  
        val commonMain by sourceSets.getting  
        val iosX64Main by sourceSets.getting  
        val iosArm64Main by sourceSets.getting  
        val iosMain by sourceSets.creating {  
            dependsOn(commonMain)  
            iosX64Main.dependsOn(this)  
            iosArm64Main.dependsOn(this)  
        }  
    }  
}
```

Configure the hierarchical structure manually

To create the hierarchical structure manually, introduce an intermediate source set that holds the shared code for several targets and create a structure of the source sets including the intermediate one.



Hierarchical structure

For example, if you want to share code among native Linux, Windows, and macOS targets – linuxX64M, mingwX64, and macosX64:

1. Add the intermediate source set desktopMain that holds the shared logic for these targets.
2. Specify the hierarchy of source sets using the dependsOn relation.

Groovy

```

kotlin {
    sourceSets {
        desktopMain {
            dependsOn(commonMain)
        }
        linuxX64Main {
            dependsOn(desktopMain)
        }
        mingwX64Main {
            dependsOn(desktopMain)
        }
        macosX64Main {
            dependsOn(desktopMain)
        }
    }
}

```

```

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)
        }
    }
}

```

You can have a shared source set for the following combinations of targets:

- JVM + JS + Native
- JVM + Native
- JS + Native
- JVM + JS
- Native

Kotlin doesn't currently support sharing a source set for these combinations:

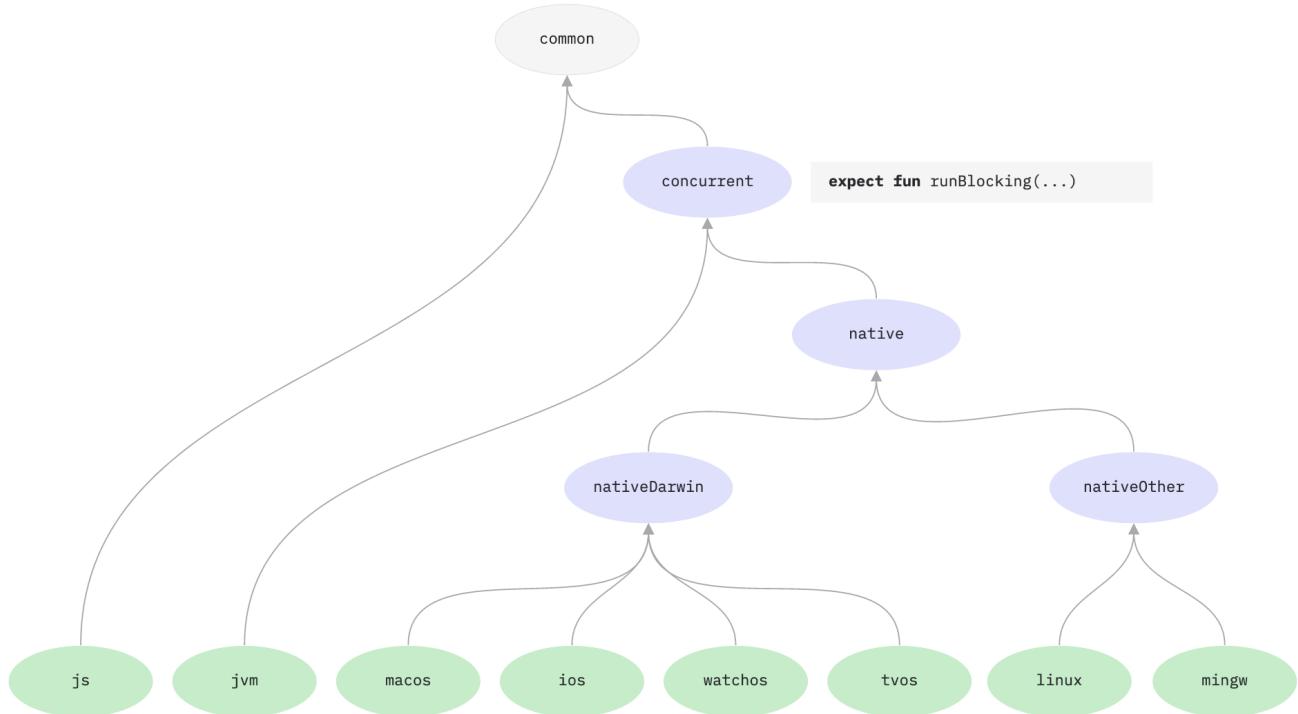
- Several JVM targets
- JVM + Android targets
- Several JS targets

If you need to access platform-specific APIs from a shared native source set, IntelliJ IDEA will help you detect common declarations that you can use in the shared native code. For other cases, use the Kotlin mechanism of [expected and actual declarations](#).

Share code in libraries

Thanks to the hierarchical project structure, libraries can also provide common APIs for a subset of targets. When a [library is published](#), the API of its intermediate source sets is embedded into the library artifacts along with information about the project structure. When you use this library, the intermediate source sets of your project access only those APIs of the library which are available to the targets of each source set.

For example, check out the following source set hierarchy from the `kotlinx.coroutines` repository:



Library hierarchical structure

The concurrent source set declares the function `runBlocking` and is compiled for the JVM and the native targets. Once the `kotlinx.coroutines` library is updated and published with the hierarchical project structure, you can depend on it and call `runBlocking` from a source set that is shared between the JVM and native targets since it matches the “targets signature” of the library’s concurrent source set.

Use native libraries in the hierarchical structure

You can use platform-dependent libraries like Foundation, UIKit, and POSIX in source sets shared among several native targets. This helps 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.

However, note that there are some limitations:

- This approach works only for a native source set that is shared among platform-specific source sets. It doesn’t work for native source sets shared at higher levels of the source set hierarchy.
For example, if you have `nativeDarwinMain` that is a parent of `watchosMain` and `iosMain`, where `iosMain` has two children – `iosArm64Main` and `iosX64Main`, you can use platform-dependent libraries only for `iosMain` but not for `nativeDarwinMain`.
- It works only for interop libraries shipped with Kotlin/Native.

To enable usage of platform-dependent libraries in shared source sets, add the following to your `gradle.properties`:

```
kotlin.mpp.enableGranularSourceSetsMetadata=true  
kotlin.native.enableDependencyPropagation=false
```

Learn more about the [technical details](#).

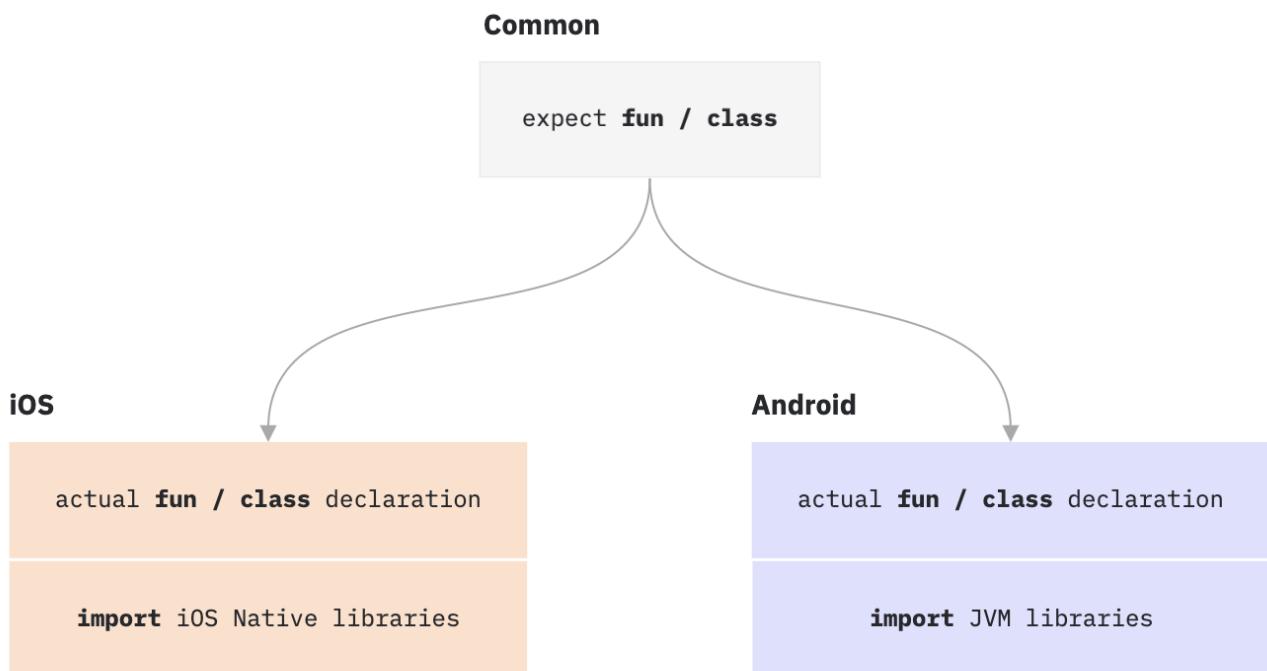
Connect to platform-specific APIs

The expect/ actual feature is in [Beta](#). It 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.

If you're developing a multiplatform application that needs to access platform-specific APIs that implement the required functionality, use the Kotlin mechanism of expected and actual declarations.

With this mechanism, a common source set defines an expected declaration, and platform source sets must provide the actual declaration that corresponds to the expected declaration. This works for most Kotlin declarations, such as functions, classes, interfaces, enumerations, properties, and annotations.



Expect and actual declarations

```
// Common  
expect fun randomUUID(): String
```

```
// Android
import java.util.*
actual fun randomUUID() = UUID.randomUUID().toString()
```

```
// iOS
import platform.Foundation.NSUUID
actual fun randomUUID(): String = NSUUID().UUIDString()
```

Here's another example of code sharing and interaction between the common and platform logic in a minimalistic logging framework.

```
// Common
enum class LogLevel {
    DEBUG, WARN, ERROR
}

internal expect fun writeLogMessage(message: String, logLevel: LogLevel)

fun logDebug(message: String) = writeLogMessage(message, LogLevel.DEBUG)
fun logWarn(message: String) = writeLogMessage(message, LogLevel.WARN)
fun logError(message: String) = writeLogMessage(message, LogLevel.ERROR)
```

```
// JVM
internal actual fun writeLogMessage(message: String, logLevel: LogLevel) {
    println("[${logLevel}]: $message")
}
```

For JavaScript, a completely different set of APIs is available, and the actual declaration will look like this.

```
// JS
internal actual fun writeLogMessage(message: String, logLevel: LogLevel) {
    when (logLevel) {
        LogLevel.DEBUG -> console.log(message)
        LogLevel.WARN -> console.warn(message)
        LogLevel.ERROR -> console.error(message)
    }
}
```

Rules for expected and actual declarations

The main rules regarding expected and actual declarations are:

- An expected declaration is marked with the `expect` keyword; the actual declaration is marked with the `actual` keyword.
- `expect` and `actual` declarations have the same name and are located in the same package (have the same fully qualified name).

- expect declarations never contain any implementation code.

During each platform compilation, the compiler ensures that every declaration marked with the expect keyword in the common or intermediate source set has the corresponding declarations marked with the actual keyword in all platform source sets. The IDE provides tools that help you create the missing actual declarations.

If you have a platform-specific library that you want to use in shared code while providing your own implementation for another platform, you can provide a typealias to an existing class as the actual declaration:

```
expect class AtomicRef<V>(value: V) {
    fun get(): V
    fun set(value: V)
    fun getAndSet(value: V): V
    fun compareAndSet(expect: V, update: V): Boolean
}
```

```
actual typealias AtomicRef<V> = java.util.concurrent.atomic.AtomicReference<V>
```

Use expected and actual declarations only for Kotlin declarations that have platform-specific dependencies. It is better to implement as much functionality as possible in the shared module even if doing so takes more time.

Don't overuse expected and actual declarations – in some cases, an [interface](#) may be a better choice because it is more flexible and easier to test.

Set up targets manually

You can add targets when [creating a project with the Project Wizard](#). If you need to add a target later, you can do this manually using target presets for [supported platforms](#).

Learn more about [additional settings for targets](#).

```
kotlin {
    jvm() // Create a JVM target with the default name 'jvm'

    linuxX64() {
        /* Specify additional settings for the 'linux' target here */
    }
}
```

Each target can have one or more [compilations](#). In addition to default compilations for test and production purposes, you can [create custom compilations](#).

Distinguish several targets for one platform

You can have several targets for one platform in a multiplatform library. For example, these targets can provide the same API but use different libraries during runtime, such as testing frameworks and logging solutions. Dependencies on such a multiplatform library may fail to resolve because it isn't clear which target to choose.

To solve this, mark the targets on both the library author and consumer sides with a custom attribute, which Gradle uses during dependency resolution.

For example, consider a testing library that supports both JUnit and TestNG in the two targets. The library author needs to add an attribute to both targets as follows:

Groovy

```
def testFrameworkAttribute = Attribute.of('com.example.testFramework', String)

kotlin {
    jvm('junit') {
        attributes.attribute(testFrameworkAttribute, 'junit')
    }
    jvm('testng') {
        attributes.attribute(testFrameworkAttribute, 'testng')
    }
}
```

Kotlin

```
val testFrameworkAttribute = Attribute.of("com.example.testFramework", String::class.java)

kotlin {
    jvm("junit") {
        attributes.attribute(testFrameworkAttribute, "junit")
    }
    jvm("testng") {
        attributes.attribute(testFrameworkAttribute, "testng")
    }
}
```

The consumer has to add the attribute to a single target where the ambiguity arises.

Add dependencies

To add a dependency on a library, set a dependency of the required `type` (for example, implementation) in the `dependencies` block in your [Gradle build script](#).

Groovy

```
kotlin {  
    sourceSets {  
        commonMain {  
            dependencies {  
                implementation 'com.example:my-library:1.0'  
            }  
        }  
    }  
}
```

Kotlin

```
kotlin {  
    sourceSets {  
        val commonMain by getting {  
            dependencies {  
                implementation("com.example:my-library:1.0")  
            }  
        }  
    }  
}
```

Alternatively, you can [set dependencies at the top level](#).

Dependency on the standard library

A dependency on a standard library (stdlib) in each source set is added automatically. The version of the standard library is the same as the version of the kotlin-multiplatform plugin.

For platform-specific source sets, the corresponding platform-specific variant of the library is used, while a common standard library is added to the rest. The Kotlin Gradle plugin 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](#).

Set dependencies on test libraries

The `kotlin.test` API is available for multiplatform tests. When you [create a multiplatform project](#), the Project Wizard automatically adds test dependencies to common and platform-specific source sets.

If you didn't use the Project Wizard to create your project, you can [add the dependencies manually](#).

Set a dependency on a kotlinc library

If you use a kotlinx library and need a platform-specific dependency, you can use platform-specific variants of libraries with suffixes such as -jvm or -js, for example, kotlinx-coroutines-core-jvm. You can also use the library base artifact name instead – kotlinx-coroutines-core.

Groovy

```
kotlin {  
    sourceSets {  
        jvmMain {  
            dependencies {  
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.5.0'  
            }  
        }  
    }  
}
```

Kotlin

```
kotlin {  
    sourceSets {  
        val jvmMain by getting {  
            dependencies {  
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.5.0")  
            }  
        }  
    }  
}
```

If you use a multiplatform library and need to depend on the shared code, set the dependency only once in the shared source set. Use the library base artifact name, such as kotlinx-coroutines-core or ktor-client-core.

Groovy

```
kotlin {  
    sourceSets {  
        commonMain {  
            dependencies {  
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.0'  
            }  
        }  
    }  
}
```

Kotlin

```
kotlin {  
    sourceSets {
```

```

    val commonMain by getting {
        dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.0")
        }
    }
}

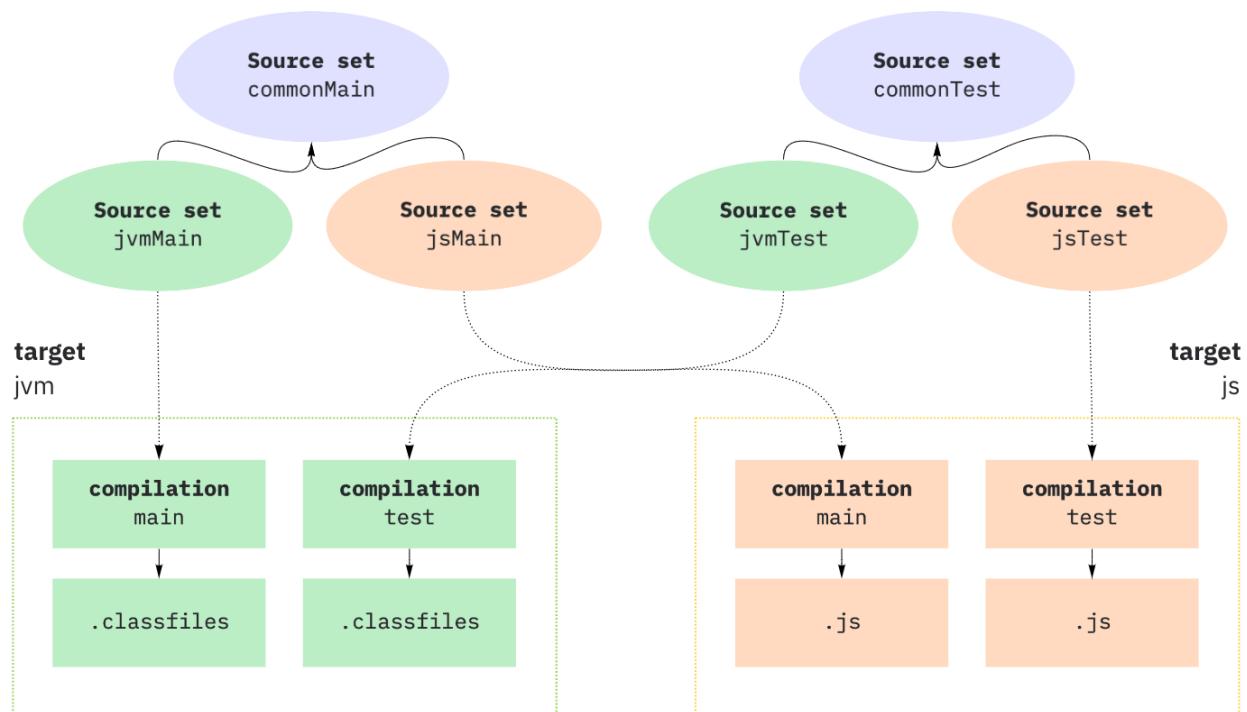
```

Configure compilations

Kotlin multiplatform projects use compilations for producing artifacts. Each target can have one or more compilations, for example, for production and test purposes.

For each target, default compilations include:

- main and test compilations for JVM, JS, and Native targets.
- A [compilation per Android build variant](#), for Android targets.



Compilations

If you need to compile something other than production code and unit tests, for example, integration or performance tests, you can [create a custom compilation](#).

You can configure how artifacts are produced in:

- All compilations in your project at once.
- Compilations for one target since one target can have multiple compilations.
- A specific compilation.

See the [list of compilation parameters](#) and [compiler options](#) available for all or specific targets.

Configure all compilations

```
kotlin {  
    targets.all {  
        compilations.all {  
            kotlinOptions {  
                allWarningsAsErrors = true  
            }  
        }  
    }  
}
```

Configure compilations for one target

Groovy

```
kotlin {  
    jvm().compilations.all {  
        kotlinOptions {  
            sourceMap = true  
            metaInfo = true  
        }  
    }  
}
```

Kotlin

```
kotlin {  
    targets.jvm.compilations.all {  
        kotlinOptions {  
            sourceMap = true  
            metaInfo = true  
        }  
    }  
}
```

Configure one compilation

Groovy

```
kotlin {  
    jvm().compilations.main {  
        kotlinOptions {  
            jvmTarget = "1.8"  
        }  
    }  
}
```

Kotlin

```
kotlin {  
    jvm {  
        val main by compilations.getting {  
            kotlinOptions {  
                jvmTarget = "1.8"  
            }  
        }  
    }  
}
```

Create a custom compilation

If you need to compile something other than production code and unit tests, for example, integration or performance tests, create a custom compilation.

For example, to create a custom compilation for integration tests of the jvm() target, add a new item to the compilations collection.

For custom compilations, you need to set up all dependencies manually. The default source set of a custom compilation does not depend on the commonMain and the commonTest source sets.

Groovy

```
kotlin {  
    jvm() {  
        compilations.create('integrationTest') {  
            defaultSourceSet {  
                dependencies {  
                    def main = compilations.main  
                    // Compile against the main compilation's compile classpath and outputs:  
                    implementation(main.compileDependencyFiles + main.output.classesDirs)  
                    implementation kotlin('test-junit')  
                }  
            }  
        }  
    }  
}
```

```
        /* ... */
    }
}

// Create a test task to run the tests produced by this compilation:
tasks.register('jvmIntegrationTest', Test) {
    // Run the tests with the classpath containing the compile dependencies
(including 'main'),
    // runtime dependencies, and the outputs of this compilation:
    classpath = compileDependencyFiles + runtimeDependencyFiles +
output.allOutputs

    // Run only the tests from this compilation's outputs:
    testClassesDirs = output.classesDirs
}
}
```

Kotlin

```
kotlin {
    jvm() {
        compilations {
            val main by getting

                val integrationTest by compilations.creating {
                    defaultSourceSet {
                        dependencies {
                            // Compile against the main compilation's compile classpath and
                            outputs:
                                implementation(main.compileDependencyFiles + main.output.classesDirs)
                                implementation(kotlin("test-junit"))
                                /* ... */
                            }
                        }

                        // Create a test task to run the tests produced by this compilation:
                        tasks.register<Test>("integrationTest") {
                            // Run the tests with the classpath containing the compile dependencies
                            (including 'main'),
                            // runtime dependencies, and the outputs of this compilation:
                            classpath = compileDependencyFiles + runtimeDependencyFiles +
                            output.allOutputs

                            // Run only the tests from this compilation's outputs:
                            testClassesDirs = output.classesDirs
                        }
                    }
                }
            }
        }
    }
}
```

You also need to create a custom compilation in other cases, for example, if you want to combine compilations for different JVM versions in your final artifact, or you have already set up source sets in Gradle and want to migrate to a

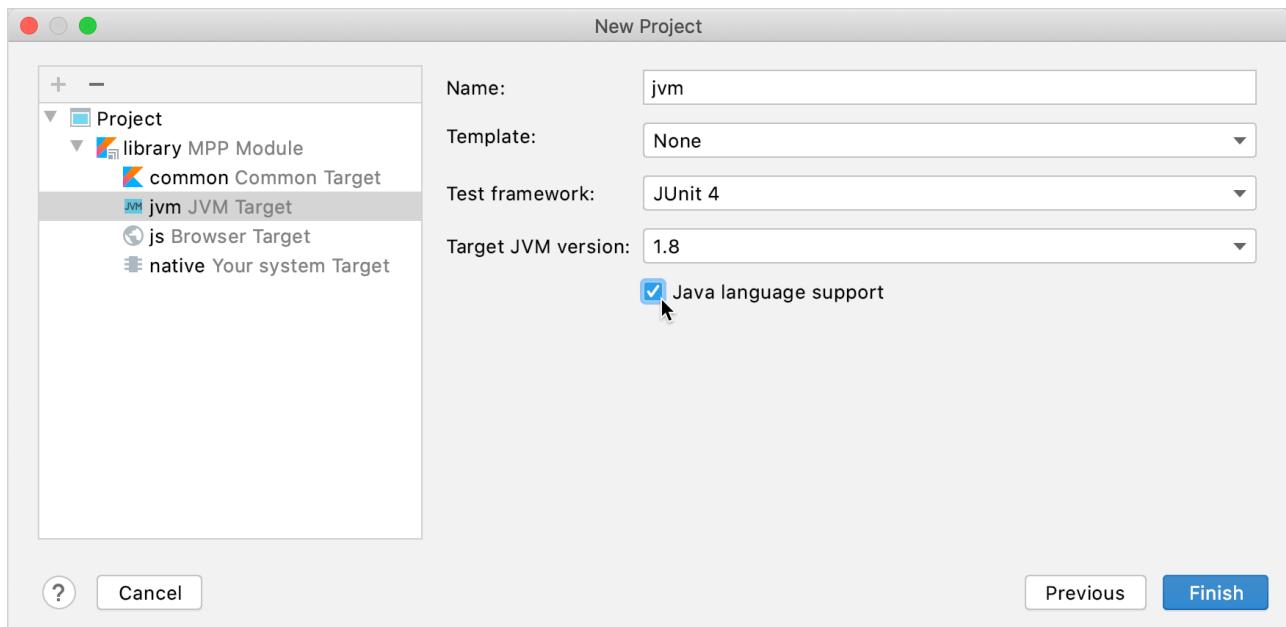
multiplatform project.

Include Java sources in JVM compilations

By default, the JVM target ignores Java sources and compiles only Kotlin source files.

To include Java sources in the compilations of the JVM target, explicitly enable the Java language support for the target:

- When [creating a project with the Project Wizard](#).



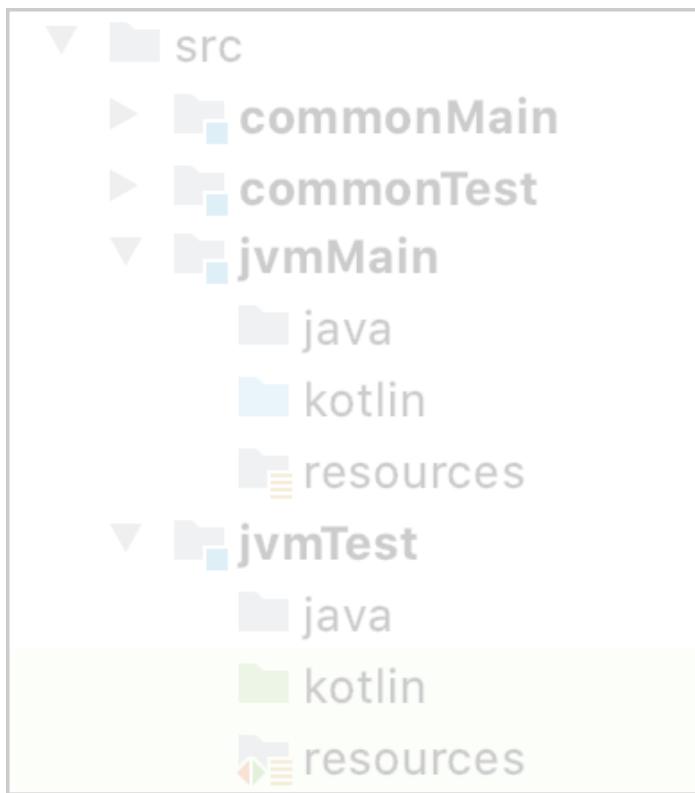
Enable Java language support

- In the build script of an existing project.

```
kotlin {  
    jvm {  
        withJava()  
    }  
}
```

This applies the Gradle java plugin and configures the target to cooperate with it.

The Java source files are placed in the child directories of the Kotlin source roots. For example, the paths are:



Java source files

The common source sets cannot include Java sources.

Due to current limitations, the Kotlin plugin replaces some tasks configured by the Java plugin:

- The target's JAR task instead of jar (for example, jvmJar).
- The target's test task instead of test (for example, jvmTest).
- The resources are processed by the equivalent tasks of the compilations instead of *ProcessResources tasks.

The publication of this target is handled by the Kotlin plugin and doesn't require steps that are specific for the Java plugin.

Configure interop with native languages

Kotlin provides [interoperability with native languages](#) and DSL to configure this for a specific compilation.

Native language	Supported platforms	Comments
C	All platforms, except for WebAssembly	

Native language	Supported platforms	Comments
Objective-C	Apple platforms (macOS, iOS, watchOS, tvOS)	
Swift via Objective-C	Apple platforms (macOS, iOS, watchOS, tvOS)	Kotlin can use only Swift declarations marked with the <code>@objc</code> attribute.

A compilation can interact with several native libraries. Configure interoperability in the `cinterop` block of the compilation with [available parameters](#).

Groovy

```
kotlin {
    linuxX64 { // Replace with a target you need.
        compilations.main {
            cinterop {
                myInterop {
                    // Def-file describing the native API.
                    // The default path is src/nativeInterop/cinterop/<interop-name>.def
                    defFile project.file("def-file.def")

                    // Package to place the Kotlin API generated.
                    packageName 'org.sample'

                    // Options to be passed to compiler by cinterop tool.
                    compilerOpts '-Ipath/to/headers'

                    // Directories for header search (an equivalent of the -I<path> compiler
                    option).
                    includeDirs.allHeaders("path1", "path2")

                    // Additional directories to search headers listed in the 'headerFilter'
                    def-file option.
                    // -headerFilterAdditionalSearchPrefix command line option equivalent.
                    includeDirs.headerFilterOnly("path1", "path2")

                    // A shortcut for includeDirs.allHeaders.
                    includeDirs("include/directory", "another/directory")
                }
            }
        }
    }
}
```

Kotlin

```
kotlin {
    linuxX64 { // Replace with a target you need.
        compilations.getByName("main") {
```

```

    val myInterop by cinterop.creating {
        // Def-file describing the native API.
        // The default path is src/nativeInterop/cinterop/<interop-name>.def
        defFile(project.file("def-file.def"))

        // Package to place the Kotlin API generated.
        packageName("org.sample")

        // Options to be passed to compiler by cinterop tool.
        compilerOpts("-Ipath/to/headers")

        // Directories to look for headers.
        includeDirs.apply {
            // Directories for header search (an equivalent of the -I<path> compiler
            option).
            allHeaders("path1", "path2")

            // Additional directories to search headers listed in the 'headerFilter'
            def-file option.
            // -headerFilterAdditionalSearchPrefix command line option equivalent.
            headerFilterOnly("path1", "path2")
        }
        // A shortcut for includeDirs.allHeaders.
        includeDirs("include/directory", "another/directory")
    }

    val anotherInterop by cinterop.creating { /* ... */ }
}
}
}

```

Compilation for Android

The compilations created for an Android target by default are tied to [Android build variants](#): for each build variant, a Kotlin compilation is created under the same name.

Then, for each [Android source set](#) compiled for each of the variants, a Kotlin source set is created under that source set name prepended by the target name, like the Kotlin source set androidDebug for an Android source set debug and the Kotlin target named android. These Kotlin source sets are added to the variants' compilations accordingly.

The default source set commonMain is added to each production (application or library) variant's compilation. The commonTest source set is similarly added to the compilations of unit test and instrumented test variants.

Annotation processing with [kapt](#) is also supported, but due to current limitations it requires that the Android target is created before the kapt dependencies are configured, which needs to be done in a top-level dependencies block rather than within Kotlin source set dependencies.

```

kotlin {
    android { /* ... */ }
}

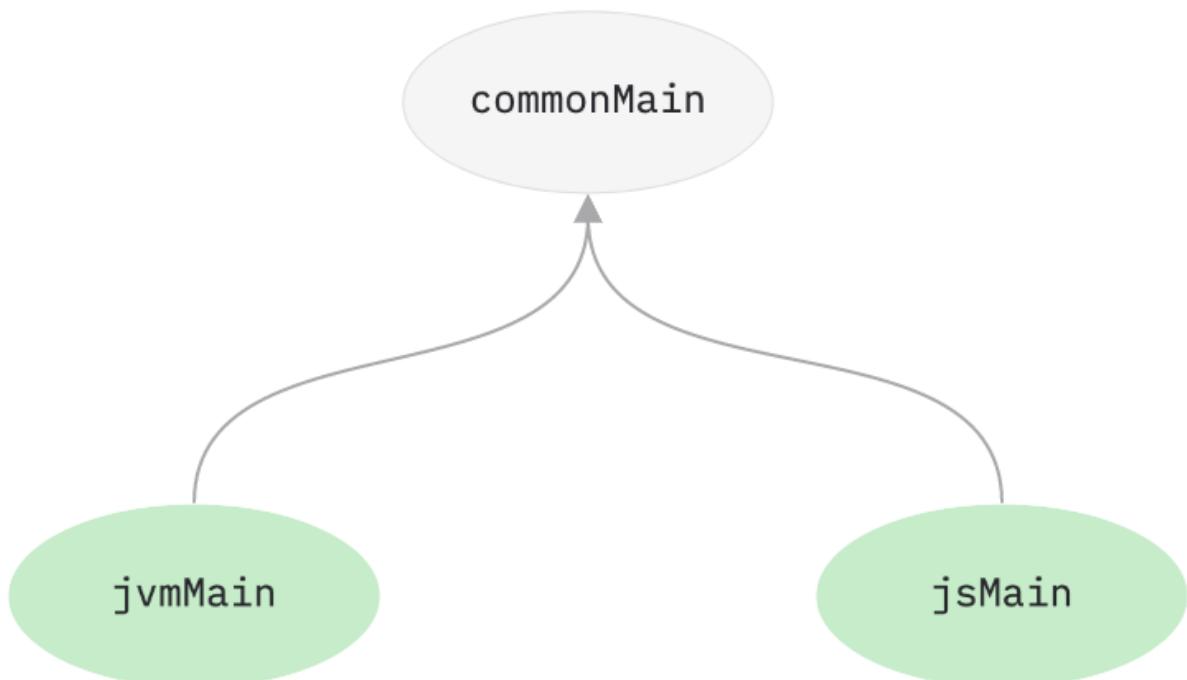
dependencies {

```

```
kapt("com.my.annotation:processor:1.0.0")
}
```

Compilation of the source set hierarchy

Kotlin can build a source set hierarchy with the dependsOn relation.



Source set hierarchy

If the source set jvmMain depends on a source set commonMain then:

- Whenever jvmMain is compiled for a certain target, commonMain takes part in that compilation as well and is also compiled into the same target binary form, such as JVM class files.
- Sources of jvmMain 'see' the declarations of commonMain, including internal declarations, and also see the dependencies of commonMain, even those specified as implementation dependencies.
- jvmMain can contain platform-specific implementations for the expected declarations of commonMain.
- The resources of commonMain are always processed and copied along with the resources of jvmMain.
- The language settings of jvmMain and commonMain should be consistent.

Language settings are checked for consistency in the following ways:

- jvmMain should set a languageVersion that is greater than or equal to that of commonMain.
- jvmMain should enable all unstable language features that commonMain enables (there's no such requirement for bugfix features).
- jvmMain should use all experimental annotations that commonMain uses.
- apiVersion, bugfix language features, and progressiveMode can be set arbitrarily.

Run tests

By default, Kotlin supports running tests for JVM, JS, Android, Linux, Windows, macOS as well as iOS, watchOS, and tvOS simulators. To run tests for other Kotlin/Native targets, you need to configure them manually in an appropriate environment, emulator, or test framework.

Required dependencies

The `kotlin.test` API is available for multiplatform tests. When you [create a multiplatform project](#), the Project Wizard automatically adds test dependencies to common and platform-specific source sets.

If you didn't use the Project Wizard to create your project, you can [add the dependencies manually](#).

Run tests for one or more targets

To run tests for all targets, run the check task.

To run tests for a particular target suitable for testing, run a test task `<targetName>Test`.

Test shared code

For testing shared code, you can use [actual declarations](#) in your tests.

For example, to test the shared code in commonMain:

```
expect object Platform {
    val name: String
}

fun hello(): String = "Hello from ${Platform.name}"

class Proxy {
    fun proxyHello() = hello()
}
```

You can use the following test in commonTest:

```
import kotlin.test.Test
import kotlin.test.assertTrue

class SampleTests {
    @Test
    fun testProxy() {
        assertTrue(Proxy().proxyHello().isNotEmpty())
    }
}
```

And the following test in iosTest:

```
import kotlin.test.Test
import kotlin.test.assertTrue

class SampleTestsIOS {
    @Test
    fun testHello() {
        assertTrue("iOS" in hello())
    }
}
```

You can also learn how to create and run multiplatform tests in the [Create and publish a multiplatform library – tutorial](#).

Publish a multiplatform library

You can publish a multiplatform library to a Maven repository with the [maven-publish Gradle plugin](#). Specify the group, version, and the [repositories](#) where the library should be published. The plugin creates publications automatically.

```
plugins {
    //...
    id("maven-publish")
}

group = "com.example"
version = "1.0"

publishing {
    repositories {
        maven {
            //...
        }
    }
}
```

Complete the [tutorial on creating and publishing a multiplatform library](#) to get hands-on experience.

Structure of publications

When used with maven-publish, the Kotlin plugin automatically creates publications for each target that can be built on the current host, except for the Android target, which needs an [additional step to configure publishing](#).

Publications of a multiplatform library include an additional root publication kotlinMultiplatform that 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 [adding dependencies](#).

This kotlinMultiplatform publication includes metadata artifacts and references the other publications as its variants.

Some repositories, such as Maven Central, require that the root module contains a JAR artifact without a classifier, for example kotlinMultiplatform-1.0.jar.

The Kotlin Multiplatform plugin automatically produces the required artifact with the embedded metadata artifacts.

This means you don't have to customize your build by adding an empty artifact to the root module of your library to meet the repository's requirements.

The kotlinMultiplatform publication may also need the sources and documentation artifacts if that is required by the repository. In that case, add those artifacts by using [artifact\(...\)](#) in the publication's scope.

Avoid duplicate publications

To avoid duplicate publications of modules that can be built on several platforms (like JVM and JS), configure the publishing tasks for these modules to run conditionally.

You can detect the platform in the script, introduce a flag such as isMainHost and set it to true for the main target platform. Alternatively, you can pass the flag from an external source, for example, from CI configuration.

This simplified example ensures that publications are only uploaded when isMainHost=true is passed. This means that a publication that can be published from multiple platforms will be published only once – from the main host.

Groovy

```
kotlin {
    jvm()
    js()
    mingwX64()
    linuxX64()
    def publicationsFromMainHost =
        [jvm(), js()].collect { it.name } + "kotlinMultiplatform"
    publishing {
        publications {
            matching { it.name in publicationsFromMainHost }.all { targetPublication ->
                tasks.withType(AbstractPublishToMaven)
```

```
        .matching { it.publication == targetPublication }
        .configureEach { onlyIf { findProperty("isMainHost") == "true" } }
    }
}
}
```

Kotlin

```
kotlin {
    jvm()
    js()
    mingwX64()
    linuxX64()
    val publicationsFromMainHost =
        listOf(jvm(), js()).map { it.name } + "kotlinMultiplatform"
    publishing {
        publications {
            matching { it.name in publicationsFromMainHost }.all {
                val targetPublication = this@all
                tasks.withType<AbstractPublishToMaven>()
                    .matching { it.publication == targetPublication }
                    .configureEach { onlyIf { findProperty("isMainHost") == "true" } }
            }
        }
    }
}
```

By default, each publication includes a sources JAR that contains the sources used by the main compilation of the target.

Publish an Android library

To publish an Android library, you need to provide additional configuration.

By default, no artifacts of an Android library are published. To publish artifacts produced by a set of [Android variants](#), specify the variant names in the Android target block:

```
kotlin {
    android {
        publishLibraryVariants("release", "debug")
    }
}
```

The example works for Android libraries without [product flavors](#). For a library with product flavors, the variant names also contain the flavors, like fooBarDebug or fooBazRelease.

If a library consumer defines variants that are missing in the library, they need to provide matching fallbacks. For example, if a library does not have or does not publish a staging build type, the library consumer must provide a fallback for the consumers who have such a build type, specifying at least one of the build types that the library publishes:

Groovy

```
android {  
    buildTypes {  
        staging {  
            // ...  
            matchingFallbacks = ['release', 'debug']  
        }  
    }  
}
```

Kotlin

```
android {  
    buildTypes {  
        val staging by creating {  
            // ...  
            matchingFallbacks = listOf("release", "debug")  
        }  
    }  
}
```

Similarly, a library consumer needs to provide matching fallbacks for custom product flavors if some are missing in the library publications.

You can also publish variants grouped by the product flavor, so that the outputs of the different build types are placed in a single module, with the build type becoming a classifier for the artifacts (the release build type is still published with no classifier). This mode is disabled by default and can be enabled as follows:

```
kotlin {  
    android {  
        publishLibraryVariantsGroupedByFlavor = true  
    }  
}
```

It is not recommended that you publish variants grouped by the product flavor in case they have different dependencies, as those will be merged into one dependencies list.

Create and publish a multiplatform library – tutorial

In this tutorial, you will learn how to create a multiplatform library for JVM, JS, and Native platforms, write common tests for all platforms, and publish the library to a local Maven repository.

This library converts raw data – strings and byte arrays – to the [Base64](#) format. It can be used on Kotlin/JVM, Kotlin/JS, and any available Kotlin/Native platform.

You will use different ways to implement the conversion to the Base64 format on different platforms:

- For JVM – the [java.util.Base64 class](#).
- For JS – the [Buffer API](#).
- For Kotlin/Native – your own implementation.

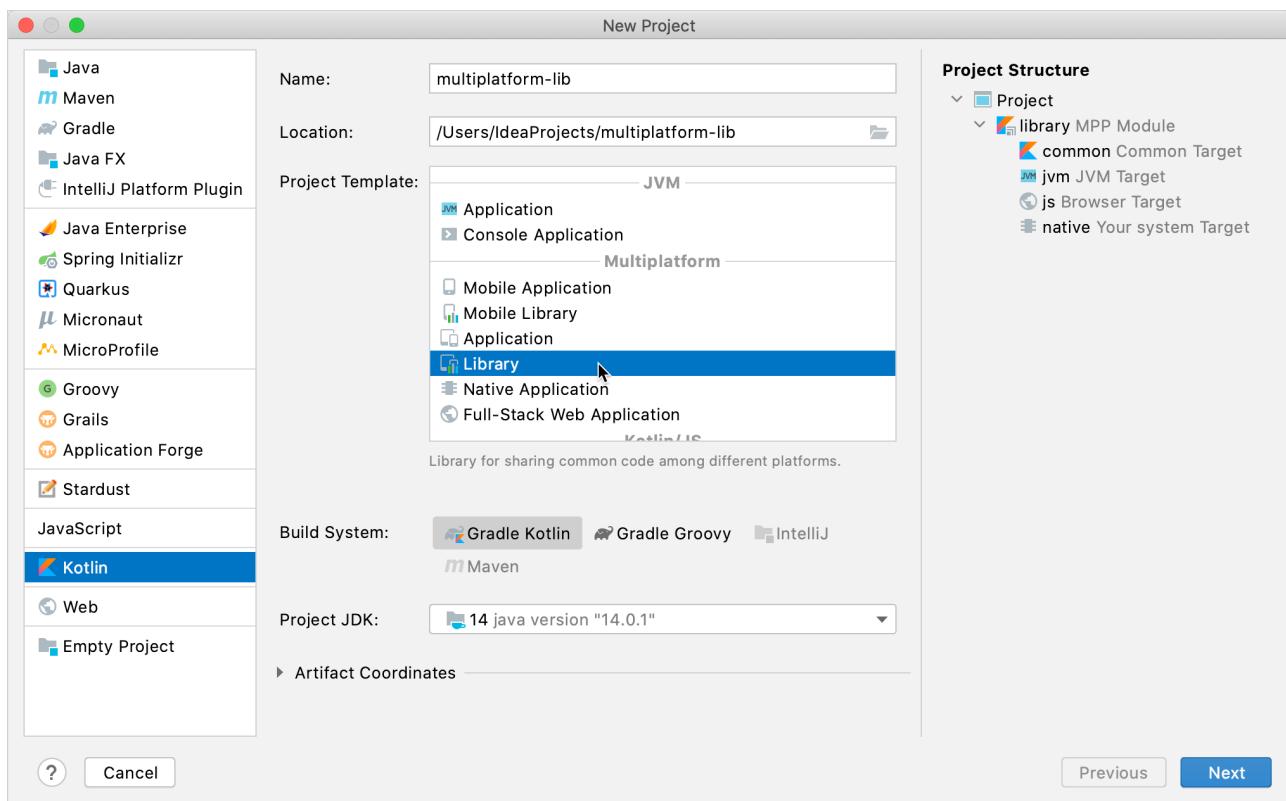
You will also test your code using common tests, and then publish the library to your local Maven repository.

Set up the environment

You can complete this tutorial on any operating system. Download and install the [latest version of IntelliJ IDEA](#) with the [latest Kotlin plugin](#).

Create a project

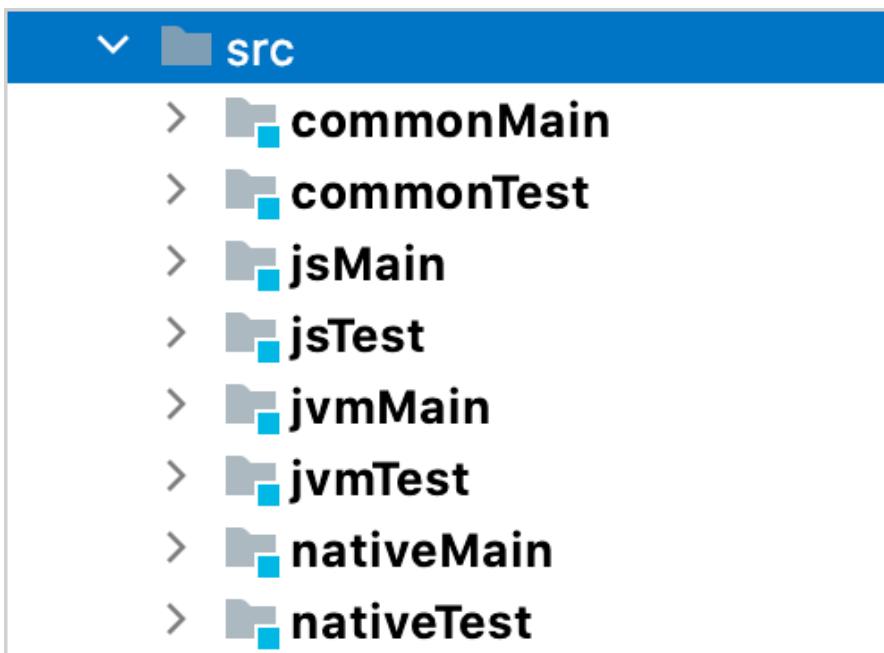
1. In IntelliJ IDEA, select File | New | Project.
2. In the left-hand panel, select Kotlin.
3. Enter a project name, then in the Multiplatform section select Library as the project template.



Select a project template

4. Select the Gradle DSL – Kotlin or Groovy.
5. Specify the JDK, which is required for developing Kotlin projects.
6. Click Next, and then click Finish.

The wizard will create a sample multiplatform library with the following structure:



Multiplatform library structure

Write cross-platform code

Define the classes and interfaces you are going to implement in the common code.

1. In the commonMain/kotlin directory, create the org.jetbrains.base64 package.
2. Create the Base64.kt file in the new package.
3. Define the Base64Encoder interface that converts bytes to the Base64 format:

```
package org.jetbrains.base64

interface Base64Encoder {
    fun encode(src: ByteArray): ByteArray
}
```

4. Define the Base64Factory object to provide an instance of the Base64Encoder interface to the common code:

```
expect object Base64Factory {
    fun createEncoder(): Base64Encoder
}
```

The factory object is marked with the expect keyword in the cross-platform code. For each platform, you should provide an actual implementation of the Base64Factory object with the platform-specific encoder. Learn more about [platform-specific implementations](#).

Provide platform-specific implementations

Now you will create the actual implementations of the Base64Factory object for each platform:

- JVM
- JS
- Native

JVM

1. In the jvmMain/kotlin directory, create the org.jetbrains.base64 package.
2. Create the Base64.kt file in the new package.
3. Provide a simple implementation of the Base64Factory object that delegates to the java.util.Base64 class:

IDEA inspections help create actual implementations for an expect declaration.

```
package org.jetbrains.base64
import java.util.*

actual object Base64Factory {
    actual fun createEncoder(): Base64Encoder = JvmBase64Encoder
}

object JvmBase64Encoder : Base64Encoder {
    override fun encode(src: ByteArray): ByteArray = Base64.getEncoder().encode(src)
}
```

Pretty simple, right? You've provided a platform-specific implementation by using a straightforward delegation to a third-party implementation.

JS

The JS implementation will be very similar to the JVM one.

1. In the jsMain/kotlin directory, create the org.jetbrains.base64 package.
2. Create the Base64.kt file in the new package.
3. Provide a simple implementation of the Base64Factory object that delegates to the NodeJS Buffer API:

```
package org.jetbrains.base64

actual object Base64Factory {
    actual fun createEncoder(): Base64Encoder = JsBase64Encoder
```

```

    }

object JsBase64Encoder : Base64Encoder {
    override fun encode(src: ByteArray): ByteArray {
        val buffer = js("Buffer").from(src)
        val string = buffer.toString("base64") as String
        return ByteArray(string.length) { string[it].toByte() }
    }
}

```

Native

Unfortunately, there is no third-party implementation available for all Kotlin/Native targets, so you need to write it yourself.

1. In the nativeMain/kotlin directory, create the org.jetbrains.base64 package.
2. Create the Base64.kt file in the new package.
3. Provide your own implementation for the Base64Factory object:

```

package org.jetbrains.base64

private val BASE64_ALPHABET: String =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
private val BASE64_MASK: Byte = 0x3f
private val BASE64_PAD: Char = '='
private val BASE64_INVERSE_ALPHABET = IntArray(256) {
    BASE64_ALPHABET.indexOf(it.toChar())
}

private fun Int.toBase64(): Char = BASE64_ALPHABET[this]

actual object Base64Factory {
    actual fun createEncoder(): Base64Encoder = NativeBase64Encoder
}

object NativeBase64Encoder : Base64Encoder {
    override fun encode(src: ByteArray): ByteArray {
        fun ByteArray.getOrZero(index: Int): Int = if (index >= size) 0 else get(index).toInt()
        // 4n / 3 is expected Base64 payload
        val result = ArrayList<Byte>(4 * src.size / 3)
        var index = 0
        while (index < src.size) {
            val symbolsLeft = src.size - index
            val padSize = if (symbolsLeft >= 3) 0 else (3 - symbolsLeft) * 8 / 6
            val chunk = (src.getOrZero(index) shr 16) or (src.getOrZero(index + 1) shr 8)
            or src.getOrZero(index + 2)
            index += 3

            for (i in 3 downTo padSize) {
                val char = (chunk shr (6 * i)) and BASE64_MASK.toInt()
                result.add(char.toBase64().code.toByte())
            }
            // Fill the pad with '='
        }
        return result.toByteArray()
    }
}

```

```

        repeat(padSize) { result.add(BASE64_PAD.code.toByte()) }

    }
}

return result.toByteArray()
}
}

```

Test your library

Now when you have actual implementations of the Base64Factory object for all platforms, it's time to test your multiplatform library.

To save time on testing, you can write common tests that will be executed on all platforms instead of testing each platform separately.

Prerequisites

Before writing tests, add the encodeToString method with the default implementation to the Base64Encoder interface, which is defined in commonMain/kotlin/org/jetbrains/base64/Base64.kt. This implementation converts byte arrays to strings, which are much easier to test.

```

interface Base64Encoder {
    fun encode(src: ByteArray): ByteArray

    fun encodeToString(src: ByteArray): String {
        val encoded = encode(src)
        return buildString(encoded.size) {
            encoded.forEach { append(it.toChar()) }
        }
    }
}

```

You can also provide a more efficient implementation of this method for a specific platform, for example, for JVM in jvmMain/kotlin/org/jetbrains/base64/Base64.kt:

```

objectJvmBase64Encoder : Base64Encoder {
    override fun encode(src: ByteArray): ByteArray = Base64.getEncoder().encode(src)
    override fun encodeToString(src: ByteArray): String =
        Base64.getEncoder().encodeToString(src)
}

```

One of the benefits of a multiplatform library is having a default implementation with optional platform-specific overrides.

Write common tests

Now you have a string-based API that you can cover with basic tests.

1. In the commonTest/kotlin directory, create the org.jetbrains.base64 package.
2. Create the Base64Test.kt file in the new package.
3. Add tests to this file:

```
package org.jetbrains.base64

import kotlin.test.Test

class Base64Test {
    @Test
    fun testEncodeToString() {
        checkEncodeToString("Kotlin is awesome", "S290bGluIGlzIGF3ZXNvbWU=")
    }

    @Test
    fun testPaddedStrings() {
        checkEncodeToString("", "")
        checkEncodeToString("1", "MQ==")
        checkEncodeToString("22", "MjI=")
        checkEncodeToString("333", "MzMz")
        checkEncodeToString("4444", "NDQ0NA==")
    }

    private fun checkEncodeToString(input: String, expectedOutput: String) {
        assertEquals(expectedOutput,
Base64Factory.createEncoder().encodeToString(input.asciiToByteArray()))
    }

    private fun String.asciiToByteArray() = ByteArray(length) {
        get(it).toByte()
    }
}
```

4. In the Terminal, execute the check Gradle task:

```
./gradlew check
```

You can also run the check Gradle task by double-clicking it in the list of Gradle tasks.

The tests will run on all platforms (JVM, JS, and Native).

Add platform-specific tests

You can also add tests that will be run only for a specific platform. For example, you can add UTF-16 tests on JVM. Just follow the same steps as for common tests, but create the Base64Test file in jvmTest/kotlin/org/jetbrains/base64:

```
package org.jetbrains.base64
```

```

import org.junit.Test
import kotlin.test.assertEquals

class Base64JvmTest {
    @Test
    fun testNonAsciiString() {
        val utf8String = "Gödel"
        val actual = Base64Factory.createEncoder().encodeToString(utf8String.toByteArray())
        assertEquals("R8O2ZGVs", actual)
    }
}

```

This test will automatically run on the JVM platform in addition to the common tests.

Publish your library to the local Maven repository

Your multiplatform library is ready for publishing so that you can use it in other projects.

To publish your library, use the [maven-publish Gradle plugin](#).

1. In the build.gradle.kts file, apply the maven-publish plugin and specify the group and version of your library:

Groovy

```

plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.5.20'
    id 'maven-publish'
}

group = 'org.jetbrains.base64'
version = '1.0.0'

```

Kotlin

```

plugins {
    kotlin("multiplatform") version "1.5.20"
    id("maven-publish")
}

group = "org.jetbrains.base64"
version = "1.0.0"

```

2. In the Terminal, run the publishToMavenLocal Gradle task to publish your library to your local Maven repository:

```
./gradlew publishToMavenLocal
```

You can also run the publishToMavenLocal Gradle task by double-clicking it in the list of Gradle tasks.

Your library will be published to the local Maven repository.

Add a dependency on the published library

Now you can add your library to other multiplatform projects as a dependency.

Add the mavenLocal() repository and add a dependency on your library to the build.gradle(.kts) file.

Groovy

```
repositories {  
    mavenCentral()  
    mavenLocal()  
}  
  
kotlin {  
    sourceSets {  
        commonMain {  
            dependencies {  
                implementation 'org.jetbrains.base64:Base64:1.0.0'  
            }  
        }  
    }  
}
```

Kotlin

```
repositories {  
    mavenCentral()  
    mavenLocal()  
}  
  
kotlin {  
    sourceSets {  
        val commonMain by getting {  
            dependencies {  
                implementation("org.jetbrains.base64:Base64:1.0.0")  
            }  
        }  
    }  
}
```

Summary

In this tutorial, you:

- Created a multiplatform library with platform-specific implementations.
- Wrote common tests that are executed on all platforms.
- Published your library to the local Maven repository.

What's next?

- Learn more about [publishing multiplatform libraries](#).
- Learn more about [Kotlin Multiplatform](#).
- [Create your first KMM application for Android and iOS – tutorial](#).
- [Create a full-stack web app with Kotlin Multiplatform – hands-on tutorial](#).

Build final native binaries

By default, a Kotlin/Native target is compiled down to a *.klib library artifact, which can be consumed by Kotlin/Native itself as a dependency but cannot be executed or used as a native library.

To declare final native binaries such as executables or shared libraries, use the binaries property of a native target. This property represents a collection of native binaries built for this target in addition to the default *.klib artifact and provides a set of methods for declaring and configuring them.

The kotlin-multiplatform plugin doesn't create any production binaries by default. The only binary available by default is a debug test executable that lets you run unit tests from the test compilation.

Declare binaries

Use the following factory methods to declare elements of the binaries collection.

Factory method	Binary kind	Available for
executable	Product executable	All native targets
test	Test executable	All native targets
sharedLib	Shared native library	All native targets, except for WebAssembly

Factory method	Binary kind	Available for
staticLib	Static native library	All native targets, except for WebAssembly
framework	Objective-C framework	macOS, iOS, watchOS, and tvOS targets only

The simplest version doesn't require any additional parameters and creates one binary for each build type. Currently, two build types are available:

- DEBUG – produces a non-optimized binary with debug information
- RELEASE – produces an optimized binary without debug information

The following snippet creates two executable binaries: debug and release.

```
kotlin {
    linuxX64 { // Use your target instead.
        binaries {
            executable {
                // Binary configuration.
            }
        }
    }
}
```

You can drop the lambda if there is no need for additional configuration:

```
binaries {
    executable()
}
```

You can specify for which build types to create binaries. In the following example, only the debug executable is created.

Groovy

```
binaries {
    executable([DEBUG]) {
        // Binary configuration.
    }
}
```

Kotlin

```
binaries {
    executable(listOf(DEBUG)) {
        // Binary configuration.
    }
}
```

```
    }  
}
```

You can also declare binaries with custom names.

Groovy

```
binaries {  
    executable('foo', [DEBUG]) {  
        // Binary configuration.  
    }  
  
    // It's possible to drop the list of build types (in which case, all the available build  
    // types will be used).  
    executable('bar') {  
        // Binary configuration.  
    }  
}
```

Kotlin

```
binaries {  
    executable("foo", listOf(DEBUG)) {  
        // Binary configuration.  
    }  
  
    // It's possible to drop the list of build types (in which case, all the available build  
    // types will be used).  
    executable("bar") {  
        // Binary configuration.  
    }  
}
```

The first argument sets a name prefix, which is the default name for the binary file. For example, for Windows the code produces the files foo.exe and bar.exe. You can also use the name prefix to [access the binary in the build script](#).

Access binaries

You can access binaries to [configure them](#) or get their properties (for example, the path to an output file).

You can get a binary by its unique name. This name is based on the name prefix (if it is specified), build type, and binary kind following the pattern: <optional-name-prefix><build-type><binary-kind>, for example, releaseFramework or testDebugExecutable.

Static and shared libraries have the suffixes static and shared respectively, for example, fooDebugStatic or barReleaseShared.

Groovy

```
// Fails if there is no such binary.  
binaries['fooDebugExecutable']  
binaries.fooDebugExecutable  
binaries.getByName('fooDebugExecutable')  
  
// Returns null if there is no such binary.  
binaries.findByName('fooDebugExecutable')
```

Kotlin

```
// Fails if there is no such binary.  
binaries["fooDebugExecutable"]  
binaries.getByName("fooDebugExecutable")  
  
// Returns null if there is no such binary.  
binaries.findByName("fooDebugExecutable")
```

Alternatively, you can access a binary by its name prefix and build type using typed getters.

Groovy

```
// Fails if there is no such binary.  
binaries.getExecutable('foo', DEBUG)  
binaries.getExecutable(DEBUG)           // Skip the first argument if the name prefix isn't set.  
binaries.getExecutable('bar', 'DEBUG') // You also can use a string for build type.  
  
// Similar getters are available for other binary kinds:  
// getFramework, getStaticLib and getSharedLib.  
  
// Returns null if there is no such binary.  
binaries.findExecutable('foo', DEBUG)  
  
// Similar getters are available for other binary kinds:  
// findFramework, findStaticLib and findSharedLib.
```

Kotlin

```
// Fails if there is no such binary.  
binaries.getExecutable("foo", DEBUG)  
binaries.getExecutable(DEBUG)           // Skip the first argument if the name prefix isn't set.  
binaries.getExecutable("bar", "DEBUG") // You also can use a string for build type.  
  
// Similar getters are available for other binary kinds:
```

```
// getFramework, getStaticLib and getSharedLib.

// Returns null if there is no such binary.
binaries.findExecutable("foo", DEBUG)

// Similar getters are available for other binary kinds:
// findFramework, findStaticLib and findSharedLib.
```

Export dependencies to binaries

When building an Objective-C framework or a native library (shared or static), you may need to pack not just the classes of the current project, but also the classes of its dependencies. Specify which dependencies to export to a binary using the `export` method.

Groovy

```
kotlin {
    sourceSets {
        macosMain.dependencies {
            // Will be exported.
            api project(':dependency')
            api 'org.example:exported-library:1.0'
            // Will not be exported.
            api 'org.example:not-exported-library:1.0'
        }
    }
    macosX64("macos").binaries {
        framework {
            export project(':dependency')
            export 'org.example:exported-library:1.0'
        }
        sharedLib {
            // It's possible to export different sets of dependencies to different binaries.
            export project(':dependency')
        }
    }
}
```

Kotlin

```
kotlin {
    sourceSets {
        macosMain.dependencies {
            // Will be exported.
            api(project(":dependency"))
            api("org.example:exported-library:1.0")
            // Will not be exported.
            api("org.example:not-exported-library:1.0")
        }
    }
    macosX64("macos").binaries {
```

```
framework {
    export(project(":dependency"))
    export("org.example:exported-library:1.0")
}
sharedLib {
    // It's possible to export different sets of dependencies to different binaries.
    export(project(':dependency'))
}
}
```

You can export only api dependencies of the corresponding source set.

You can export maven dependencies, but due to current limitations of Gradle metadata, such a dependency should be either a platform dependency (for example, kotlinx-coroutines-core-native_debug_macos_x64 instead of kotlinx-coroutines-core-native) or be exported transitively.

By default, export works non-transitively. This means that if you export the library foo depending on the library bar, only methods of foo are added to the output framework.

You can change this behavior using the transitiveExport flag. If set to true, the declarations of the library bar are exported as well.

Groovy

```
binaries {
    framework {
        export project(':dependency')
        // Export transitively.
        transitiveExport = true
    }
}
```

Kotlin

```
binaries {
    framework {
        export(project(":dependency"))
        // Export transitively.
        transitiveExport = true
    }
}
```

For example, assume that you write several modules in Kotlin and then want to access them from Swift. Since usage of several Kotlin/Native frameworks in one Swift application is limited, you can create a single umbrella framework and export all these modules to it.

Build universal frameworks

By default, an Objective-C framework produced by Kotlin/Native supports only one platform. However, you can merge such frameworks into a single universal (fat) binary using the [lipo tool](#). This operation especially makes sense for 32-bit and 64-bit iOS frameworks. In this case, you can use the resulting universal framework on both 32-bit and 64-bit devices.

The fat framework must have the same base name as the initial frameworks.

Groovy

```
import org.jetbrains.kotlin.gradle.tasks.FatFrameworkTask

kotlin {
    // Create and configure the targets.
    targets {
        iosArm32("ios32")
        iosArm64("ios64")
        configure([ios32, ios64]) {
            binaries.framework {
                basePath = "my_framework"
            }
        }
    }
    // Create a task building a fat framework.
    tasks.register("debugFatFramework", FatFrameworkTask) {
        // The fat framework must have the same base name as the initial frameworks.
        basePath = "my_framework"
        // The default destination directory is "<build directory>/fat-framework".
        destinationDir = file("$buildDir/fat-framework/debug")
        // Specify the frameworks to be merged.
        from(
            targets.ios32.binaries.getFramework("DEBUG"),
            targets.ios64.binaries.getFramework("DEBUG")
        )
    }
}
```

Kotlin

```
import org.jetbrains.kotlin.gradle.tasks.FatFrameworkTask

kotlin {
    // Create and configure the targets.
    val ios32 = iosArm32("ios32")
    val ios64 = iosArm64("ios64")
    configure(listOf(ios32, ios64)) {
        binaries.framework {
            basePath = "my_framework"
        }
    }
}
```

```

// Create a task to build a fat framework.
tasks.register<FatFrameworkTask>("debugFatFramework") {
    // The fat framework must have the same base name as the initial frameworks.
    basePath = "my_framework"
    // The default destination directory is "<build directory>/fat-framework".
    destinationDir = buildDir.resolve("fat-framework/debug")
    // Specify the frameworks to be merged.
    from(
        ios32.binaries.getFramework("DEBUG"),
        ios64.binaries.getFramework("DEBUG")
    )
}
}

```

Supported platforms

Kotlin supports the following platforms and provides target presets for each platform. See how to [use a target preset](#).

Target platform	Target preset	Comments
Kotlin/JVM	jvm	
Kotlin/JS	js	<p>Select the execution environment:</p> <ul style="list-style-type: none"> • browser {} for applications running in the browser. • nodejs{} for applications running on Node.js. <p>Learn more in Setting up a Kotlin/JS project.</p>
Android applications and libraries	android	<p>Manually apply an Android Gradle plugin – com.android.application or com.android.library.</p> <p>You can only create one Android target per Gradle subproject.</p>
Android NDK	androidNativeArm32, androidNativeArm64	<p>The 64-bit target requires a Linux or macOS host.</p> <p>You can build the 32-bit target on any supported host.</p>
iOS	iosArm32, iosArm64, iosX64	Requires a macOS host.
watchOS	watchosArm32, watchosArm64, watchosX86, watchosX64	Requires a macOS host.

Target platform	Target preset	Comments
tvOS	tvosArm64, tvosX64	Requires a macOS host.
macOS	macosX64	Requires a macOS host.
Linux	linuxArm64, linuxArm32Hfp, linuxMips32, linuxMipsel32, linuxX64	Linux MIPS targets (linuxMips32 and linuxMipsel32) require a Linux host. You can build other Linux targets on any supported host.
Windows	mingwX64, mingwX86	Requires a Windows host.
WebAssembly	wasm32	

A target that is not supported by the current host is ignored during building and therefore not published.

Multiplatform Gradle DSL reference

Multiplatform projects are in [Alpha](#). Language features and tooling may change in future Kotlin versions.

The Kotlin Multiplatform Gradle plugin is a tool for creating [Kotlin multiplatform](#) projects. Here we provide a reference of its contents; use it as a reminder when writing Gradle build scripts for Kotlin multiplatform projects. Learn the [concepts of Kotlin multiplatform projects, how to create and configure them](#).

Id and version

The fully qualified name of the Kotlin Multiplatform Gradle plugin is `org.jetbrains.kotlin.multiplatform`. If you use the Kotlin Gradle DSL, you can apply the plugin with `kotlin("multiplatform")`. The plugin versions match the Kotlin release versions. The most recent version is 1.5.20.

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.5.20'
}
```

```
plugins {
    kotlin("multiplatform") version "1.5.20"
}
```

Top-level blocks

kotlin is the top-level block for multiplatform project configuration in the Gradle build script. Inside kotlin, you can write the following blocks:

Block	Description
<targetName>	Declares a particular target of a project. The names of available targets are listed in the Targets section.
targets	All targets of the project.
presets	All predefined targets. Use this for configuring multiple predefined targets at once.
sourceSets	Configures predefined and declares custom source sets of the project.

Targets

Target is a part of the build responsible for compiling, testing, and packaging a piece of software aimed for one of the [supported platforms](#).

Each target can have one or more [compilations](#). In addition to default compilations for test and production purposes, you can [create custom compilations](#).

The targets of a multiplatform project are described in the corresponding blocks inside kotlin, for example, jvm, android, iosArm64. The complete list of available targets is the following:

Name	Description
jvm	Java Virtual Machine
js	JavaScript
android	Android (APK)
androidNativeArm32	Android NDK on ARM (ARM32) platforms

Name	Description
androidNativeArm64	Android NDK on ARM64 platforms
androidNativeX86	Android NDK on x86 platforms
androidNativeX64	Android NDK on x86_64 platforms
iosArm32	Apple iOS on ARM (ARM32) platforms (Apple iPhone 5 and earlier)
iosArm64	Apple iOS on ARM64 platforms (Apple iPhone 5s and newer)
iosX64	Apple iOS 64-bit simulator
watchosArm32	Apple watchOS on ARM (ARM32) platforms (Apple Watch Series 3 and earlier)
watchosArm64	Apple watchOS on ARM64_32 platforms (Apple Watch Series 4 and newer)
watchosX86	Apple watchOS 32-bit simulator (watchOS 6.3 and earlier)
watchosX64	Apple watchOS 64-bit simulator (watchOS 7.0 and newer)
tvosArm64	Apple tvOS on ARM64 platforms (Apple TV 4th generation and newer)
tvosX64	Apple tvOS simulator
linuxArm64	Linux on ARM64 platforms, for example, Raspberry Pi
linuxArm32Hfp	Linux on hard-float ARM (ARM32) platforms
linuxMips32	Linux on MIPS platforms
linuxMipsel32	Linux on little-endian MIPS (mipsel) platforms
linuxX64	Linux on x86_64 platforms
macosX64	Apple macOS
mingwX64	64-bit Microsoft Windows
mingwX86	32-bit Microsoft Windows

Name	Description
wasm32	WebAssembly

```

kotlin {
    jvm()
    iosX64()
    macosX64()
    js().browser()
}

```

Configuration of a target can include two parts:

- [Common configuration](#) available for all targets.
- Target-specific configuration.

Each target can have one or more [compilations](#).

Common target configuration

In any target block, you can use the following declarations:

Name	Description
attributes	Attributes used for disambiguating targets for a single platform.
preset	The preset that the target has been created from, if any.
platformType	Designates the Kotlin platform of this target. Available values: jvm, androidJvm, js, native, common.
artifactsTaskName	The name of the task that builds the resulting artifacts of this target.
components	The components used to setup Gradle publications.

JVM targets

In addition to [common target configuration](#), jvm targets have a specific function:

Name	Description
withJava()	Includes Java sources into the JVM target's compilations.

Use this function for projects that contain both Java and Kotlin source files. Note that the default source directories for

Java sources don't follow the Java plugin's defaults. Instead, they are derived from the Kotlin source sets. For example, if the JVM target has the default name jvm, the paths are src/jvmMain/java (for production Java sources) and src/jvmTest/java for test Java sources. Learn how to [include Java sources in JVM compilations](#).

```
kotlin {  
    jvm {  
        withJava()  
    }  
}
```

JavaScript targets

The js block describes the configuration of JavaScript targets. It can contain one of two blocks depending on the target execution environment:

Name	Description
browser	Configuration of the browser target.
nodejs	Configuration of the Node.js target.

Learn more about [configuring Kotlin/JS projects](#).

Browser

browser can contain the following configuration blocks:

Name	Description
testRuns	Configuration of test execution.
runTask	Configuration of project running.
webpackTask	Configuration of project bundling with Webpack .
dceTask	Configuration of Dead Code Elimination .
distribution	Path to output files.

```
kotlin {  
    js().browser {  
        webpackTask { /* ... */ }  
        testRuns { /* ... */ }  
        dceTask {  
            keep("myKotlinJsApplication.org.example.keepFromDce")  
        }  
    }  
}
```

```

        }
        distribution {
            directory = File("$projectDir/customdir/")
        }
    }
}

```

Node.js

nodejs can contain configurations of test and run tasks:

Name	Description
testRuns	Configuration of test execution.
runTask	Configuration of project running.

kotlin {	
js().nodejs {	
runTask { /* ... */ }	
testRuns { /* ... */ }	
}	
}	

Native targets

For native targets, the following specific blocks are available:

Name	Description
binaries	Configuration of <u>binaries</u> to produce.
cinterop	Configuration of <u>interop with C libraries</u> .

Binaries

There are the following kinds of binaries:

Name	Description
executable	Product executable.
test	Test executable.

Name	Description
------	-------------

sharedLib Shared library.

staticLib Static library.

framework Objective-C framework.

```
kotlin {
    linuxX64 { // Use your target instead.
        binaries {
            executable {
                // Binary configuration.
            }
        }
    }
}
```

For binaries configuration, the following parameters are available:

Name	Description
------	-------------

compilation The compilation from which the binary is built. By default, test binaries are based on the test compilation while other binaries - on the main compilation.

linkerOpts Options passed to a system linker during binary building.

baseName Custom base name for the output file. The final file name will be formed by adding system-dependent prefix and postfix to this base name.

entryPoint The entry point function for executable binaries. By default, it's main() in the root package.

outputFile Access to the output file.

linkTask Access to the link task.

runTask Access to the run task for executable binaries. For targets other than linuxX64, macosX64, or mingwX64 the value is null.

isStatic For Objective-C frameworks. Includes a static library instead of a dynamic one.

Groovy

```
binaries {
    executable('my_executable', [RELEASE]) {
```

```

// Build a binary on the basis of the test compilation.
compilation = compilations.test

// Custom command line options for the linker.
linkerOpts = ['-L/lib/search/path', '-L/another/search/path', '-lmylib']

// Base name for the output file.
baseName = 'foo'

// Custom entry point function.
entryPoint = 'org.example.main'

// Accessing the output file.
println("Executable path: ${outputFile.getAbsolutePath()")

// Accessing the link task.
linkTask.dependsOn(additionalPreprocessingTask)

// Accessing the run task.
// Note that the runTask is null for non-host platforms.
runTask?.dependsOn(prepareForRun)
}

framework('my_framework' [RELEASE]) {
    // Include a static library instead of a dynamic one into the framework.
    isStatic = true
}
}

```

Kotlin

```

binaries {
    executable("my_executable", listOf(RELEASE)) {
        // Build a binary on the basis of the test compilation.
        compilation = compilations["test"]

        // Custom command line options for the linker.
        linkerOpts = mutableListOf("-L/lib/search/path", "-L/another/search/path", "-lmylib")

        // Base name for the output file.
        baseName = "foo"

        // Custom entry point function.
        entryPoint = "org.example.main"

        // Accessing the output file.
        println("Executable path: ${outputFile.getAbsolutePath()")

        // Accessing the link task.
        linkTask.dependsOn(additionalPreprocessingTask)

        // Accessing the run task.
        // Note that the runTask is null for non-host platforms.
        runTask?.dependsOn(prepareForRun)
    }

    framework("my_framework" listOf(RELEASE)) {
        // Include a static library instead of a dynamic one into the framework.
    }
}

```

```
        isStatic = true
    }
}
```

Learn more about [building native binaries](#).

CInterops

cinterops is a collection of descriptions for interop with native libraries. To provide an interop with a library, add an entry to cinterops and define its parameters:

Name	Description
defFile	def file describing the native API.
packageName	Package prefix for the generated Kotlin API.
compilerOpts	Options to pass to the compiler by the cinterop tool.
includeDirs	Directories to look for headers.

Learn more how to [configure interop with native languages](#).

Groovy

```
kotlin {
    linuxX64 { // Replace with a target you need.
        compilations.main {
            cinterops {
                myInterop {
                    // Def-file describing the native API.
                    // The default path is src/nativeInterop/cinterop/<interop-name>.def
                    defFile project.file("def-file.def")

                    // Package to place the Kotlin API generated.
                    packageName 'org.sample'

                    // Options to be passed to compiler by cinterop tool.
                    compilerOpts '-Ipath/to/headers'

                    // Directories for header search (an analogue of the -I<path> compiler
                    option).
                    includeDirs.allHeaders("path1", "path2")

                    // A shortcut for includeDirs.allHeaders.
                    includeDirs("include/directory", "another/directory")
                }
            anotherInterop { /* ... */ }
        }
    }
}
```

```
        }
    }
}
```

Kotlin

```
kotlin {
    linuxX64 { // Replace with a target you need.
        compilations.getByName("main") {
            val myInterop by cinterop.creating {
                // Def-file describing the native API.
                // The default path is src/nativeInterop/cinterop/<interop-name>.def
                defFile(project.file("def-file.def"))

                // Package to place the Kotlin API generated.
                packageName("org.sample")

                // Options to be passed to compiler by cinterop tool.
                compilerOpts("-Ipath/to/headers")

                // Directories for header search (an analogue of the -I<path> compiler option).
                includeDirs.allHeaders("path1", "path2")

                // A shortcut for includeDirs.allHeaders.
                includeDirs("include/directory", "another/directory")
            }

            val anotherInterop by cinterop.creating { /* ... */ }
        }
    }
}
```

Android targets

The Kotlin multiplatform plugin contains two specific functions for android targets. Two functions help you configure [build variants](#):

Name	Description
<code>publishLibraryVariants()</code>	Specifies build variants to publish. Learn more about publishing Android libraries .
<code>publishAllLibraryVariants()</code>	Publishes all build variants.

```
kotlin {
    android {
        publishLibraryVariants("release", "debug")
    }
}
```

Learn more about [compilation for Android](#).

The android configuration inside kotlin doesn't replace the build configuration of any Android project. Learn more about writing build scripts for Android projects in [Android developer documentation](#).

Source sets

The sourceSets block describes source sets of the project. A source set contains Kotlin source files that participate in compilations together, along with their resources, dependencies, and language settings.

A multiplatform project contains [predefined](#) source sets for its targets; developers can also create [custom](#) source sets for their needs.

Predefined source sets

Predefined source sets are set up automatically upon creation of a multiplatform project. Available predefined source sets are the following:

Name	Description
commonMain	Code and resources shared between all platforms. Available in all multiplatform projects. Used in all main compilations of a project.
commonTest	Test code and resources shared between all platforms. Available in all multiplatform projects. Used in all test compilations of a project.
<targetName>	Target-specific sources for a compilation. <targetName> is the name of a predefined target and <compilationName> <compilationName> is the name of a compilation for this target. Examples: jsTest, jvmMain.

With Kotlin Gradle DSL, the sections of predefined source sets should be marked by getting.

Groovy

```
kotlin {  
    sourceSets {  
        commonMain { /* ... */ }  
    }  
}
```

Kotlin

```
kotlin {  
    sourceSets {
```

```
    val commonMain by getting { /* ... */ }
}
}
```

Learn more about [source sets](#).

Custom source sets

Custom source sets are created by the project developers manually. To create a custom source set, add a section with its name inside the `sourceSets` section. If using Kotlin Gradle DSL, mark custom source sets by creating.

Groovy

```
kotlin {
    sourceSets {
        myMain { /* ... */ } // create or configure a source set by the name 'myMain'
    }
}
```

Kotlin

```
kotlin {
    sourceSets {
        val myMain by creating { /* ... */ } // create a new source set by the name 'MyMain'
    }
}
```

Note that a newly created source set isn't connected to other ones. To use it in the project's compilations, [connect it with other source sets](#).

Source set parameters

Configurations of source sets are stored inside the corresponding blocks of `sourceSets`. A source set has the following parameters:

Name	Description
<code>kotlin.srcDir</code>	Location of Kotlin source files inside the source set directory.
<code>resources.srcDir</code>	Location of resources inside the source set directory.
<code>dependsOn</code>	Connection with another source set .

Name	Description
dependencies	Dependencies of the source set.
languageSettings	Language settings applied to the source set.

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            kotlin.srcDir('src')
            resources.srcDir('res')

            dependencies {
                /* ... */
            }
        }
    }
}
```

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting {
            kotlin.srcDir("src")
            resources.srcDir("res")

            dependencies {
                /* ... */
            }
        }
    }
}
```

Compilations

A target can have one or more compilations, for example, for production or testing. There are [predefined compilations](#) that are added automatically upon target creation. You can additionally create [custom compilations](#).

To refer to all or some particular compilations of a target, use the `compilations` object collection. From `compilations`, you can refer to a compilation by its name.

Learn more about [configuring compilations](#).

Predefined compilations

Predefined compilations are created automatically for each target of a project except for Android targets. Available predefined compilations are the following:

Name Description

main Compilation for production sources.

test Compilation for tests.

Groovy

```
kotlin {  
    jvm {  
        compilations.main.output // get the main compilation output  
        compilations.test.runtimeDependencyFiles // get the test runtime classpath  
    }  
}
```

Kotlin

```
kotlin {  
    jvm {  
        val main by compilations.getting {  
            output // get the main compilation output  
        }  
  
        compilations["test"].runtimeDependencyFiles // get the test runtime classpath  
    }  
}
```

Custom compilations

In addition to predefined compilations, you can create your own custom compilations. To create a custom compilation, add a new item into the compilations collection. If using Kotlin Gradle DSL, mark custom compilations by creating.

Learn more about creating a [custom compilation](#).

Groovy

```
kotlin {  
    jvm() {  
        compilations.create('integrationTest') {  
            defaultSourceSet {  
                dependencies {  
                    /* ... */  
                }  
            }  
        }  
    }  
}
```

```

        }

        // Create a test task to run the tests produced by this compilation:
        tasks.register('jvmIntegrationTest', Test) {
            /* ... */
        }
    }
}

```

Kotlin

```

kotlin {
    jvm() {
        compilations {
            val integrationTest by compilations.creating {
                defaultSourceSet {
                    dependencies {
                        /* ... */
                    }
                }
            }

            // Create a test task to run the tests produced by this compilation:
            tasks.register<Test>("integrationTest") {
                /* ... */
            }
        }
    }
}

```

Compilation parameters

A compilation has the following parameters:

Name	Description
defaultSourceSet	The compilation's default source set.
kotlinSourceSets	Source sets participating in the compilation.
allKotlinSourceSets	Source sets participating in the compilation and their connections via <code>dependsOn()</code> .
kotlinOptions	Compiler options applied to the compilation. For the list of available options, see Compiler options .
compileKotlinTask	Gradle task for compiling Kotlin sources.
compileKotlinTaskName	Name of <code>compileKotlinTask</code> .

Name	Description
compileAllTaskName	Name of the Gradle task for compiling all sources of a compilation.
output	The compilation output.
compileDependencyFiles	Compile-time dependency files (classpath) of the compilation.
runtimeDependencyFiles	Runtime dependency files (classpath) of the compilation.

Groovy

```

kotlin {
    jvm {
        compilations.main.kotlinOptions {
            // Setup the Kotlin compiler options for the 'main' compilation:
            jvmTarget = "1.8"
        }

        compilations.main.compileKotlinTask // get the Kotlin task 'compileKotlinJvm'
        compilations.main.output // get the main compilation output
        compilations.test.runtimeDependencyFiles // get the test runtime classpath
    }

    // Configure all compilations of all targets:
    targets.all {
        compilations.all {
            kotlinOptions {
                allWarningsAsErrors = true
            }
        }
    }
}

```

Kotlin

```

kotlin {
    jvm {
        val main by compilations.getting {
            kotlinOptions {
                // Setup the Kotlin compiler options for the 'main' compilation:
                jvmTarget = "1.8"
            }

            compileKotlinTask // get the Kotlin task 'compileKotlinJvm'
            output // get the main compilation output
        }

        compilations["test"].runtimeDependencyFiles // get the test runtime classpath
    }
}

```

```
// Configure all compilations of all targets:
targets.all {
    compilations.all {
        kotlinOptions {
            allWarningsAsErrors = true
        }
    }
}
```

Dependencies

The dependencies block of the source set declaration contains the dependencies of this source set.

Learn more about [configuring dependencies](#).

There are four types of dependencies:

Name	Description
api	Dependencies used in the API of the current module.
implementation	Dependencies used in the module but not exposed outside it.
compileOnly	Dependencies used only for compilation of the current module.
runtimeOnly	Dependencies available at runtime but not visible during compilation of any module.

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                api 'com.example:foo-metadata:1.0'
            }
        }
        jvm6Main {
            dependencies {
                implementation 'com.example:foo-jvm6:1.0'
            }
        }
    }
}
```

Kotlin

```

kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                api("com.example:foo-metadata:1.0")
            }
        }
        val jvm6Main by getting {
            dependencies {
                implementation("com.example:foo-jvm6:1.0")
            }
        }
    }
}

```

Additionally, source sets can depend on each other and form a hierarchy. In this case, the `dependsOn()` relation is used.

Source set dependencies can also be declared in the top-level dependencies block of the build script. In this case, their declarations follow the pattern <sourceSetName><DependencyKind>, for example, `commonMainApi`.

Groovy

```

dependencies {
    commonMainApi 'com.example:foo-common:1.0'
    jvm6MainApi 'com.example:foo-jvm6:1.0'
}

```

Kotlin

```

dependencies {
    "commonMainApi"("com.example:foo-common:1.0")
    "jvm6MainApi"("com.example:foo-jvm6:1.0")
}

```

Language settings

The `languageSettings` block of a source set defines certain aspects of project analysis and build. The following language settings are available:

Name	Description
<code>languageVersion</code>	Provides source compatibility with the specified version of Kotlin.
<code>apiVersion</code>	Allows using declarations only from the specified version of Kotlin bundled libraries.

Name	Description
enableLanguageFeature	Enables the specified language feature. The available values correspond to the language features that are currently experimental or have been introduced as such at some point.
useExperimentalAnnotation	Allows using the specified opt-in annotation .
progressiveMode	Enables the progressive mode .

Groovy

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.4' // possible values: '1.0', '1.1', '1.2', '1.3', '1.4'
            apiVersion = '1.4' // possible values: '1.0', '1.1', '1.2', '1.3', '1.4'
            enableLanguageFeature('InlineClasses') // language feature name
            useExperimentalAnnotation('kotlin.ExperimentalUnsignedTypes') // annotation FQ-name
            progressiveMode = true // false by default
        }
    }
}
```

Kotlin

```
kotlin {
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.4" // possible values: "1.0", "1.1", "1.2", "1.3", "1.4"
            apiVersion = "1.4" // possible values: "1.0", "1.1", "1.2", "1.3", "1.4"
            enableLanguageFeature("InlineClasses") // language feature name
            useExperimentalAnnotation("kotlin.ExperimentalUnsignedTypes") // annotation FQ-name
            progressiveMode = true // false by default
        }
    }
}
```

Migrating multiplatform projects to Kotlin 1.4.0

Kotlin 1.4.0 comes with lots of features and improvements in the tooling for multiplatform programming. Some of them just work out of the box on existing projects, and some require additional configuration steps. This guide will help you migrate your multiplatform projects to 1.4.0 or higher and get the benefits of all its new features.

For multiplatform project authors

Update Gradle

Starting with 1.4.0, Kotlin multiplatform projects require Gradle 6.0 or later. Make sure that your projects use the proper version of Gradle and upgrade it if needed. See the [Gradle documentation](#) for non-Kotlin-specific migration instructions.

Simplify your build configuration

Gradle module metadata provides rich publishing and dependency resolution features that are used in Kotlin Multiplatform Projects. In Gradle 6.0 and above, module metadata is used in dependency resolution and included in publications by default. Thus, once you update to such a version, you can remove `enableFeaturePreview("GRADLE_METADATA")` from the project's `settings.gradle` file.

If you use libraries published with metadata, you only have to specify dependencies on them only once in the shared source set, as opposed to specifying dependencies on different variants of the same library in the shared and platform-specific source sets prior to 1.4.0.

Starting from 1.4.0, you also no longer need to declare a dependency on `stdlib` in each source set manually – it [will now be added by default](#). The version of the automatically added standard library will be the same as the version of the Kotlin Gradle plugin, since they have the same versioning.

With these features, you can make your Gradle build file much more concise and easy to read:

```
//...
android()
ios()
js()

sourceSets {
    commonMain {
        dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:$coroutinesVersion")
        }
    }
}
//...
```

Don't use `kotlinx` library artifact names with suffixes `-common` or `-native`, as they are no longer supported. Instead, use the library root artifact name, which in the example above is `kotlinx-coroutines-core`.

Try the hierarchical project structure

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

By enabling the hierarchical structure along with its ability to use platform-dependent libraries in shared source sets, you can eliminate the need to use certain workarounds to get IDE support for sharing source sets among several native

targets, for example iosArm64 and iosX64:

```
kotlin {  
    // workaround 1: select iOS target platform depending on the Xcode environment variables  
    val iosTarget: (String, KotlinNativeTarget.() -> Unit) -> KotlinNativeTarget =  
        if (System.getenv("SDK_NAME")?.startsWith("iphoneos") == true)  
            ::iosArm64  
        else  
            ::iosX64  
  
    iosTarget("ios")  
}
```

```
# workaround 2: make symbolic links to use one source set for two targets  
ln -s iosMain iosArm64Main && ln -s iosMain iosX64Main
```

Instead of doing this, you can create a hierarchical structure with [target shortcuts](#) available for typical multi-target scenarios, or you can manually declare and connect the source sets. For example, you can create two iOS targets and a shared source set with the ios() shortcut:

```
kotlin {  
    ios() // iOS device and simulator targets; iosMain and iosTest source sets  
}
```

To enable the hierarchical project structure along with the use of platform-dependent libraries in shared source sets, just add the following to your gradle.properties:

```
kotlin.mpp.enableGranularSourceSetsMetadata=true  
kotlin.native.enableDependencyPropagation=false
```

In future versions, the hierarchical project structure will become default for Kotlin multiplatform project, so we strongly encourage you to start using it now.

For library authors

Migrate from Gradle Bintray plugin to Maven Publish plugin

If you're using gradle-bintray-plugin for library publication, migrate your projects to maven-publish plugin instead. [See how we've done this for kotlinc.serialization](#). Learn more about [publishing multiplatform libraries](#).

If for some reason you need to publish to Bintray and use the Gradle Bintray plugin, remember that this plugin doesn't support publishing Gradle module metadata. Use [this workaround](#) to fix this.

Follow the default libraries' layout

The layout of kotlinx libraries has changed and now corresponds to the default layout, which we recommend using: The root or umbrella library module now has a name without a suffix (for example, kotlinx-coroutines-core instead of kotlinx-coroutines-core-native). Publishing libraries with [maven-publish Gradle plugin](#) follows this layout by default. Learn more about [publishing multiplatform libraries](#).

Migrate to the hierarchical project structure

A hierarchical project structure allows reusing code in similar targets, as well as publishing and consuming libraries with granular APIs targeting similar platforms. We recommend that you switch to the hierarchical project structure in your libraries when migrating to Kotlin 1.4.0 or higher:

- By default, libraries published with the hierarchical project structure are compatible only with projects that have hierarchical project structure. To enable compatibility with non-hierarchical projects, add the following to the gradle.properties file in your library project:

```
kotlin.mpp.enableCompatibilityMetadataVariant=true
```

- Libraries published without the hierarchical project structure can't be used in a shared native source set. For example, users with ios() shortcuts in their build.gradle.(kts) files won't be able to use your library in their iOS-shared code.

The compatibility between multiplatform projects and libraries is as follows:

Library with hierarchical project structure	Project with hierarchical project structure	Compatibility
Yes	Yes	
Yes	No	Need to enable with enableCompatibilityMetadataVariant
No	Yes	Library can't be used in a shared native source set
No	Yes	

In future versions, the hierarchical project structure with the usage of platform-dependent libraries in shared source sets will be the default in multiplatform projects. So the sooner you support it, the sooner users will be able to migrate. We'll also be very grateful if you report any bugs you find to our issue tracker.

To enable hierarchical project structure support, add the following to your gradle.properties file:

```
kotlin.mpp.enableGranularSourceSetsMetadata=true  
kotlin.mpp.enableCompatibilityMetadataVariant=true // to enable compatibility with projects  
without hierarchical structure
```

For build authors

Check task names

The introduction of the hierarchical project structure in multiplatform projects resulted in a couple of changes to the names of some Gradle tasks:

- The metadataJar task has been renamed to allMetadataJar.
- There are new compile<SourceSet>KotlinMetadata tasks for all published intermediate source sets.

These changes are relevant only for projects with the hierarchical project structure.

For using the Kotlin/JS target

Changes related to npm dependency management

When declaring dependencies on npm packages, you are now required to explicitly specify a version or version range based on [npm's semver syntax](#). Specifying multiple version ranges is also supported.

While we don't recommend it, you can use a wildcard * in place of a version number if you do not want to specify a version or version range explicitly.

Changes related to the Kotlin/JS IR compiler

Kotlin 1.4.0 introduces the Alpha IR compiler for Kotlin/JS. Learn more about the [Kotlin/JS IR compiler's backend](#) and [how to configure it](#).

To choose between the different Kotlin/JS compiler options, set the key kotlin.js.compiler in your gradle.properties to legacy, ir, or both. Alternatively, pass LEGACY, IR, or BOTH to the js function in your build.gradle(kts).

```
kotlin {  
    js(IR) { // or: LEGACY, BOTH  
        // ...  
    }  
    binaries.executable()  
}
```

Changes in both mode

Choosing both as the compiler option (so that it will compile with both the legacy and the IR backend) means that some Gradle tasks are renamed to explicitly mark them as only affecting the legacy compilation. compileKotlinJs is renamed to compileKotlinJsLegacy, and compileTestKotlinJs is renamed to compileTestKotlinJsLegacy.

Explicitly toggling the creation of executable files

When using the IR compiler, the `binaries.executable()` instruction must be present in the `js` target configuration block of your `build.gradle(.kts)`. If this option is omitted, only Kotlin-internal library files are generated. These files can be used from other projects, but not run on their own.

For backwards compatibility, when using the legacy compiler for Kotlin/JS, including or omitting `binaries.executable()` will have no effect – executable files will be generated in either case. To make the legacy backend stop producing executable files without the presence of `binaries.executable()` (for example, to improve build times where runnable artifacts aren't required), set `kotlin.js.generate.executable.default=false` in your `gradle.properties`.

Changes related to Dukat

The Dukat integration for Gradle has received minor naming and functionality changes with Kotlin 1.4.0.

- The `kotlin.js.experimental.generateKotlinExternals` flag has been renamed to `kotlin.js.generate.externals`. It controls the default behavior of Dukat for all specified npm dependencies.
- The npm dependency function now takes a third parameter after the package name and version: `generateExternals`. This allows you to individually control whether Dukat should generate declarations for a specific dependency, and it overrides the `generateKotlinExternals` setting.

Learn how to [manually trigger the generation of Kotlin externals](#).

Using artifacts built with Kotlin 1.4.x in a Kotlin 1.3.x project

The choice between the IR and LEGACY compilers was not yet available in Kotlin 1.3.xx. Because of this, you may encounter a Gradle error `Cannot choose between the following variants...` if one of your dependencies (or any transitive dependency) was built using Kotlin 1.4+ but your project uses Kotlin 1.3.xx. A workaround is provided [here](#).

Tutorials for Kotlin Multiplatform

A number of tutorials are available to help you get acquainted with [Kotlin Multiplatform](#). These tutorials come in two varieties:

- Plain tutorials describe how to complete a specific task from start to finish, with screenshots and code samples.
- Hands-on tutorials provide a self-contained project related to a specific topic.

Name	Type	You will learn how to
Creating your first KMM application	Tutorial	Create a basic Kotlin Multiplatform Mobile (KMM) application using the KMM plugin for Android Studio.
Networking and data storage	Hands-on	Create a KMM application for Android and iOS that will retrieve data over the internet from a public API, save it in a local database, and display it in a list in the application.

Name	Type	You will learn how to
Creating and publishing a multiplatform library	Tutorial	Create a multiplatform library that is available for JVM, JS, and Native, and that can be used from any other common code.
Building a full-stack web application	Hands-on	Create a client-server application that targets Kotlin/JVM and Kotlin/JS and uses common code, serialization, and other multiplatform paradigms.

Get started with Kotlin/JVM

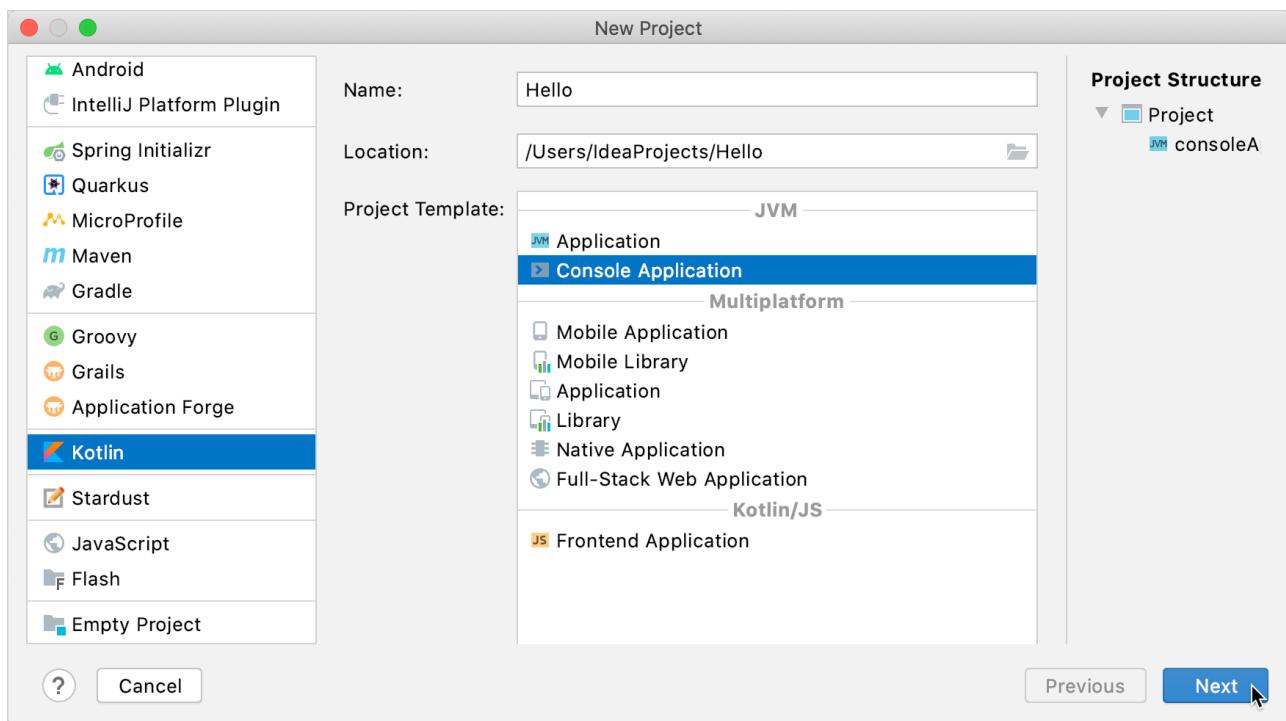
This tutorial demonstrates how to use IntelliJ IDEA for creating a console application.

To get started, first download and install the latest version of [IntelliJ IDEA](#).

Create an application

Once you've installed IntelliJ IDEA, it's time to create your first Kotlin application.

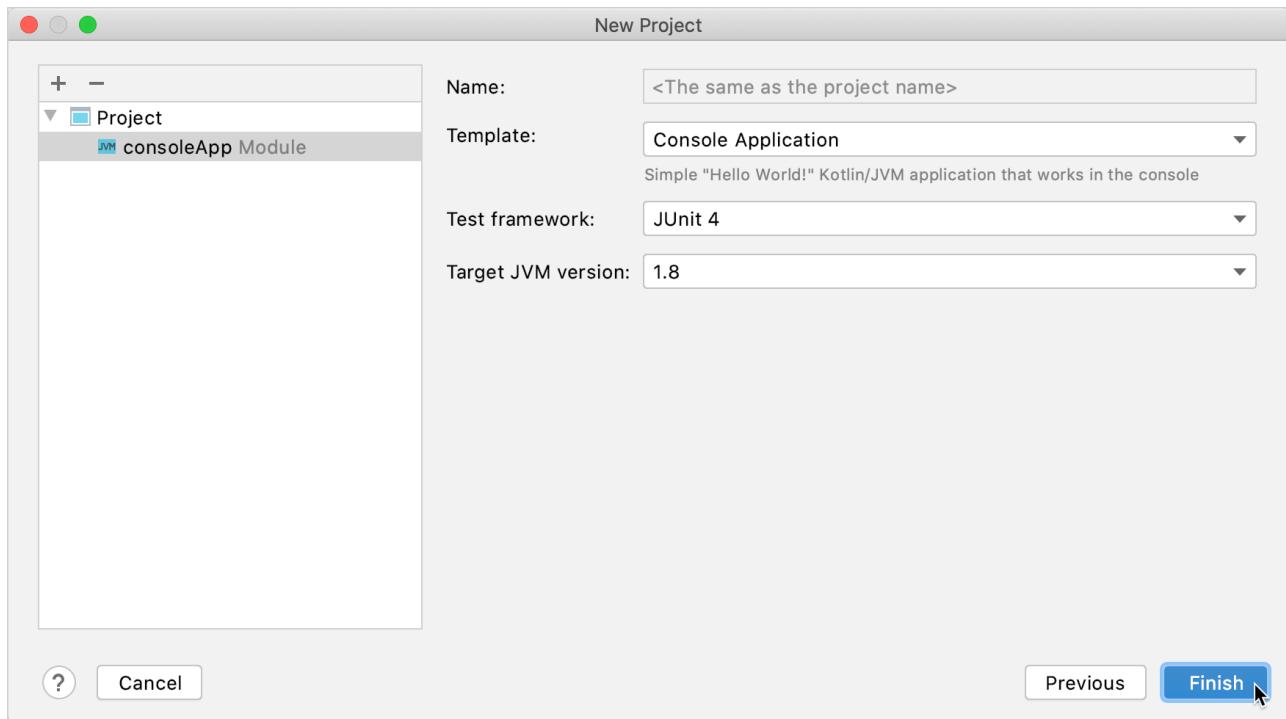
1. In IntelliJ IDEA, select File | New | Project.
2. In the panel on the left, select Kotlin.
3. Enter a project name, select Console Application as the project template, and click Next.



Create a console application

By default, your project will use the Gradle build system with Kotlin DSL.

4. Go through and accept the default configuration, then click Finish.

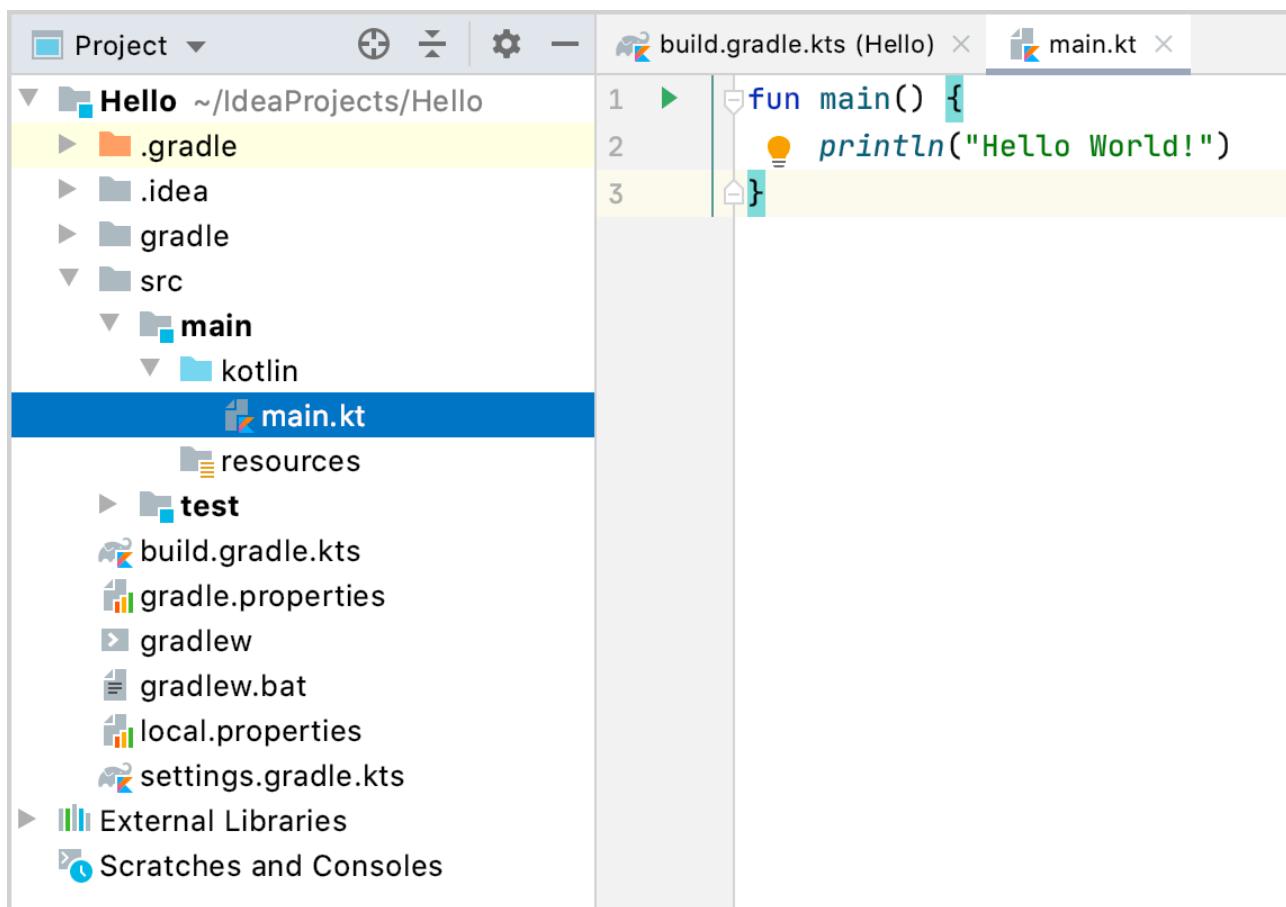


Configure a console application

Your project will open. By default, you see the file `build.gradle.kts`, which is the build script created by the Project Wizard based on your configuration. It includes the `kotlin("jvm")` plugin and dependencies required for your console application.

5. Open the `main.kt` file in `src/main/kotlin`.

The `src` directory contains Kotlin source files and resources. The `main.kt` file contains sample code that will print `Hello World!`.



main.kt with main fun

6. Modify the code so that it requests your name and says Hello to you specifically, and not to the whole world.

- Introduce a local variable name with the keyword val. It will get its value from an input where you will enter your name – readLine().
- Use a string template by adding a dollar sign \$ before this variable name directly in the text output like this – \$name.

```
fun main() {  
    println("What's your name?")  
    val name= readLine()  
    println("Hello $name!")  
}
```

```
build.gradle.kts (Hello) × main.kt ×
1 ► fun main() {
2     println("What's your name?")
3     val name= readLine()
4     println("Hello $name!")
5 }
```

Updated main fun

Run the application

Now the application is ready to run. The easiest way to do this is to click the green Run icon in the gutter and select Run 'MainKt'.

```
1 ► fun main() {
2     ▶ Run 'MainKt' ^♂R
3     ⛳ Debug 'MainKt' ^♂D
4     ▶ Run 'MainKt' with Coverage
5     ▶ Run 'MainKt' with 'CPU Profiler'
     ▶ Run 'MainKt' with 'Allocation Profiler'
     ▶ Run 'MainKt' with 'Java Flight Recorder'
     ▶ Edit 'MainKt'...
```

Running a console app

You can see the result in the Run tool window.

```
/Applications/IntelliJ IDEA.app/Contents/jbr/Contents/Home/bin/java
What's your name?
```

Kotlin run output

Enter your name and accept the greetings from your application!

```
/Applications/IntelliJ IDEA.app/Contents/jbr/Contents/Home/bin/java
What's your name?
Kate
Hello Kate!
Process finished with exit code 0
```

Kotlin run output

Congratulations! You have just run your first Kotlin application.

What's next?

Once you've created this application, you can start to dive deeper into Kotlin syntax:

- Add sample code from [Kotlin examples](#)
- Install the [EduTools plugin](#) for IDEA and complete exercises from the [Kotlin Koans course](#)

Test code using JUnit in JVM – tutorial

This tutorial will show you how to write a simple unit test and run it with the Gradle build tool.

The example in the tutorial has the `kotlin.test` library under the hood and runs the test using JUnit.

To get started, first download and install the latest version of [IntelliJ IDEA](#).

Add dependencies

1. Open a Kotlin project in IntelliJ IDEA. If you don't already have a project, [create one](#).

Specify JUnit 5 as your test framework when creating your project.

2. Open the `build.gradle(.kts)` file and add the following dependency to the Gradle configuration. This dependency will allow you to work with `kotlin.test` and JUnit:

Groovy

```
dependencies {  
    // Other dependencies.  
    testImplementation 'org.jetbrains.kotlin:kotlin-test'  
}
```

Kotlin

```
dependencies {  
    // Other dependencies.  
    testImplementation(kotlin("test"))  
}
```

3. Add the test task to the `build.gradle(.kts)` file:

Groovy

```
test {  
    useJUnitPlatform()  
}
```

Kotlin

```
tasks.test {  
    useJUnitPlatform()  
}
```

If you created the project using the Project Wizard, the task will be added automatically.

Add the code to test it

1. Open the main.kt file in src/main/kotlin.

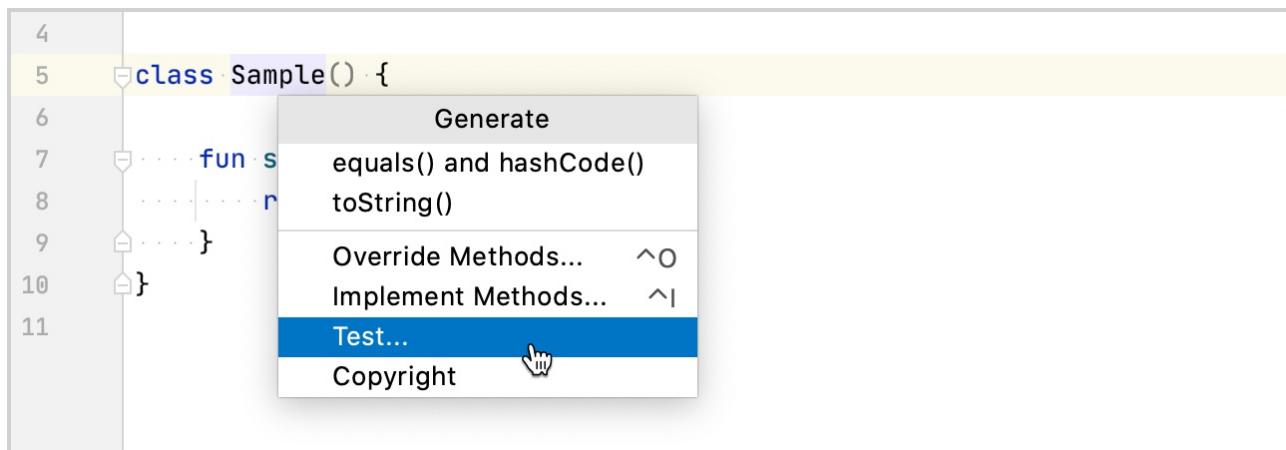
The src directory contains Kotlin source files and resources. The main.kt file contains sample code that will print Hello, World!.

2. Create the Sample class with the sum() function that adds two integers together:

```
class Sample() {  
  
    fun sum(a: Int, b: Int): Int {  
        return a + b  
    }  
}
```

Create a test

1. In IntelliJ IDEA, select Code | Generate | Test... for the Sample class.



Create a test

2. Specify the name of the test class. For example, SampleTest.

IntelliJ IDEA creates the SampleTest.kt file in the test directory. This directory contains Kotlin test source files and resources.

You can also manually create a *.kt file for tests in src/test/kotlin.

3. Add the test code for the sum() function in SampleTest.kt:

- Define the test testSum() function using the @Test annotation.
- Check that the sum() function returns the expected value by using the assertEquals() function.

```
import kotlin.test.Test
import kotlin.test.assertEquals

internal class SampleTest {

    private val testSample: Sample = Sample()

    @Test
    fun testSum() {
        val expected = 42
        assertEquals(expected, testSample.sum(40, 2))
    }
}
```

Run a test

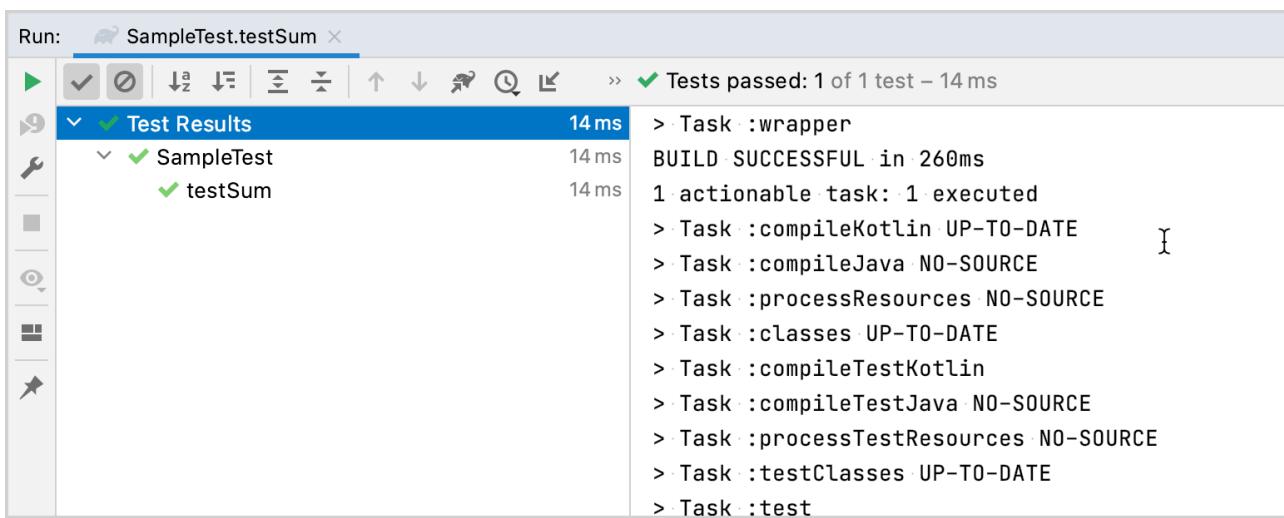
1. Run the test using the gutter icon.

```
7
8     ... @Test
9     ◀ fun testSum() {
10         ... val expected = 42
11         ... assertEquals(expected, testSample.sum(a: 40, b: 2))
12     }
13 }
```

Run the test

You can also run all project tests via the command-line interface using the ./gradlew check command.

2. Check the result in the Run tool window:



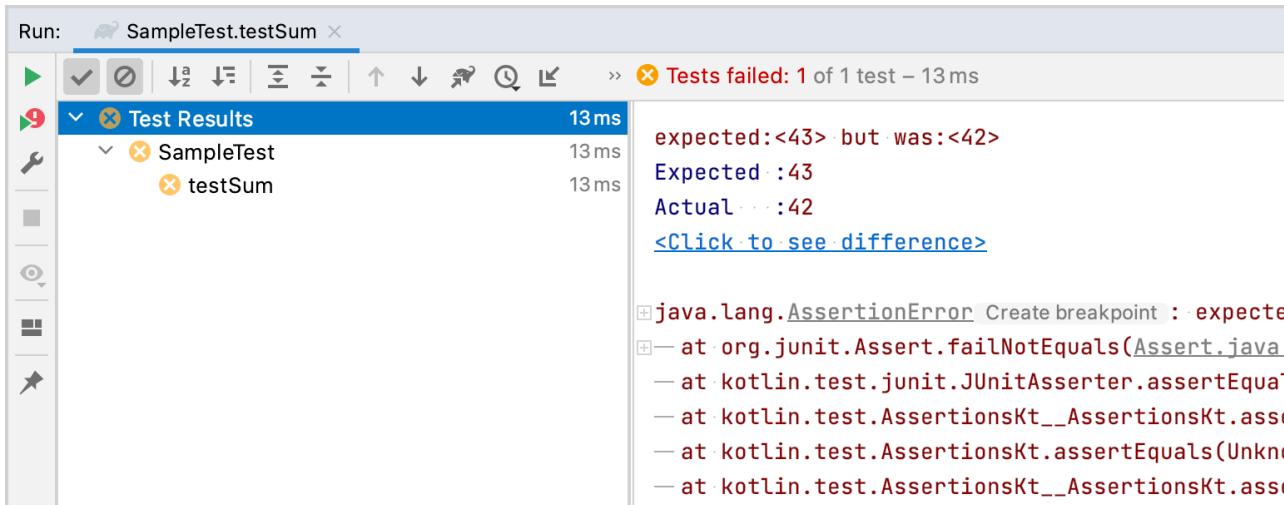
Check the test result. The test passed successfully

The test function was executed successfully.

3. Make sure that the test works correctly by changing the expected variable value to 43:

```
@Test
fun testSum() {
    val expected = 43
    assertEquals(expected, classForTesting.sum(40, 2))
}
```

4. Run the test again and check the result:



Check the test result. The test has been failed

The test execution failed.

What's next

Once you've finished your first test, you can:

- Try to write another test using other `kotlin.test` functions. For example, you could use the `assertNotEquals()` function.
- [Create your first application](#) with Kotlin and Spring Boot.
- Watch [these video tutorials](#) on YouTube, which demonstrate how to use Spring Boot with Kotlin and JUnit 5.

Create a RESTful web service with Spring Boot – tutorial

This tutorial walks you through the process of creating a simple application with Spring Boot.

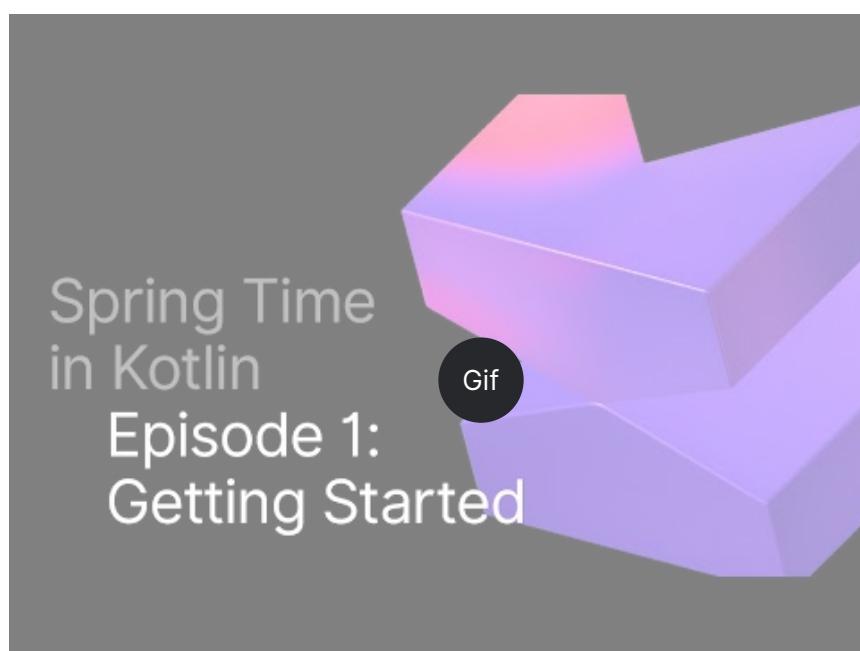
You will create an application with an HTTP endpoint that returns a data objects list in JSON format.

This tutorial consists of two parts:

- Create a RESTful Web Service with Spring Boot
- [Add a database to a Spring Boot RESTful web service](#)

To get started, first download and install the latest version of [IntelliJ IDEA](#).

You can also watch a video of this tutorial:



[Watch video online.](#)

Bootstrap the project

Use Spring Initializr to create a new project:

You can also create a new project using [IntelliJ IDEA with the Spring Boot plugin](#)

1. Open [Spring Initializr](#). This link opens the page with the project settings for this tutorial already filled in. This project uses Gradle, Kotlin, Spring Web, Spring Data JDBC, and H2 Database:

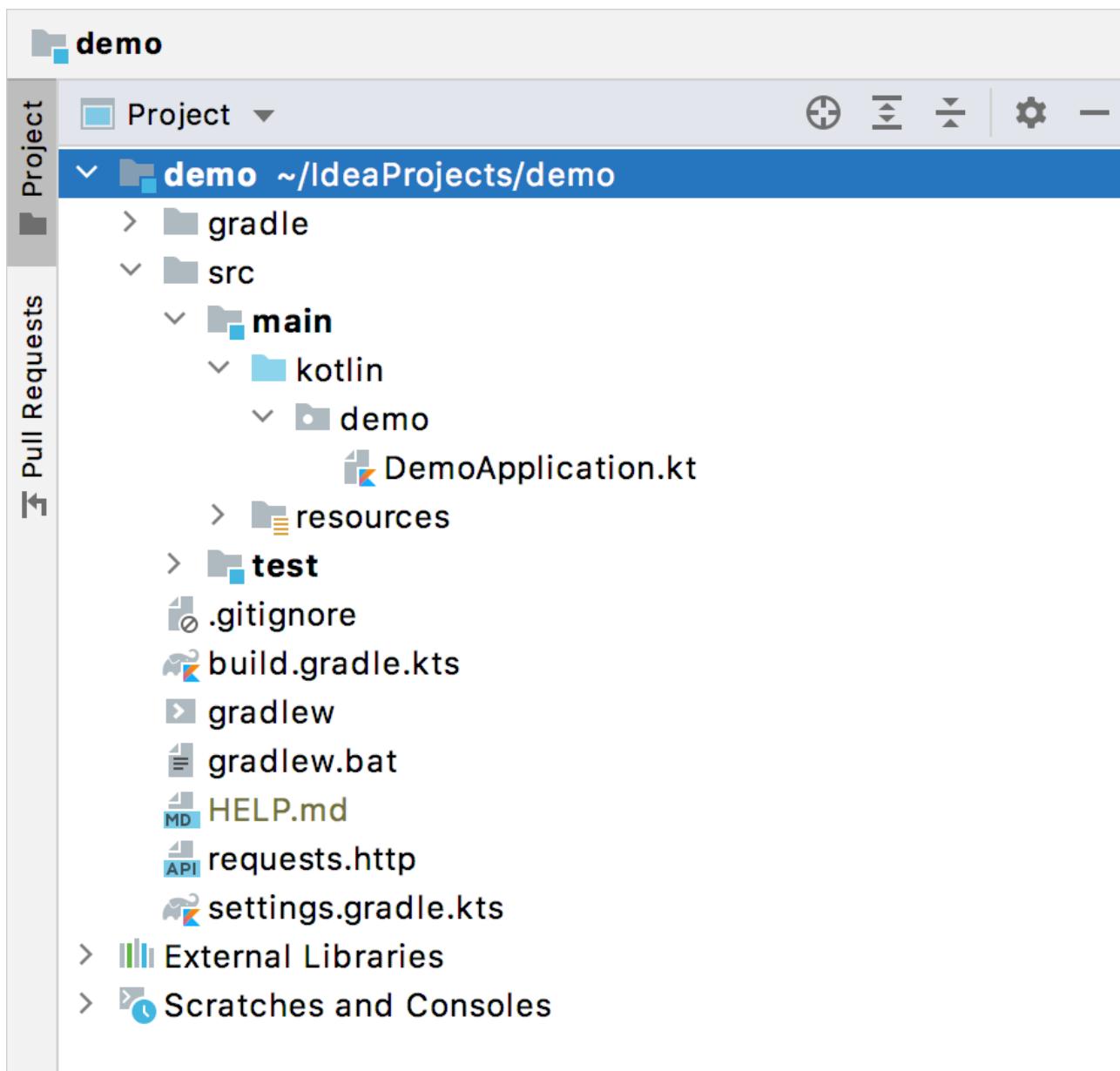
The screenshot shows the Spring Initializr web interface at start.spring.io. The interface is divided into several sections:

- Project**: Maven Project (radio button) selected, Gradle Project (radio button) is available.
- Language**: Java (radio button) selected, Kotlin (radio button) selected, Groovy (radio button) is available.
- Spring Boot**: Version 2.4.2 (radio button) selected, 2.5.0 (SNAPSHOT), 2.5.0 (M1), 2.4.3 (SNAPSHOT), 2.3.9 (SNAPSHOT), 2.3.8 are available.
- Project Metadata**: Group: com.example, Artifact: demo, Name: demo, Description: Demo project for Spring Boot, Package name: demo, Packaging: Jar (radio button) selected, War (radio button) is available. Java version: 15 (radio button), 11 (radio button) selected, 8 (radio button) is available.
- Dependencies**:
 - Spring Web** (WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
 - Spring Data JDBC** (SQL): Persist data in SQL stores with plain JDBC using Spring Data.
 - H2 Database** (SQL): Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.
- Buttons at the bottom**: GENERATE (⌘ + ↩), EXPLORE (CTRL + SPACE), SHARE...

Create a new project with Spring Initializr

2. Click GENERATE at the bottom of the screen. Spring Initializr will generate the project with the specified settings. The download starts automatically.
3. Unpack the .zip file and open it in IntelliJ IDEA.

The project has the following structure:



The Spring Boot project structure

There are packages and classes under the main/kotlin folder that belong to the application. The entry point to the application is the main() method of the DemoApplication.kt file.

Explore the project build file

Open the build.gradle.kts file.

This is the Gradle Kotlin build script, which contains a list of the dependencies required for the application.

The Gradle file is standard for Spring Boot, but it also contains necessary Kotlin dependencies, including the [kotlin-spring](#) Gradle plugin.

Explore the Spring Boot application

Open the DemoApplication.kt file:

```
package demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}
```

Note that the Kotlin application file differs from a Java application file:

- While Spring Boot looks for a public static main() method, the Kotlin application uses a [top-level function](#) defined outside DemoApplication class.
- The DemoApplication class is not declared as open, since the [kotlin-spring](#) plugin does that automatically.

Create a data class and a controller

To create an endpoint, add a [data class](#) and a controller to your project:

1. In the DemoApplication.kt file, create a Message data class with two properties: id and text:

```
data class Message(val id: String?, val text: String)
```

2. In the same file, create a MessageResource class which will serve the requests and return a JSON document containing a collection of Message objects:

```
@RestController
class MessageResource {
    @GetMapping
    fun index(): List<Message> = listOf(
        Message("1", "Hello!"),
        Message("2", "Bonjour!"),
        Message("3", "Privet!"),
    )
}
```

Full code of the DemoApplication.kt:

```
package demo
```

```

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.SpringApplication
import org.springframework.data.annotation.Id
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RestController

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}

@RestController
class MessageResource {
    @GetMapping
    fun index(): List<Message> = listOf(
        Message("1", "Hello!"),
        Message("2", "Bonjour!"),
        Message("3", "Privet!"),
    )
}

data class Message(val id: String?, val text: String)

```

Run the application

The application is now ready to run:

1. Click the green Run icon in the gutter beside the main() method or use the Alt+Enter shortcut to invoke the launch menu in IntelliJ IDEA:

```

package demo

...

```

```

    @SpringBootApplication
    class DemoApplication

    fun main(args: Array<String>) {
        runApplication<DemoApplication>()
    }

    @RestController
    class MessageResource {
        @GetMapping
        fun index(): List<Message> = listOf(
            Message(id: "1", text: "Hello!"),
            Message(id: "2", text: "Bonjour!"),
            Message(id: "1", text: "Privet!")
        )
    }

    data class Message(@Id val id: String?, val text: String)
}

```

The screenshot shows the IntelliJ IDEA interface with the code editor open to `DemoApplication.kt`. A context menu is displayed over the `main()` method, listing options such as 'Run', 'Debug', and 'Coverage'. The project structure on the left shows files like `gradle`, `build.gradle.kts`, and `src/main/kotlin/demo/DemoApplication.kt`. The bottom status bar indicates indexing completed in 23 seconds.

Run the application

You can also run the `./gradlew bootRun` command in the terminal.

- Once the application starts, open the following URL: <http://localhost:8080>.

You will see a page with a collection of messages in JSON format:



A screenshot of a web browser window titled "localhost:8080". The address bar also shows "localhost:8080". The main content area displays a JSON array with three objects. Each object has an "id" field and a "text" field. The first object has id "1" and text "Hello!". The second object has id "2" and text "Bonjour!". The third object has id "1" and text "Privet!". There are collapse arrows next to each object. At the bottom right of the JSON view, there are "Raw" and "Parsed" buttons.

```
[{"id": "1", "text": "Hello!"}, {"id": "2", "text": "Bonjour!"}, {"id": "1", "text": "Privet!"}]
```

Application output

Proceed to the next tutorial

In the next part of this tutorial you will add a database for storing objects, as well as two endpoints for writing and retrieving those objects: [Add a database to a Spring Boot RESTful web service](#).

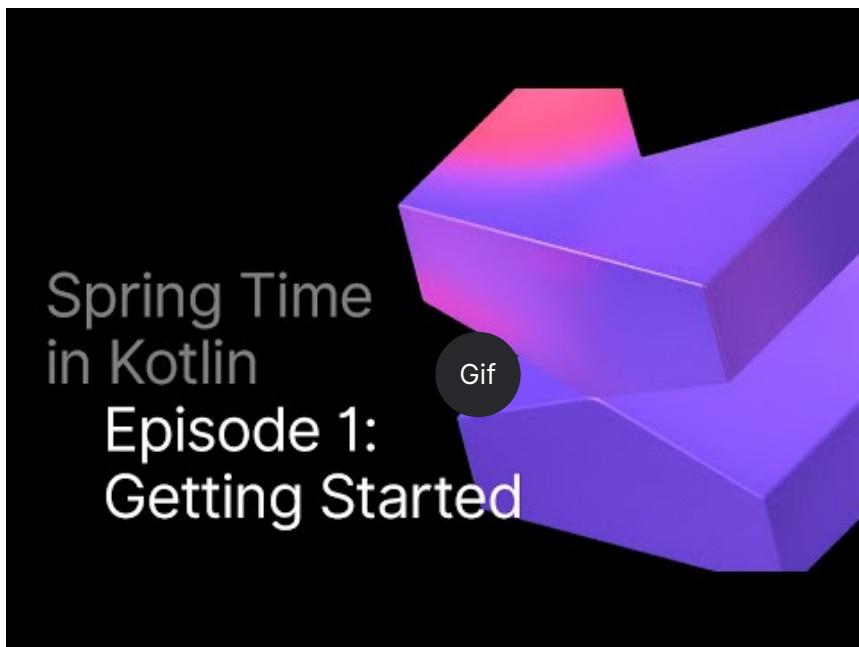
Add a database to a Spring Boot RESTful web service – tutorial

This tutorial walks you through the process of adding a database to a RESTful web service with Spring Boot and shows you how to work with it.

On completing this tutorial you'll have an application with a database for storing objects, as well as endpoints for writing and retrieving them.

This tutorial is a continuation of the previous tutorial – [Create a RESTful web service with Spring Boot](#). Complete the previous tutorial before proceeding with this one.

You can also watch a video of this tutorial (starting from 4:17):



[Watch video online.](#)

Add database support

In this section, you will create two endpoints: one for saving messages to the database, and one for retrieving them:

1. Add the `@Table` annotation to the `Message` class to declare mapping to a database table. Add the `@Id` annotation before the `id` field. These annotations also require additional imports:

```
import org.springframework.data.annotation.Id  
import org.springframework.data.relational.core.mapping.Table  
  
@Table("MESSAGES")  
data class Message(@Id val id: String?, val text: String)
```

2. Use the [Spring Data Repository API](#) to access the database:

```
import org.springframework.data.jdbc.repository.query.Query  
import org.springframework.data.repository.CrudRepository  
  
interface MessageRepository : CrudRepository<Message, String>{  
  
    @Query("select * from messages")  
    fun findMessages(): List<Message>  
}
```

When you call the `findMessages()` method on an instance of `MessageRepository`, it will execute the corresponding database query:

```
select * from messages
```

This query retrieves a list of all Message objects in the database table.

3. Create the MessageService class:

```
import org.springframework.stereotype.Service

@Service
class MessageService(val db: MessageRepository) {

    fun findMessages(): List<Message> = db.findMessages()

    fun post(message: Message){
        db.save(message)
    }
}
```

This class contains two methods:

- `post()` for writing a new Message object to the database
- `findMessages()` for getting all the messages from the database

4. Update the MessageResource class:

```
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.PostMapping

@RestController
class MessageResource(val service: MessageService) {
    @GetMapping
    fun index(): List<Message> = service.findMessages()

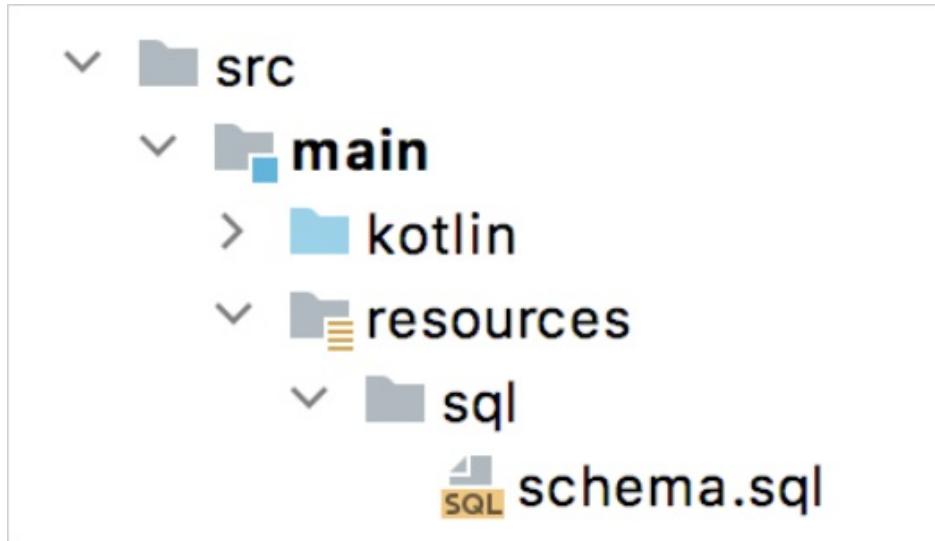
    @PostMapping
    fun post(@RequestBody message: Message) {
        service.post(message)
    }
}
```

Now it uses MessageService to work with the database.

Configure the database

Configure the database in the application:

1. Create a new folder called `sql` in the `src/main/resources` with the `schema.sql` file inside. It will store the database scheme:



Create a new folder

2. Update the src/main/resources/sql/schema.sql file with the following code:

```
CREATE TABLE IF NOT EXISTS messages (
    id          VARCHAR(60)  DEFAULT RANDOM_UUID() PRIMARY KEY,
    text        VARCHAR      NOT NULL
);
```

It creates the messages table with two fields: id and text. The table structure matches the structure of the Message class.

3. Open the application.properties file located in the src/main/resources folder and add the following application properties:

```
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:file:./data/testdb
spring.datasource.username=sa
spring.datasource.password=password
spring.datasource.schemaclasspath:sql/schema.sql
spring.datasource.initialization-mode=always
```

These settings enable the database for the Spring Boot application. See the full list of common application properties in the [Spring documentation](#).

Run the application

You should use an HTTP client to work with previously created endpoints. In IntelliJ IDEA, you can use the embedded [HTTP client](#):

1. [Run the application](#). Once the application is up and running, you can execute POST requests to store messages in the database.

2. Create the requests.http file and add the following HTTP requests:

```
### Post 'Hello!'
POST http://localhost:8080/
Content-Type: application/json

{
  "text": "Hello!"
}

### Post "Bonjour!"

POST http://localhost:8080/
Content-Type: application/json

{
  "text": "Bonjour!"
}

### Post "Privet!"

POST http://localhost:8080/
Content-Type: application/json

{
  "text": "Privet!"
}

### Get all the messages
GET http://localhost:8080/
```

3. Execute all POST requests. Use the green Run icon in the gutter next to the request declaration. These requests write the text messages to the database.

```

1  ### Post "Hello!"
2  POST http://localhost:8080
3  Content-Type: application/json
4
5  {
6      "text": "Hello!"
7  }
8
9  ### Post "Bonjour!"
10 POST http://localhost:8080
11 Content-Type: application/json
12 {
13     "text": "Bonjour!"
14 }
15
16
17  ### Post "Privet!"
18 POST http://localhost:8080
19 Content-Type: application/json
20
21 {
22     "text": "Privet!"
23 }
24
25
26  ### get all the messages
27 GET http://localhost:8080/

```

Run HTTP POST requests

4. Execute the GET request and see the result in the Run tool window:

```

1  ### get all the messages
2  GET http://localhost:8080/

```

HTTP/1.1 200
Content-Type: application/json
Transfer-Encoding: chunked
Date: Wed, 27 Jan 2021 14:35:21 GMT
Keep-Alive: timeout=60
Connection: keep-alive

```

[
  {
    "id": "5bbcfcc7-bf51-40a9-a35e-549e6ad61320",
    "text": "Hello!"
  },
  {
    "id": "fcf89b37-c98a-4a33-99f7-4d71c18b761f",
    "text": "Bonjour!"
  },
  {
    "id": "7738d1f1-03bf-4e4f-8fee-db3703028935",
    "text": "Privet!"
  }
]

```

Run HTTP GET request

Alternative way to execute requests

You can also use any other HTTP client or cURL command-line tool. For example, you can run the following commands in the terminal to get the same result:

```
curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json" -d "{\"text\": \"Hello!\" }"  
curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json" -d "{\"text\": \"Bonjour!\" }"  
curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json" -d "{\"text\": \"Privet!\" }"  
curl -X GET --location "http://localhost:8080"
```

What's next?

If you encountered any difficulties while following this tutorial, you can download and explore the [completed project](#).

Records

Records are [classes](#) in Java for storing immutable data. Records carry a fixed set of values – the records components. They have a concise syntax in Java and save you from having to write boilerplate code:

```
// Java  
public record Person (String name, int age) {}
```

The compiler automatically generates a final class inherited from [java.lang.Record](#) with the following members:

- a private final field for each record component
- a public constructor with parameters for all fields
- a set of methods to implement structural equality: equals(), hashCode(), toString()
- a public method for reading each record component

Records are very similar to Kotlin [data classes](#).

Using Java records from Kotlin code

You can use record classes with components that are declared in Java the same way you would use classes with properties in Kotlin. To access the record component, just use its name like you do for [Kotlin properties](#):

```
val firstName = Person.name
```

Declare records in Kotlin

Kotlin supports record declaration only for data classes, and the data class must meet the [requirements](#).

To declare a record class in Kotlin, use the `@JvmRecord` annotation:

Applying `@JvmRecord` to an existing class is not a binary compatible change. It alters the naming convention of the class property accessors.

```
@JvmRecord  
data class Person(val name: String, val age: Int)
```

This JVM-specific annotation enables generating:

- the record components corresponding to the class properties in the class file
- the property accessor methods named according to the Java record naming convention

The data class provides `equals()`, `hashCode()`, and `toString()` method implementations.

Requirements

To declare a data class with the `@JvmRecord` annotation, it must meet the following requirements:

- The class must be in a module that targets JVM 16 bytecode (or 15 if the `-Xjvm-enable-preview` compiler option is enabled).
- The class cannot explicitly inherit any other class (including `Any`) because all JVM records implicitly inherit `java.lang.Record`. However, the class can implement interfaces.
- The class cannot declare any properties with backing fields – except those initialized from the corresponding primary constructor parameters.
- The class cannot declare any mutable properties with backing fields.
- The class cannot be local.
- The primary constructor of the class must be as visible as the class itself.

Enabling JVM records

JVM records require the 16 target version or higher of the generated JVM bytecode.

To specify it explicitly, use the `jvmTarget` compiler option in [Gradle](#) or [Maven](#).

Further discussion

See this [language proposal for JVM records](#) for further technical details and discussion.

Calling Java from Kotlin

Kotlin is designed with Java interoperability in mind. Existing Java code can be called from Kotlin in a natural way, and Kotlin code can be used from Java rather smoothly as well. In this section, we describe some details about calling Java code from Kotlin.

Pretty much all Java code can be used without any issues:

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 'for'-loops work for Java collections:
    for (item in source) {
        list.add(item)
    }
    // Operator conventions work as well:
    for (i in 0..source.size - 1) {
        list[i] = source[i] // get and set are called
    }
}
```

Getters and setters

Methods that follow the Java conventions for getters and setters (no-argument methods with names starting with `get` and single-argument methods with names starting with `set`) are represented as properties in Kotlin. Boolean accessor methods (where the name of the getter starts with `is` and the name of the setter starts with `set`) are represented as properties which have the same name as the getter method.

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) { // call getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY // call setFirstDayOfWeek()
    }
}
```

```
    if (!calendar.isLenient) { // call isLenient()
        calendar.isLenient = true // call setLenient()
    }
}
```

Note that, if the Java class only has a setter, it isn't visible as a property in Kotlin because Kotlin doesn't support set-only properties.

Methods returning void

If a Java method returns void, it will return Unit when called from Kotlin. If by any chance someone uses that return value, it will be assigned at the call site by the Kotlin compiler since the value itself is known in advance (being Unit).

Escaping for Java identifiers that are keywords in Kotlin

Some of the Kotlin keywords are valid identifiers in Java: in, object, is, and other. If a Java library uses a Kotlin keyword for a method, you can still call the method escaping it with the backtick () character:

```
foo.`is`(bar)
```

Null-safety and platform types

Any reference in Java may be null, which makes Kotlin's requirements of strict null-safety impractical for objects coming from Java. Types of Java declarations are treated in Kotlin in a specific manner and called platform types. Null-checks are relaxed for such types, so that safety guarantees for them are the same as in Java (see more [below](#)).

Consider the following examples:

```
val list = ArrayList<String>() // non-null (constructor result)
list.add("Item")
val size = list.size // non-null (primitive int)
val item = list[0] // platform type inferred (ordinary Java object)
```

When you call methods on variables of platform types, Kotlin does not issue nullability errors at compile time, but the call may fail at runtime, because of a null-pointer exception or an assertion that Kotlin generates to prevent nulls from propagating:

```
item.substring(1) // allowed, may throw an exception if item == null
```

Platform types are non-denotable, meaning that you can't write them down explicitly in the language. When a platform value is assigned to a Kotlin variable, you can rely on the type inference (the variable will have an inferred platform type then, as item has in the example above), or you can choose the type you expect (both nullable and non-null types are

allowed):

```
val nullable: String? = item // allowed, always works
val notNull: String = item // allowed, may fail at runtime
```

If you choose a non-null type, the compiler will emit an assertion upon assignment. This prevents Kotlin's non-null variables from holding nulls. Assertions are also emitted when you pass platform values to Kotlin functions expecting non-null values and in other cases. Overall, the compiler does its best to prevent nulls from propagating far through the program although sometimes this is impossible to eliminate entirely, because of generics.

Notation for platform types

As mentioned above, platform types can't be mentioned explicitly in the program, so there's no syntax for them in the language. Nevertheless, the compiler and IDE need to display them sometimes (for example, in error messages or parameter info), so there is a mnemonic notation for them:

- T! means "T or T? ",
- (Mutable)Collection<T>! means "Java collection of T may be mutable or not, may be nullable or not",
- Array<(out) T>! means "Java array of T (or a subtype of T), nullable or not"

Nullability annotations

Java types that have nullability annotations are represented not as platform types, but as actual nullable or non-null Kotlin types. The compiler supports several flavors of nullability annotations, including:

- [JetBrains](#) (@Nullable and @NotNull from the org.jetbrains.annotations package)
- [JSpecify](#) (org.jspecify.nullness)
- Android (com.android.annotations and android.support.annotations)
- JSR-305 (javax.annotation, more details below)
- FindBugs (edu.umd.cs.findbugs.annotations)
- Eclipse (org.eclipse.jdt.annotation)
- Lombok (lombok.NonNull)

You can find the full list in the [Kotlin compiler source code](#).

Annotating type arguments and type parameters

You can annotate the type arguments and type parameters of generic types to provide nullability information for them as well.

All examples in the section use JetBrains nullability annotations from the org.jetbrains.annotations package.

Type arguments

Consider these annotations on a Java declaration:

```
@NotNull  
Set<@NotNull String> toSet(@NotNull Collection<@NotNull String> elements) { ... }
```

They result in the following signature in Kotlin:

```
fun toSet(elements: (Mutable)Collection<String>) : (Mutable)Set<String> { ... }
```

When the @NotNull annotation is missing from a type argument, you get a platform type instead:

```
fun toSet(elements: (Mutable)Collection<String!>) : (Mutable)Set<String!> { ... }
```

Kotlin also takes into account nullability annotations on type arguments of base classes and interfaces. For example, there are two Java classes with the signatures provided below:

```
public class Base<T> {}
```

```
public class Derived extends Base<@Nullable String> {}
```

In the Kotlin code, passing the instance of Derived where the Base<String> is assumed produces the warning.

```
fun takeBaseOfNotNullStrings(x: Base<String>) {}

fun main() {
    takeBaseOfNotNullStrings(Derived()) // warning: nullability mismatch
}
```

The upper bound of Derived is set to Base<String?>, which is different from Base<String>.

Learn more about [Java generics in Kotlin](#).

Type parameters

By default, the nullability of plain type parameters in both Kotlin and Java is undefined. In Java, you can specify it using nullability annotations. Let's annotate the type parameter of the Base class:

```
public class Base<@NotNull T> {}
```

When inheriting from Base, Kotlin expects a non-null type argument or type parameter. Thus, the following Kotlin code produces a warning:

```
class Derived<K> : Base<K> {} // warning: K has undefined nullability
```

You can fix it by specifying the upper bound K : Any.

Kotlin also supports nullability annotations on the bounds of Java type parameters. Let's add bounds to Base:

```
public class BaseWithBound<T extends @NotNull Number> {}
```

Kotlin translates this just as follows:

```
class BaseWithBound<T : Number> {}
```

So passing nullable type as a type argument or type parameter produces a warning.

Annotating type arguments and type parameters works with the Java 8 target or higher. The feature requires that the nullability annotations support the TYPE_USE target (org.jetbrains.annotations supports this in version 15 and above). Pass the -Xtype-enhancement-improvements-strict-mode compiler option to report errors in Kotlin code that uses nullability which deviates from the nullability annotations from Java.

Note: If a nullability annotation supports other targets that are applicable to a type in addition to the TYPE_USE target, then TYPE_USE takes priority. For example, if @Nullable has both TYPE_USE and METHOD targets, the Java method signature @Nullable String[] f() becomes fun f(): Array<String?>! in Kotlin.

JSR-305 support

The [@Nonnull](#) annotation defined in [JSR-305](#) is supported for denoting nullability of Java types.

If the `@Nonnull(when = ...)` value is `When.ALWAYS`, the annotated type is treated as non-null; `When.MAYBE` and `When.NEVER` denote a nullable type; and `When.UNKNOWN` forces the type to be [platform one](#).

A library can be compiled against the JSR-305 annotations, but there's no need to make the annotations artifact (e.g. `jsr305.jar`) a compile dependency for the library consumers. The Kotlin compiler can read the JSR-305 annotations from a library without the annotations present on the classpath.

[Custom nullability qualifiers \(KEEP-79\)](#) are also supported (see below).

Type qualifier nicknames

If an annotation type is annotated with both [@TypeQualifierNickname](#) and JSR-305 `@Nonnull` (or its another nickname, such as `@CheckForNull`), then the annotation type is itself used for retrieving precise nullability and has the same meaning as that nullability annotation:

```

@TypeQualifierNickname
@Nonnull(when = When.ALWAYS)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNonnull {
}

@TypeQualifierNickname
@CheckForNull // a nickname to another type qualifier nickname
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNullable {
}

interface A {
    @MyNullable String foo(@Nonnull String x);
    // in Kotlin (strict mode): `fun foo(x: String): String?`

    String bar(List<@Nonnull String> x);
    // in Kotlin (strict mode): `fun bar(x: List<String>!): String!`
}

```

Type qualifier defaults

`@TypeQualifierDefault` allows introducing annotations that, when being applied, define the default nullability within the scope of the annotated element.

Such annotation type should itself be annotated with both `@Nonnull` (or its nickname) and `@TypeQualifierDefault(...)` with one or more `ElementType` values:

- `ElementType.METHOD` for return types of methods
- `ElementType.PARAMETER` for value parameters
- `ElementType.FIELD` for fields
- `ElementType.TYPE_USE` for any type including type arguments, upper bounds of type parameters and wildcard types

The default nullability is used when a type itself is not annotated by a nullability annotation, and the default is determined by the innermost enclosing element annotated with a type qualifier default annotation with the `ElementType` matching the type usage.

```

@Nonnull
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
public @interface NonNullApi {

}

@Nonnull(when = When.MAYBE)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE_USE})
public @interface NullableApi {

}

@interface A {

```

```

String foo(String x); // fun foo(x: String?): String?

@NotNullApi // overriding default from the interface
String bar(String x, @Nullable String y); // fun bar(x: String, y: String?): String

// The List<String> type argument is seen as nullable because of `@NullableApi`
// having the `TYPE_USE` element type:
String baz(List<String> x); // fun baz(List<String?>?): String?

// The type of `x` parameter remains platform because there's an explicit
// UNKNOWN-marked nullability annotation:
String qux(@Nonnull(when = When.UNKNOWN) String x); // fun baz(x: String!): String?
}

```

The types in this example only take place with the strict mode enabled; otherwise, the platform types remain. See the [@UnderMigration annotation](#) and [Compiler configuration](#) sections.

Package-level default nullability is also supported:

```

// FILE: test/package-info.java
@NotNullApi // declaring all types in package 'test' as non-nullable by default
package test;

```

@UnderMigration annotation

The `@UnderMigration` annotation (provided in a separate artifact `kotlin-annotations-jvm`) can be used by library maintainers to define the migration status for the nullability type qualifiers.

The status value in `@UnderMigration(status = ...)` specifies how the compiler treats inappropriate usages of the annotated types in Kotlin (e.g. using a `@MyNullable`-annotated type value as non-null):

- `MigrationStatus.STRICT` makes annotation work as any plain nullability annotation, i.e. report errors for the inappropriate usages and affect the types in the annotated declarations as they are seen in Kotlin
- `MigrationStatus.WARN`: the inappropriate usages are reported as compilation warnings instead of errors, but the types in the annotated declarations remain platform
- `MigrationStatus.IGNORE` makes the compiler ignore the nullability annotation completely

A library maintainer can add `@UnderMigration` status to both type qualifier nicknames and type qualifier defaults:

```

@Nonnull(when = When.ALWAYS)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
@UnderMigration(status = MigrationStatus.WARN)
public @interface NonNullApi {
}

// The types in the class are non-null, but only warnings are reported
// because `@NotNullApi` is annotated `@UnderMigration(status = MigrationStatus.WARN)`

```

```
@NonNullApi  
public class Test {}
```

The migration status of a nullability annotation is not inherited by its type qualifier nicknames but is applied to its usages in default type qualifiers.

If a default type qualifier uses a type qualifier nickname and they are both @UnderMigration, the status from the default type qualifier is used.

Compiler configuration

The JSR-305 checks can be configured by adding the -Xjsr305 compiler flag with the following options (and their combination):

- -Xjsr305={strict|warn|ignore} to set up the behavior for non- @UnderMigration annotations. Custom nullability qualifiers, especially @TypeQualifierDefault, are already spread among many well-known libraries, and users may need to migrate smoothly when updating to the Kotlin version containing JSR-305 support. Since Kotlin 1.1.60, this flag only affects non- @UnderMigration annotations.
- -Xjsr305=under-migration:{strict|warn|ignore} to override the behavior for the @UnderMigration annotations. Users may have different view on the migration status for the libraries: they may want to have errors while the official migration status is WARN, or vice versa, they may wish to postpone errors reporting for some until they complete their migration.
- -Xjsr305=@<fq.name>:{strict|warn|ignore} to override the behavior for a single annotation, where <fq.name> is the fully qualified class name of the annotation. May appear several times for different annotations. This is useful for managing the migration state for a particular library.

The strict, warn and ignore values have the same meaning as those of MigrationStatus, and only the strict mode affects the types in the annotated declarations as they are seen in Kotlin.

Note: the built-in JSR-305 annotations `@Nonnull`, `@Nullable` and `@CheckForNull` are always enabled and affect the types of the annotated declarations in Kotlin, regardless of compiler configuration with the -Xjsr305 flag.

For example, adding -Xjsr305=ignore -Xjsr305=under-migration:ignore -Xjsr305=@org.library.MyNullable:warn to the compiler arguments makes the compiler generate warnings for inappropriate usages of types annotated by `@org.library.MyNullable` and ignore all other JSR-305 annotations.

The default behavior is the same to -Xjsr305=warn. The strict value should be considered experimental (more checks may be added to it in the future).

Mapped types

Kotlin treats some Java types specifically. Such types are not loaded from Java "as is", but are mapped to corresponding Kotlin types. The mapping only matters at compile time, the runtime representation remains unchanged. Java's primitive types are mapped to corresponding Kotlin types (keeping [platform types](#) in mind):

Java type	Kotlin type
byte	kotlin.Byte
short	kotlin.Short
int	kotlin.Int
long	kotlin.Long
char	kotlin.Char
float	kotlin.Float
double	kotlin.Double
boolean	kotlin.Boolean

Some non-primitive built-in classes are also mapped:

Java type	Kotlin type
java.lang.Object	kotlin.Any!
java.lang.Cloneable	kotlin.Cloneable!
java.lang.Comparable	kotlin.Comparable!
java.lang.Enum	kotlin.Enum!
java.lang.Annotation	kotlin.Annotation!
java.lang(CharSequence	kotlin.CharSequence!
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!

Java type	Kotlin type
------------------	--------------------

java.lang.Throwable kotlin.Throwable!

Java's boxed primitive types are mapped to nullable Kotlin types:

Java type	Kotlin type
------------------	--------------------

java.lang.Byte kotlin.Byte?

java.lang.Short kotlin.Short?

java.lang.Integer kotlin.Int?

java.lang.Long kotlin.Long?

java.lang.Character kotlin.Char?

java.lang.Float kotlin.Float?

java.lang.Double kotlin.Double?

java.lang.Boolean kotlin.Boolean?

Note that a boxed primitive type used as a type parameter is mapped to a platform type: for example, `List<java.lang.Integer>` becomes a `List<Int!>` in Kotlin.

Collection types may be read-only or mutable in Kotlin, so Java's collections are mapped as follows (all Kotlin types in this table reside in the package `kotlin.collections`):

Java type	Kotlin read-only type	Kotlin mutable type	Loaded platform type
<code>Iterator<T></code>	<code>Iterator<T></code>	<code>MutableIterator<T></code>	<code>(Mutable)Iterator<T>!</code>
<code>Iterable<T></code>	<code>Iterable<T></code>	<code>MutableIterable<T></code>	<code>(Mutable)Iterable<T>!</code>
<code>Collection<T></code>	<code>Collection<T></code>	<code>MutableCollection<T></code>	<code>(Mutable)Collection<T>!</code>
<code>Set<T></code>	<code>Set<T></code>	<code>MutableSet<T></code>	<code>(Mutable)Set<T>!</code>
<code>List<T></code>	<code>List<T></code>	<code>MutableList<T></code>	<code>(Mutable)List<T>!</code>

Java type	Kotlin read-only type	Kotlin mutable type	Loaded platform type
ListIterator<T>	ListIterator<T>	MutableListIterator<T>	(Mutable)ListIterator<T>!
Map<K, V>	Map<K, V>	MutableMap<K, V>	(Mutable)Map<K, V>!
Map.Entry<K, V>	Map.Entry<K, V>	MutableMap.MutableEntry<K,V>	(Mutable)Map.(Mutable)Entry<K, V>!

Java's arrays are mapped as mentioned [below](#):

Java type Kotlin type

int[] kotlin.IntArray!

String[] kotlin.Array<(out) String>!

The static members of these Java types are not directly accessible on the [companion objects](#) of the Kotlin types. To call them, use the full qualified names of the Java types, e.g. java.lang.Integer.toHexString(foo).

Java generics in Kotlin

Kotlin's generics are a little different from Java's ([see Generics](#)). When importing Java types to Kotlin, the following conversions are done:

- Java's wildcards are converted into type projections:
 - Foo<? extends Bar> becomes Foo<out Bar!>!
 - Foo<? super Bar> becomes Foo<in Bar!>!
- Java's raw types are converted into star projections:
 - List becomes List<*>! that is List<out Any?>!

Like Java's, Kotlin's generics are not retained at runtime: objects do not carry information about actual type arguments passed to their constructors. For example, `ArrayList<Integer>()` is indistinguishable from `ArrayList<Character>()`. This makes it impossible to perform `is`-checks that take generics into account. Kotlin only allows `is`-checks for star-projected generic types:

```
if (a is List<Int>) // Error: cannot check if it is really a List of Ints
// but
if (a is List<*>) // OK: no guarantees about the contents of the list
```

Java arrays

Arrays in Kotlin are invariant, unlike Java. This means that Kotlin won't let you assign an `Array<String>` to an `Array<Any>`, which prevents a possible runtime failure. Passing an array of a subclass as an array of superclass to a Kotlin method is also prohibited, but for Java methods this is allowed through platform types of the form `Array<(out) String>!`.

Arrays are used with primitive datatypes on the Java platform to avoid the cost of boxing/unboxing operations. As Kotlin hides those implementation details, a workaround is required to interface with Java code. There are specialized classes for every type of primitive array (`IntArray`, `DoubleArray`, `CharArray`, and so on) to handle this case. They are not related to the `Array` class and are compiled down to Java's primitive arrays for maximum performance.

Suppose there is a Java method that accepts an int array of indices:

```
public class JavaArrayExample {  
    public void removeIndices(int[] indices) {  
        // code here...  
    }  
}
```

To pass an array of primitive values, you can do the following in Kotlin:

```
val javaObj = JavaArrayExample()  
val array = intArrayOf(0, 1, 2, 3)  
javaObj.removeIndices(array) // passes int[] to method
```

When compiling to the JVM bytecode, the compiler optimizes access to arrays so that there's no overhead introduced:

```
val array = arrayOf(1, 2, 3, 4)  
array[1] = array[1] * 2 // no actual calls to get() and set() generated  
for (x in array) { // no iterator created  
    print(x)  
}
```

Even when you navigate with an index, it does not introduce any overhead:

```
for (i in array.indices) { // no iterator created  
    array[i] += 2  
}
```

Finally, in -checks have no overhead either:

```
if (i in array.indices) { // same as (i >= 0 && i < array.size)  
    print(array[i])  
}
```

Java varargs

Java classes sometimes use a method declaration for the indices with a variable number of arguments (varargs):

```
public class JavaArrayExample {  
  
    public void removeIndicesVarArg(int... indices) {  
        // code here...  
    }  
}
```

In that case you need to use the spread operator * to pass the IntArray:

```
val javaObj = JavaArrayExample()  
val array = intArrayOf(0, 1, 2, 3)  
javaObj.removeIndicesVarArg(*array)
```

Operators

Since Java has no way of marking methods for which it makes sense to use the operator syntax, Kotlin allows using any Java methods with the right name and signature as operator overloads and other conventions (invoke() etc.) Calling Java methods using the infix call syntax is not allowed.

Checked exceptions

In Kotlin, all exceptions are unchecked, meaning that the compiler does not force you to catch any of them. So, when you call a Java method that declares a checked exception, Kotlin does not force you to do anything:

```
fun render(list: List<*>, to: Appendable) {  
    for (item in list) {  
        to.append(item.toString()) // Java would require us to catch IOException here  
    }  
}
```

Object methods

When Java types are imported into Kotlin, all the references of the type `java.lang.Object` are turned into `Any`. Since `Any` is not platform-specific, it only declares `toString()`, `hashCode()` and `equals()` as its members, so to make other members of `java.lang.Object` available, Kotlin uses extension functions.

wait()/notify()

Methods `wait()` and `notify()` are not available on references of type `Any`. Their usage is generally discouraged in favor of

`java.util.concurrent`. If you really need to call these methods, you can cast to `java.lang.Object`:

```
(foo as java.lang.Object).wait()
```

getClass()

To retrieve the Java class of an object, use the `java` extension property on a [class reference](#):

```
val fooClass = foo::class.java
```

The code above uses a [bound class reference](#). You can also use the `javaClass` extension property:

```
val fooClass = foo.javaClass
```

clone()

To override `clone()`, your class needs to extend `kotlin.Cloneable`:

```
class Example : Cloneable {
    override fun clone(): Any { ... }
}
```

Don't forget about [Effective Java, 3rd Edition](#), Item 13: Override `clone` judiciously.

finalize()

To override `finalize()`, all you need to do is simply declare it, without using the `override` keyword:

```
class C {
    protected fun finalize() {
        // finalization logic
    }
}
```

According to Java's rules, `finalize()` must not be private.

Inheritance from Java classes

At most one Java class (and as many Java interfaces as you like) can be a supertype for a class in Kotlin.

Accessing static members

Static members of Java classes form "companion objects" for these classes. You can't pass such a "companion object" around as a value but can access the members explicitly, for example:

```
if (Character.isLetter(a)) { ... }
```

To access static members of a Java type that is mapped to a Kotlin type, use the full qualified name of the Java type: java.lang.Integer.bitCount(foo).

Java reflection

Java reflection works on Kotlin classes and vice versa. As mentioned above, you can use instance::class.java, ClassName::class.java or instance.javaClass to enter Java reflection through java.lang.Class. Do not use ClassName.javaClass for this purpose because it refers to ClassName's companion object class, which is the same as ClassName.Companion::class.java and not ClassName::class.java.

For each primitive type, there are two different Java classes, and Kotlin provides ways to get both. For example, Int::class.java will return the class instance representing the primitive type itself, corresponding to Integer.TYPE in Java. To get the class of the corresponding wrapper type, use Int::class.javaObjectType, which is equivalent of Java's Integer.class.

Other supported cases include acquiring a Java getter/setter method or a backing field for a Kotlin property, a KProperty for a Java field, a Java method or constructor for a KFunction and vice versa.

SAM conversions

Kotlin supports SAM conversions for both Java and [Kotlin interfaces](#). This support for Java means that Kotlin function literals can be automatically converted into implementations of Java interfaces with a single non-default method, as long as the parameter types of the interface method match the parameter types of the Kotlin function.

You can use this for creating instances of SAM interfaces:

```
val runnable = Runnable { println("This runs in a runnable") }
```

...and in method calls:

```
val executor = ThreadPoolExecutor()
// Java signature: void execute(Runnable command)
executor.execute { println("This runs in a thread pool") }
```

If the Java class has multiple methods taking functional interfaces, you can choose the one you need to call by using an adapter function that converts a lambda to a specific SAM type. Those adapter functions are also generated by the compiler when needed:

```
executor.execute(Runnable { println("This runs in a thread pool") })
```

SAM conversions only work for interfaces, not for abstract classes, even if those also have just a single abstract method.

Using JNI with Kotlin

To declare a function that is implemented in native (C or C++) code, you need to mark it with the `external` modifier:

```
external fun foo(x: Int): Double
```

The rest of the procedure works in exactly the same way as in Java.

You can also mark property getters and setters as `external`:

```
var myProperty: String
    external get
    external set
```

Behind the scenes, this will create two functions `getMyProperty` and `setMyProperty`, both marked as `external`.

Using Lombok-generated declarations in Kotlin

You can use Java's Lombok-generated declarations in Kotlin code. If you need to generate and use these declarations in the same mixed Java/Kotlin module, you can learn how to do this on the [Lombok compiler plugin's page](#). If you call such declarations from another module, then you don't need to use this plugin to compile that module.

Calling Kotlin from Java

Kotlin code can be easily called from Java. For example, instances of a Kotlin class can be seamlessly created and operated in Java methods. However, there are certain differences between Java and Kotlin that require attention when integrating Kotlin code into Java. On this page, we'll describe the ways to tailor the interop of your Kotlin code with its Java clients.

Properties

A Kotlin property is compiled to the following Java elements:

- a getter method, with the name calculated by prepending the get prefix
- a setter method, with the name calculated by prepending the set prefix (only for var properties)
- a private field, with the same name as the property name (only for properties with backing fields)

For example, var firstName: String compiles to the following Java declarations:

```
private String firstName;

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

If the name of the property starts with is, a different name mapping rule is used: the name of the getter will be the same as the property name, and the name of the setter will be obtained by replacing is with set. For example, for a property isOpen, the getter will be called isOpen() and the setter will be called setOpen(). This rule applies for properties of any type, not just Boolean.

Package-level functions

All the functions and properties declared in a file app.kt inside a package org.example, including extension functions, are compiled into static methods of a Java class named org.example.AppKt.

```
// app.kt
package org.example

class Util

fun getTime() { /*...*/ }
```

```
// Java
new org.example.Util();
org.example.AppKt.getTime();
```

To set a custom name to the generated Java class, use the @JvmName annotation:

```
@file:JvmName("DemoUtils")

package org.example

class Util

fun getTime() { /*...*/ }
```

```
// Java
new org.example.Util();
org.example.DemoUtils.getTime();
```

Having multiple files with the same generated Java class name (the same package and the same name or the same `@JvmName` annotation) is normally an error. However, the compiler can generate a single Java facade class which has the specified name and contains all the declarations from all the files which have that name. To enable the generation of such a facade, use the `@JvmMultifileClass` annotation in all such files.

```
// oldutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package org.example

fun getTime() { /*...*/ }
```

```
// newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package org.example

fun getDate() { /*...*/ }
```

```
// Java
org.example.Utils.getTime();
org.example.Utils.getDate();
```

Instance fields

If you need to expose a Kotlin property as a field in Java, annotate it with the `@JvmField` annotation. The field will have the same visibility as the underlying property. You can annotate a property with `@JvmField` if it:

- has a backing field
- is not private
- does not have open, override or const modifiers
- is not a delegated property

```
class User(id: String) {
    @JvmField val ID = id
```

```
}
```

```
// Java
class JavaClient {
    public String getID(User user) {
        return user.ID;
    }
}
```

Late-Initialized properties are also exposed as fields. The visibility of the field will be the same as the visibility of `lateinit` property setter.

Static fields

Kotlin properties declared in a named object or a companion object will have static backing fields either in that named object or in the class containing the companion object.

Usually these fields are private but they can be exposed in one of the following ways:

- `@JvmField` annotation
- `lateinit` modifier
- `const` modifier

Annotating such a property with `@JvmField` makes it a static field with the same visibility as the property itself.

```
class Key(val value: Int) {
    companion object {
        @JvmField
        val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value }
    }
}
```

```
// Java
Key.COMPARATOR.compare(key1, key2);
// public static final field in Key class
```

A late-initialized property in an object or a companion object has a static backing field with the same visibility as the property setter.

```
object Singleton {
    lateinit var provider: Provider
}
```

```
// Java  
Singleton.provider = new Provider();  
// public static non-final field in Singleton class
```

Properties declared as const (in classes as well as at the top level) are turned into static fields in Java:

```
// file example.kt  
  
object Obj {  
    const val CONST = 1  
}  
  
class C {  
    companion object {  
        const val VERSION = 9  
    }  
}  
  
const val MAX = 239
```

In Java:

```
int const = Obj.CONST;  
int max = ExampleKt.MAX;  
int version = C.VERSION;
```

Static methods

As mentioned above, Kotlin represents package-level functions as static methods. Kotlin can also generate static methods for functions defined in named objects or companion objects if you annotate those functions as [@JvmStatic](#). If you use this annotation, the compiler will generate both a static method in the enclosing class of the object and an instance method in the object itself. For example:

```
class C {  
    companion object {  
        @JvmStatic fun callStatic() {}  
        fun callNonStatic() {}  
    }  
}
```

Now, `callStatic()` is static in Java while `callNonStatic()` is not:

```
C.callStatic(); // works fine  
C.callNonStatic(); // error: not a static method  
C.Companion.callStatic(); // instance method remains
```

```
C.Companion.callNonStatic(); // the only way it works
```

Same for named objects:

```
object Obj {
    @JvmStatic fun callStatic() {}
    fun callNonStatic() {}
}
```

In Java:

```
Obj.callStatic(); // works fine
Obj.callNonStatic(); // error
Obj.INSTANCE.callNonStatic(); // works, a call through the singleton instance
Obj.INSTANCE.callStatic(); // works too
```

Starting from Kotlin 1.3, `@JvmStatic` applies to functions defined in companion objects of interfaces as well. Such functions compile to static methods in interfaces. Note that static method in interfaces were introduced in Java 1.8, so be sure to use the corresponding targets.

```
interface ChatBot {
    companion object {
        @JvmStatic fun greet(username: String) {
            println("Hello, $username")
        }
    }
}
```

`@JvmStatic` annotation can also be applied on a property of an object or a companion object making its getter and setter methods static members in that object or the class containing the companion object.

Default methods in interfaces

Default methods are available only for targets JVM 1.8 and above.

Starting from JDK 1.8, interfaces in Java can contain default methods. To make all non-abstract members of Kotlin interfaces default for the Java classes implementing them, compile the Kotlin code with the `-Xjvm-default=all` compiler option.

Here is an example of a Kotlin interface with a default method:

```
// compile with -Xjvm-default=all
```

```
interface Robot {  
    fun move() { println(~walking~) } // will be default in the Java interface  
    fun speak(): Unit  
}
```

The default implementation is available for Java classes implementing the interface.

```
//Java implementation  
public class C3PO implements Robot {  
    // move() implementation from Robot is available implicitly  
    @Override  
    public void speak() {  
        System.out.println("I beg your pardon, sir");  
    }  
}
```

```
C3PO c3po = new C3PO();  
c3po.move(); // default implementation from the Robot interface  
c3po.speak();
```

Implementations of the interface can override default methods.

```
//Java  
public class BB8 implements Robot {  
    //own implementation of the default method  
    @Override  
    public void move() {  
        System.out.println(~rolling~);  
    }  
  
    @Override  
    public void speak() {  
        System.out.println("Beep-beep");  
    }  
}
```

Prior to Kotlin 1.4, to generate default methods, you could use the `@JvmDefault` annotation on these methods. Compiling with `-Xjvm-default=all` in 1.4+ generally works as if you annotated all non-abstract methods of interfaces with `@JvmDefault` and compiled with `-Xjvm-default=enable`. However, there are cases when their behavior differs. Detailed information about the changes in default methods generation in Kotlin 1.4 is provided in [this post](#) on the Kotlin blog.

Compatibility mode for default methods

If there are clients that use your Kotlin interfaces compiled without the new `-Xjvm-default=all` option, then they can be incompatible with the same code compiled with this option.

To avoid breaking the compatibility with such clients, compile your Kotlin code in the compatibility mode by specifying

the `-Xjvm-default=all-compatibility` compiler option. In this case, all the code that uses the previous version will work fine with the new one. However, the compatibility mode adds some overhead to the resulting bytecode size.

There is no need to consider compatibility for new interfaces, as no clients have used them before. You can minimize the compatibility overhead by excluding these interfaces from the compatibility mode. To do this, annotate them with the `@JvmDefaultWithoutCompatibility` annotation. Such interfaces compile the same way as with `-Xjvm-default=all`.

Additionally, in the all-compatibility mode you can use `@JvmDefaultWithoutCompatibility` to annotate all interfaces which are not exposed in the public API and therefore aren't used by the existing clients.

Visibility

The Kotlin visibility modifiers map to Java in the following way:

- private members are compiled to private members
- private top-level declarations are compiled to package-local declarations
- protected remains protected (note that Java allows accessing protected members from other classes in the same package and Kotlin doesn't, so Java classes will have broader access to the code)
- internal declarations become public in Java. Members of internal classes go through name mangling, to make it harder to accidentally use them from Java and to allow overloading for members with the same signature that don't see each other according to Kotlin rules
- public remains public

KClass

Sometimes you need to call a Kotlin method with a parameter of type KClass. There is no automatic conversion from Class to KClass, so you have to do it manually by invoking the equivalent of the `Class<T>.kotlin` extension property:

```
kotlin.jvm.JvmClassMappingKt.getKotlinClass(MainView.class)
```

Handling signature clashes with @JvmName

Sometimes we have a named function in Kotlin, for which we need a different JVM name in the bytecode. The most prominent example happens due to type erasure:

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

These two functions can not be defined side-by-side, because their JVM signatures are the same:

`filterValid(List<String>)` and `filterValid(List<Int>)`. If we really want them to have the same name in Kotlin, we can annotate one (or both) of them with `@JvmName` and specify a different name as an argument:

```
fun List<String>.filterValid(): List<String>
    @JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

From Kotlin they will be accessible by the same name `filterValid`, but from Java it will be `filterValid` and `filterValidInt`.

The same trick applies when we need to have a property `x` alongside with a function `getX()`:

```
val x: Int
    @JvmName("getX_prop")
    get() = 15

fun getX() = 10
```

To change the names of generated accessor methods for properties without explicitly implemented getters and setters, you can use `@get:JvmName` and `@set:JvmName`:

```
@get:JvmName("x")
@Setter:JvmName("changeX")
var x: Int = 23
```

Overloads generation

Normally, if you write a Kotlin function with default parameter values, it will be visible in Java only as a full signature, with all parameters present. If you wish to expose multiple overloads to Java callers, you can use the `@JvmOverloads` annotation.

The annotation also works for constructors, static methods, and so on. It can't be used on abstract methods, including methods defined in interfaces.

```
class Circle @JvmOverloads constructor(centerX: Int, centerY: Int, radius: Double = 1.0) {
    @JvmOverloads fun draw(label: String, lineWidth: Int = 1, color: String = "red") { /*...*/ }
}
```

For every parameter with a default value, this will generate one additional overload, which has this parameter and all parameters to the right of it in the parameter list removed. In this example, the following will be generated:

```
// Constructors:
Circle(int centerX, int centerY, double radius)
Circle(int centerX, int centerY)
```

```
// Methods
void draw(String label, int lineWidth, String color) { }
void draw(String label, int lineWidth) { }
void draw(String label) { }
```

Note that, as described in [Secondary constructors](#), if a class has default values for all constructor parameters, a public constructor with no arguments will be generated for it. This works even if the @JvmOverloads annotation is not specified.

Checked exceptions

Kotlin does not have checked exceptions. So, normally the Java signatures of Kotlin functions do not declare exceptions thrown. Thus, if you have a function in Kotlin like this:

```
// example.kt
package demo

fun writeToFile() {
    /*...*/
    throw IOException()
}
```

And you want to call it from Java and catch the exception:

```
// Java
try {
    demo.Example.writeToFile();
} catch (IOException e) {
    // error: writeToFile() does not declare IOException in the throws list
    // ...
}
```

You get an error message from the Java compiler, because `writeToFile()` does not declare `IOException`. To work around this problem, use the `@Throws` annotation in Kotlin:

```
@Throws(IOException::class)
fun writeToFile() {
    /*...*/
    throw IOException()
}
```

Null-safety

When calling Kotlin functions from Java, nobody prevents us from passing null as a non-null parameter. That's why Kotlin generates runtime checks for all public functions that expect non-nulls. This way we get a `NullPointerException`

in the Java code immediately.

Variant generics

When Kotlin classes make use of [declaration-site variance](#), there are two options of how their usages are seen from the Java code. For example, imagine you have the following class and two functions that use it:

```
class Box<out T>(val value: T)

interface Base
class Derived : Base

fun boxDerived(value: Derived): Box<Derived> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value
```

A naive way of translating these functions into Java would be this:

```
Box<Derived> boxDerived(Derived value) { ... }
Base unboxBase(Box<Base> box) { ... }
```

The problem is that in Kotlin you can write `unboxBase(boxDerived(Derived))` but in Java that would be impossible because in Java the class `Box` is invariant in its parameter `T`, and thus `Box<Derived>` is not a subtype of `Box<Base>`. To make this work in Java, you would have to define `unboxBase` as follows:

```
Base unboxBase(Box<? extends Base> box) { ... }
```

This declaration uses Java's wildcards types (`? extends Base`) to emulate declaration-site variance through use-site variance, because it is all Java has.

To make Kotlin APIs work in Java, the compiler generates `Box<Super>` as `Box<? extends Super>` for covariantly defined `Box` (or `Foo<? super Bar>` for contravariantly defined `Foo`) when it appears as a parameter. When it's a return value, wildcards are not generated, because otherwise Java clients will have to deal with them (and it's against the common Java coding style). Therefore, the functions from our example are actually translated as follows:

```
// return type - no wildcards
Box<Derived> boxDerived(Derived value) { ... }

// parameter - wildcards
Base unboxBase(Box<? extends Base> box) { ... }
```

When the argument type is final, there's usually no point in generating the wildcard, so `Box<String>` is always `Box<String>`, no matter what position it takes.

If you need wildcards where they are not generated by default, use the `@JvmWildcard` annotation:

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> = Box(value)
// is translated to
// Box<? extends Derived> boxDerived(Derived value) { ... }
```

In the opposite case, if you don't need wildcards where they are generated, use `@JvmSuppressWildcards`:

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
// is translated to
// Base unboxBase(Box<Base> box) { ... }
```

`@JvmSuppressWildcards` can be used not only on individual type arguments, but on entire declarations, such as functions or classes, causing all wildcards inside them to be suppressed.

Translation of type `Nothing`

The type `Nothing` is special, because it has no natural counterpart in Java. Indeed, every Java reference type, including `java.lang.Void`, accepts null as a value, and `Nothing` doesn't accept even that. So, this type cannot be accurately represented in the Java world. This is why Kotlin generates a raw type where an argument of type `Nothing` is used:

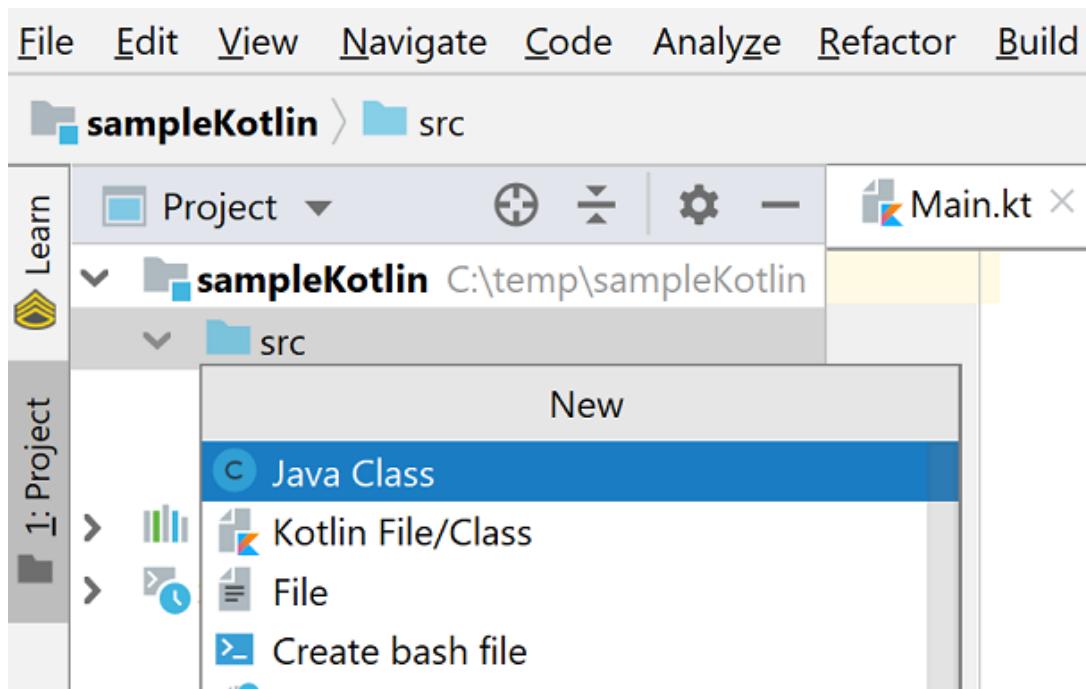
```
fun emptyList(): List<Nothing> = listOf()
// is translated to
// List emptyList() { ... }
```

Mixing Java and Kotlin in one project – tutorial

Kotlin provides the first-class interoperability with Java, and modern IDEs make it even better. In this tutorial, you'll learn how to use both Kotlin and Java sources in the same project in IntelliJ IDEA. To learn how to start a new Kotlin project in IntelliJ IDEA, see [Getting started with IntelliJ IDEA](#).

Adding Java source code to an existing Kotlin project

Adding Java classes to a Kotlin project is pretty straightforward. All you need to do is create a new Java file (`Alt + Insert/ Cmd + N`) in a directory or package inside your project.



Add new Java class

If you already have the Java classes, you can just copy them to the project directories.

You can now consume the Java class from Kotlin or vice versa without any further actions.

For example, adding the following Java class:

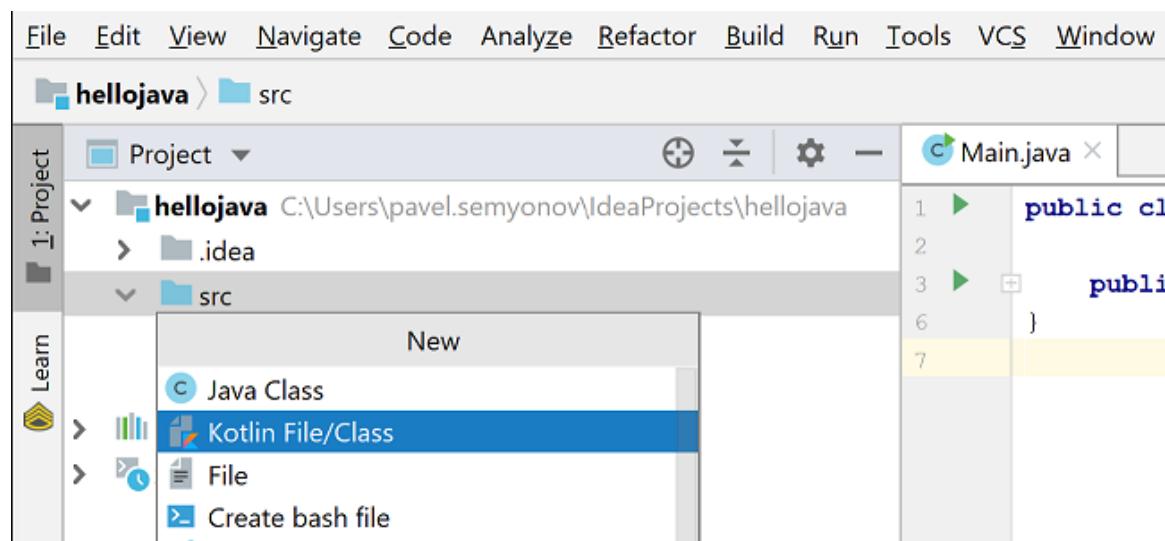
```
public class Customer {  
  
    private String name;  
  
    public Customer(String s){  
        name = s;  
    }  
  
    public String getName(){  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public void placeOrder() {  
        System.out.println("A new order is placed by " + name);  
    }  
}
```

lets you call it from Kotlin like any other type in Kotlin.

```
val customer = Customer("Phase")
println(customer.name)
println(customer.placeOrder())
```

Adding Kotlin source code to an existing Java project

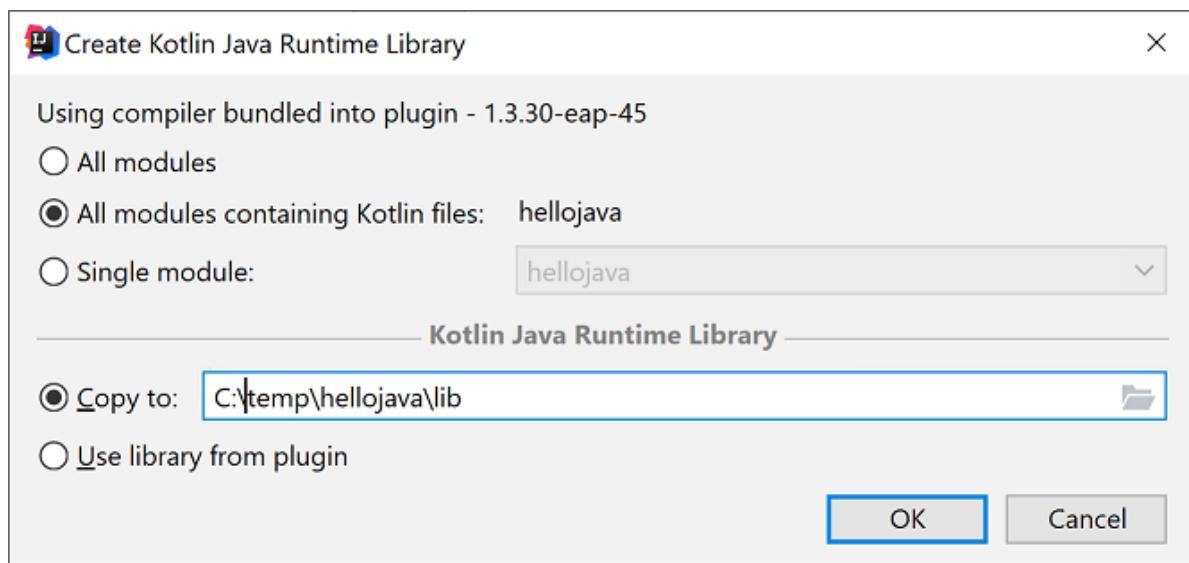
Adding a Kotlin file to an existing Java project is pretty much the same.



Add new Kotlin file class

If this is the first time you're adding a Kotlin file to this project, IntelliJ IDEA will prompt you to add the required Kotlin runtime. For a Java project, configure the Kotlin runtime as a Kotlin Java Module.

The next step is to decide which modules to configure (if the project has more than one module) and whether you want to add the runtime library to the project or use those provided by the current Kotlin plugin.

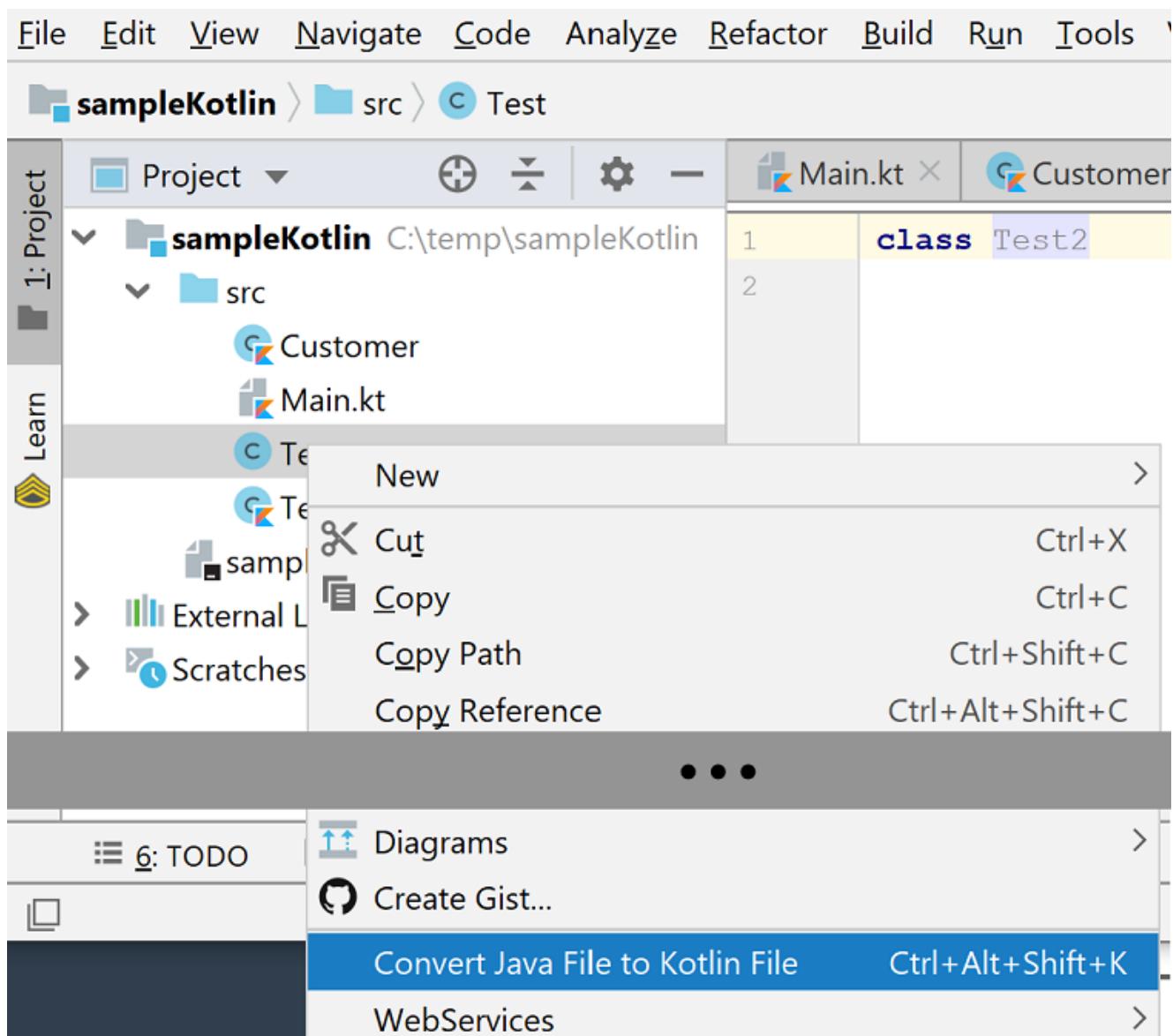


Bundling Kotlin runtime

You can also open the Kotlin runtime configuration manually from Tools | Kotlin | Configure Kotlin in Project.

Converting an existing Java file to Kotlin with J2K

The Kotlin plugin also bundles a Java to Kotlin converter (J2K) that automatically converts Java files to Kotlin. To use J2K on a file, click Convert Java File to Kotlin File in its context menu or in the Code menu of IntelliJ IDEA.



Convert Java to Kotlin

While the converter is not fool-proof, it does a pretty decent job of converting most boilerplate code from Java to Kotlin. Some manual tweaking however is sometimes required.

Comparison to Java

Some Java issues addressed in Kotlin

Kotlin fixes a series of issues that Java suffers from:

- Null references are controlled by the type system.

- No raw types
- Arrays in Kotlin are invariant
- Kotlin has proper function types, as opposed to Java's SAM-conversions
- Use-site variance without wildcards
- Kotlin does not have checked exceptions

What Java has that Kotlin does not

- Checked exceptions
- Primitive types that are not classes. The byte-code uses primitives where possible, but they are not explicitly available.
- Static members are replaced with companion objects, top-level functions, extension functions, or @JvmStatic.
- Wildcard-types are replaced with declaration-site variance and type projections.
- Ternary-operator a ? b : c is replaced with if expression.

What Kotlin has that Java does not

- Lambda expressions + Inline functions = performant custom control structures
- Extension functions
- Null-safety
- Smart casts
- String templates
- Properties
- Primary constructors
- First-class delegation
- Type inference for variable and property types
- Singletons
- Declaration-site variance & Type projections
- Range expressions

- [Operator overloading](#)
- [Companion objects](#)
- [Data classes](#)
- [Separate interfaces for read-only and mutable collections](#)
- [Coroutines](#)

What's next?

Learn how to perform [typical tasks with strings in Java and Kotlin](#).

Migrating from Java to Kotlin: Strings

This guide contains examples of how to perform typical tasks with strings in Java and Kotlin. It will help you migrate from Java to Kotlin and write your code in the authentic Kotlin way.

Concatenate strings

In Java, you can do this in the following way:

```
// Java
String name = "Joe";
System.out.println("Hello, " + name);
System.out.println("Your name is " + name.length() + " characters long");
```

In Kotlin, use the dollar sign (\$) before the variable name to interpolate the value of this variable into your string:

```
fun main() {
//sampleStart
    // Kotlin
    val name = "Joe"
    println("Hello, $name")
    println("Your name is ${name.length} characters long")
//sampleEnd
}
```

You can interpolate the value of a complicated expression by surrounding it with curly braces, like in \${name.length}. See [string templates](#) for more information.

Build a string

In Java, you can use the [StringBuilder](#):

```
// Java
StringBuilder countDown = new StringBuilder();
for (int i = 5; i > 0; i--) {
    countDown.append(i);
    countDown.append("\n");
}
System.out.println(countDown);
```

In Kotlin, use [buildString\(\)](#) – an [inline function](#) that takes logic to construct a string as a lambda argument:

```
fun main() {
//sampleStart
    // Kotlin
    val countDown = buildString {
        for (i in 5 downTo 1) {
            append(i)
            appendLine()
        }
    }
    println(countDown)
//sampleEnd
}
```

Under the hood, the `buildString` uses the same `StringBuilder` class as in Java, and you access it via an implicit `this` inside the [lambda](#).

Learn more about [lambda coding conventions](#).

Create a string from collection items

In Java, you use the [Stream API](#) to filter, map, and then collect the items:

```
// Java
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
String oddNumbers = numbers
    .stream()
    .filter(it -> it % 2 != 0)
    .map(it -> (-1) * it)
    .map(Object::toString)
    .collect(Collectors.joining(", "));
System.out.println(oddNumbers);
```

In Kotlin, use the [joinToString\(\)](#) function, which Kotlin defines for every `List`:

```
fun main() {
```

```
//sampleStart
// Kotlin
val numbers = mutableListOf(1, 2, 3, 4, 5, 6)
val oddNumbers = numbers
    .filter { it % 2 != 0 }
    .joinToString{ "${-it}" }
println(oddNumbers)
//sampleEnd
}
```

Set default value if the string is blank

In Java, you can use the [ternary operator](#):

```
// Java
public void defaultValueIfStringIsBlank() {
    String nameValue = getName();
    String name = nameValue.isBlank() ? "John Doe" : nameValue;
    System.out.println(name);
}

public String getName() {
    Random rand = new Random();
    return rand.nextBoolean() ? "" : "David";
}
```

Kotlin provides the inline function [ifBlank\(\)](#) that accepts the default value as an argument:

```
// Kotlin
import kotlin.random.Random

//sampleStart
fun main() {
    val name = getName().ifBlank { "John Doe" }
    println(name)
}

fun getName(): String =
    if (Random.nextBoolean()) "" else "David"
//sampleEnd
```

Replace characters at the beginning and end of a string

In Java, you can use the [replaceFirst\(\)](#) and the [replaceAll\(\)](#) functions. The replaceAll() function in this case accepts the regular expression `##$`, which defines a string ending with `##`:

```
// Java
String input = "##place##holder##";
String result = input.replaceFirst("##", "").replaceAll("##$", "");
```

```
System.out.println(result);
```

In Kotlin, use the [removeSurrounding\(\)](#) function with the string delimiter ##:

```
fun main() {
//sampleStart
    // Kotlin
    val input = "##place##holder##"
    val result = input.removeSurrounding("##")
    println(result)
//sampleEnd
}
```

Replace occurrences

In Java, you can use the [Pattern](#) and the [Matcher](#) classes, for example, to obfuscate some data:

```
// Java
String input = "login: Pokemon5, password: 1q2w3e4r5t";
Pattern pattern = Pattern.compile("\\w*\\d+\\w*");
Matcher matcher = pattern.matcher(input);
String replacementResult = matcher.replaceAll(it -> "xxx");
System.out.println("Initial input: '" + input + "'");
System.out.println("Anonymized input: '" + replacementResult + "'");
```

In Kotlin, you use the [Regex](#) class that simplifies working with regular expressions. Additionally, use [raw strings](#) to simplify a regex pattern by reducing the count of backslashes:

```
fun main() {
//sampleStart
    // Kotlin
    val regex = Regex("""\w*\d+\w*""") // raw string
    val input = "login: Pokemon5, password: 1q2w3e4r5t"
    val replacementResult = regex.replace(input, replacement = "xxx")
    println("Initial input: '$input'")
    println("Anonymized input: '$replacementResult'")
//sampleEnd
}
```

Split a string

In Java, to split a string with the period character (.), you need to use shielding (\.). This happens because the [split\(\)](#) function of the String class accepts a regular expression as an argument:

```
// Java
System.out.println(Arrays.toString("Sometimes.text.should.be.split".split("\.")));
```

In Kotlin, use the Kotlin function `split()`, which accepts varargs of delimiters as input parameters:

```
fun main() {  
    //sampleStart  
    // Kotlin  
    println("Sometimes.text.should.be.split".split("."))  
    //sampleEnd  
}
```

If you need to split with a regular expression, use the overloaded `split()` version that accepts the Regex as a parameter.

Take a substring

In Java, you can use the `substring()` function, which accepts an inclusive beginning index of a character to start taking the substring from. To take a substring after this character, you need to increment the index:

```
// Java  
String input = "What is the answer to the Ultimate Question of Life, the Universe, and  
Everything? 42";  
String answer = input.substring(input.indexOf("?") + 1);  
System.out.println(answer);
```

In Kotlin, you use the `substringAfter()` function and don't need to calculate the index of the character you want to take a substring after:

```
fun main() {  
    //sampleStart  
    // Kotlin  
    val input = "What is the answer to the Ultimate Question of Life, the Universe, and  
    Everything? 42"  
    val answer = input.substringAfter("?")  
    println(answer)  
    //sampleEnd  
}
```

Additionally, you can take a substring after the last occurrence of a character:

```
fun main() {  
    //sampleStart  
    // Kotlin  
    val input = "To be, or not to be, that is the question."  
    val question = input.substringAfterLast(",")  
    println(question)  
    //sampleEnd  
}
```

Use multiline strings

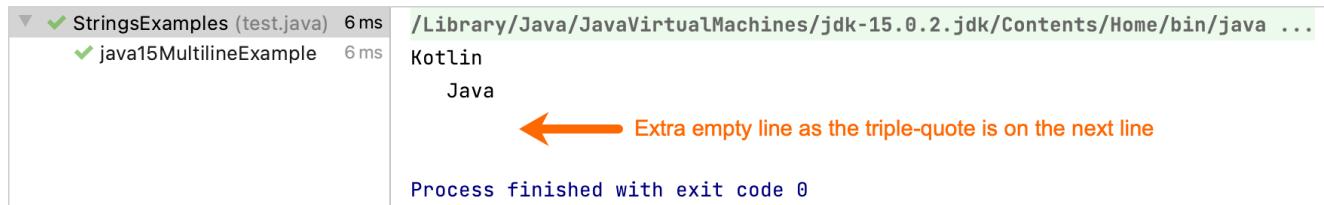
Before Java 15, there were several ways to create a multiline string. For example, using the `join()` function of the `String` class:

```
// Java
String lineSeparator = System.getProperty("line.separator");
String result = String.join(lineSeparator,
    "Kotlin",
    "Java");
System.out.println(result);
```

In Java 15, text blocks appeared. There is one thing to keep in mind: if you print a multiline string and the triple-quote is on the next line, there will be an extra empty line:

```
// Java
String result = """
    Kotlin
    Java
""".stripIndent();
System.out.println(result);
```

The output:



```
StringsExamples (test.java) 6 ms
  ✓ java15MultilineExample 6 ms
      Kotlin
      Java
      ← Extra empty line as the triple-quote is on the next line
      Process finished with exit code 0
```

Java 15 multiline output

If you put the triple-quote on the same line as the last word, this difference in behavior disappears.

In Kotlin, you can format your line with the quotes on the new line, and there will be no extra empty line in the output. The left-most character of any line identifies the beginning of the line.

```
fun main() {
//sampleStart
    // Kotlin
    val result = """
        Kotlin
        Java
    """.trimIndent()
    println(result)
//sampleEnd
}
```

The output:

```
/Library/Java/JavaVirtualMachines/jdk-15.0.2.jdk/Contents/Home/bin/java ...
Kotlin
Java
Process finished with exit code 0
```

Kotlin multiline output

In Kotlin, you can also use the [trimMargin\(\)](#) function to customize the indents:

```
// Kotlin
fun main() {
    val result = """
        # Kotlin
        # Java
    """.trimMargin("#")
    println(result)
}
```

Learn more about [multiline strings](#).

What's next?

- Look through other [Kotlin idioms](#).
- Learn how to convert existing Java code to Kotlin with [Java to Kotlin converter](#).

If you have a favorite idiom, contribute it by sending a pull request.

Get started with Kotlin/JS for React

This tutorial demonstrates how to use IntelliJ IDEA for creating a frontend application with Kotlin/JS for React.

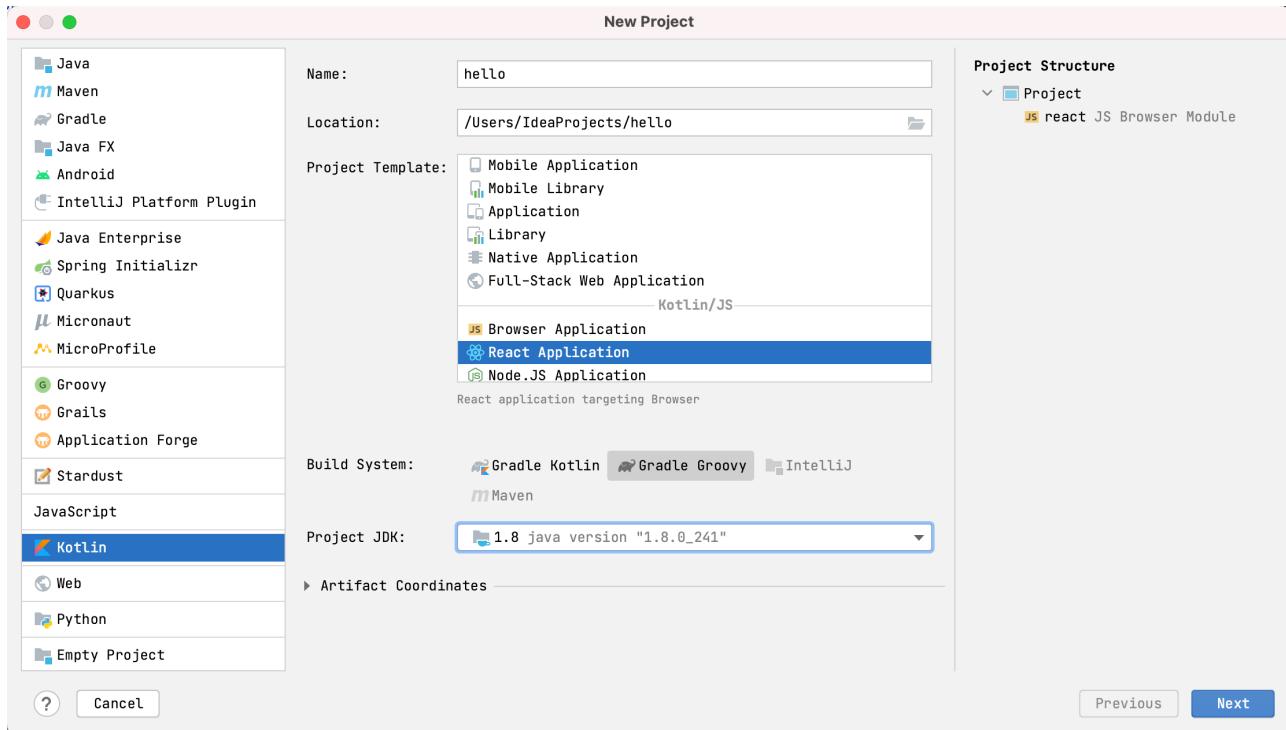
To get started, install the latest version of [IntelliJ IDEA](#).

Create an application

Once you've installed IntelliJ IDEA, it's time to create your first frontend application based on Kotlin/JS with React.

1. In IntelliJ IDEA, select File | New | Project.
2. In the panel on the left, select Kotlin.

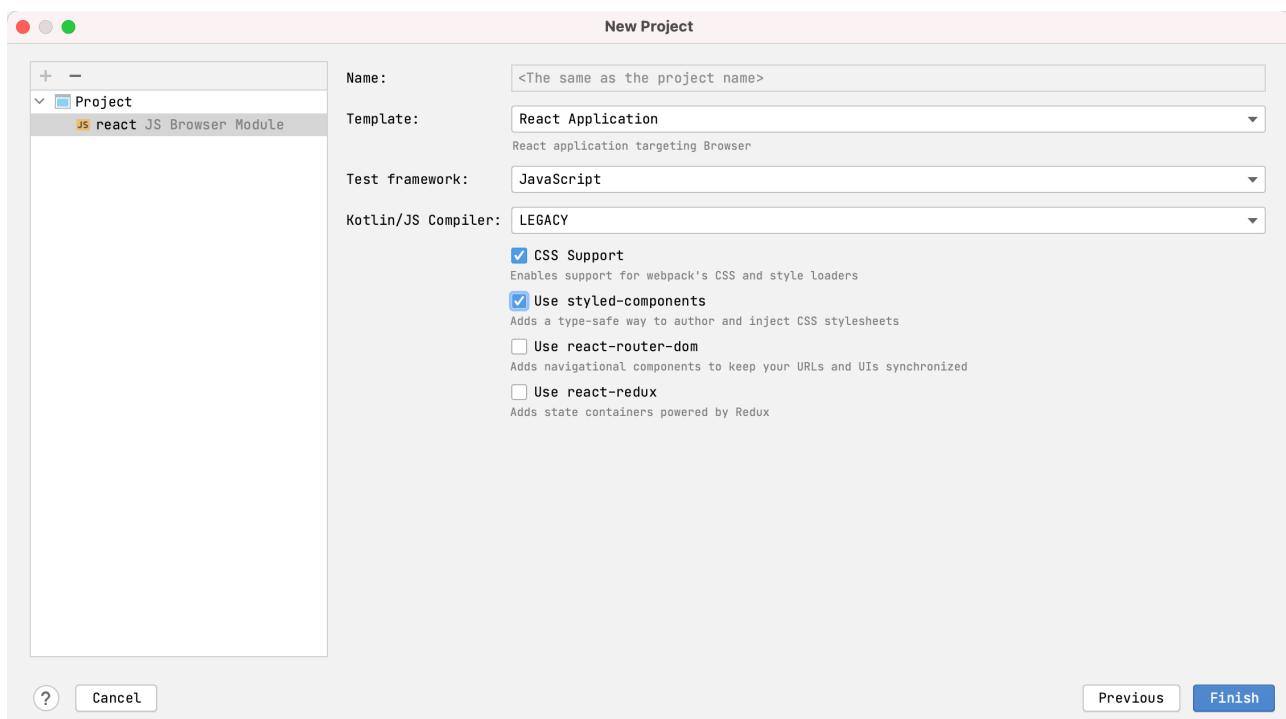
3. Enter a project name, select React Application as the project template, and click Next.



Create a react application

By default, your project will use Gradle with Kotlin DSL as the build system.

4. Select the CSS Support and Use styled-components checkboxes and click Finish.

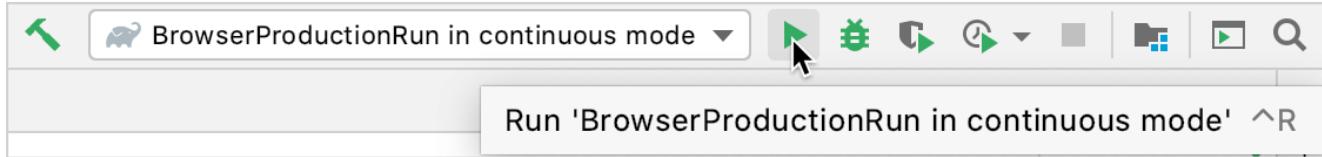


Configure a frontend application

Your project opens. By default, you see the file build.gradle.kts, which is the build script created by the Project Wizard based on your configuration. It includes the [kotlin\("js"\) plugin](#) and [dependencies](#) required for your frontend application.

Run the application

Start the application by clicking Run next to the run configuration at the top of the screen.



Running a frontend app

Your default web browser opens the URL <http://localhost:8080/> with your frontend application.



Web browser with JS application

Enter your name in the text box and accept the greetings from your application!

Update the application

Show your name backwards

1. Open the file welcome.kt in src/main/kotlin.

The src directory contains Kotlin source files and resources. The file welcome.kt includes sample code that renders the web page you've just seen.

The screenshot shows the IntelliJ IDEA interface. On the left is the project tree, which includes a 'JS' folder containing '.gradle', '.idea', 'build', 'gradle', 'src' (with 'main' and 'kotlin' subfolders), 'test', 'build.gradle.kts', 'gradle.properties', 'gradlew', 'gradlew.bat', 'local.properties', and 'settings.gradle.kts'. It also lists 'External Libraries' and 'Scratches and Consoles'. The right pane displays the 'welcome.kt' file's code:

```

17 class Welcome(props: WelcomeProps) : RComponent<WelcomeProps, WelcomeState>(props) {
18     init {
19         state = WelcomeState(props.name)
20     }
21
22     override fun RBuilder.render() {
23         styledDiv { this: StyledDOMBuilder<DIV>
24             css { this: CSSBuilder
25                 +WelcomeStyles.textContainer
26             }
27             +"Hello, ${state.name}"
28         }
29         styledInput { this: StyledDOMBuilder<INPUT>
30             css { this: CSSBuilder
31                 +WelcomeStyles.textInput
32             }
33             attrs { this: INPUT
34                 type = InputType.text
35                 value = state.name
36                 onChangeFunction = { event ->
37                     setState(
38                         WelcomeState(name = (event.target as HTMLInputElement).value)
39                     )
40                 }
41             }
42         }
43     }
44 }

```

Source code for frontend application

2. Change the code of styledDiv to show your name backwards.

- Use the standard library function `reversed()` to reverse your name.
- Use a string template for your reversed name by adding a dollar sign \$ and enclosing it in curly braces – `${state.name.reversed()}`.

```

styledDiv {
    css {
        +WelcomeStyles.textContainer
    }
    +"Hello ${state.name}!"
    +" Your name backwards is ${state.name.reversed()}!"
}

```

3. Save your changes to the file.

4. Go to the browser and enjoy the result.

You will see the changes only if your previous application is still running. If you've stopped your application, run it again.

Hello Kate! Your name backwards is etaK!

Kate

Web browser with a reversed name

Add an image

1. Open the file welcome.kt in src/main/kotlin.
2. Add a div container with a child image element img after the styledInput block.

Make sure that you import the react.dom.* and styled.* packages.

```
div {  
  img(src = "https://placekitten.com/408/287") {}  
}
```

3. Save your changes to the file.
4. Go to the browser and enjoy the result.

You will only see the changes if your previous application is still running. If you've stopped your application, [run it again](#).

Hello Kotlin/JS! Your name backwards is SJ/niltoK!

Kotlin/JS



Web page with with an image

Add a button that changes text

1. Open the file welcome.kt in src/main/kotlin.
2. Add a button element with an onClickFunction event handler.

Make sure that you import the package kotlinx.html.js.*.

```
button {  
    attrs.onClickFunction = {  
        setState(  
            WelcomeState(name = "Some name")  
    )  
}
```

```
    }
    +"Change name"
}
```

3. Save your changes to the file.
4. Go to the browser and enjoy the result.

You will only see the changes if your previous application is still running. If you've stopped your application, [run it again](#).

Hello Kotlin/JS! Your name backwards is SJ/niltoK!

Kotlin/JS



Web page with a button

What's next?

Once you have created your first application, you can go to Kotlin hands-on labs and complete long-form tutorials on Kotlin/JS. They include sample projects, which can serve as nice jumping-off points for your own projects, and contain

useful snippets and patterns.

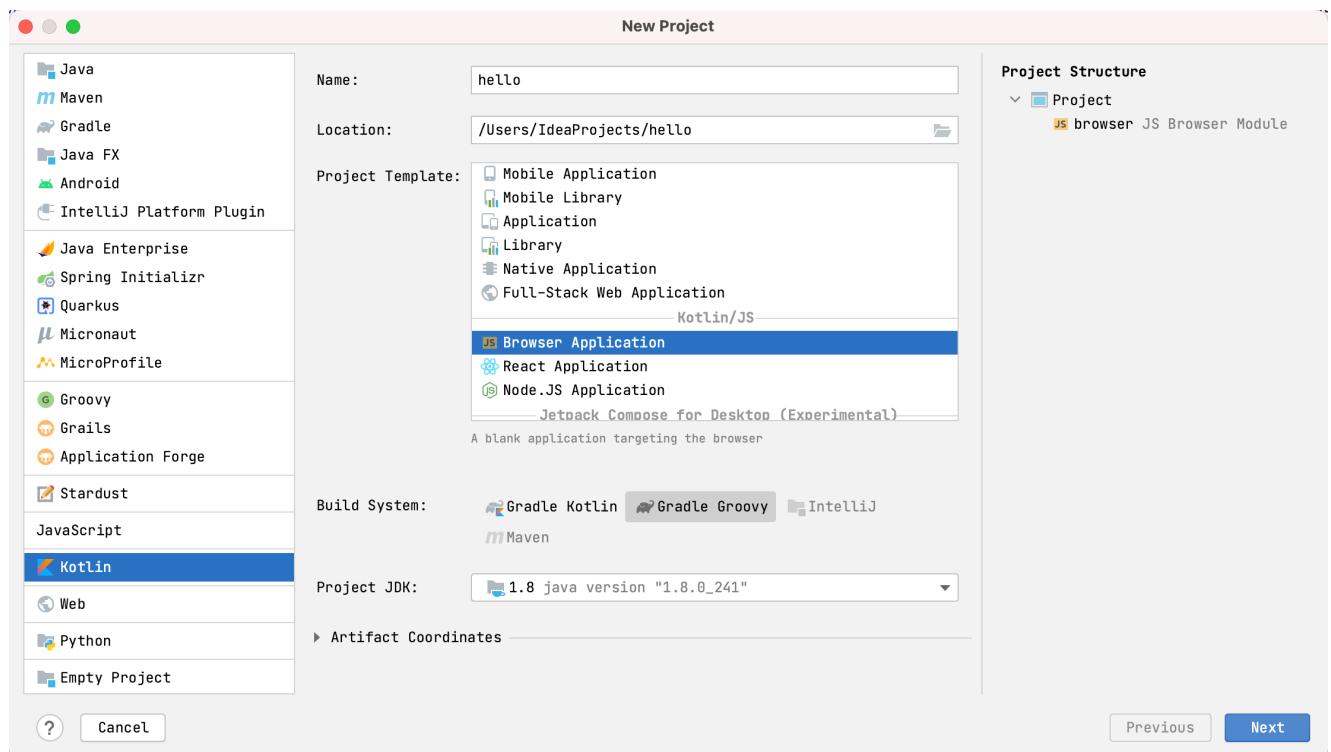
For Kotlin/JS, the following hands-on labs are currently available:

- [Building Web Applications with React and Kotlin/JS](#) guides you through the process of building a simple web application using the React framework, shows how a type-safe Kotlin DSL for HTML makes it easy to build reactive DOM elements, and illustrates how to use third-party React components, and how to obtain information from APIs, while writing the whole application logic in pure Kotlin/JS.
- [Building a Full Stack Web App with Kotlin Multiplatform](#) teaches the concepts behind building an application that targets Kotlin/JVM and Kotlin/JS by building a client-server application that makes use of shared code, serialization, and other multiplatform paradigms. It also provides a brief introduction to working with Ktor both as a server- and client-side framework.

Set up a Kotlin/JS project

Kotlin/JS projects use Gradle as a build system. To let developers easily manage their Kotlin/JS projects, we offer the `kotlin.js` Gradle plugin that provides project configuration tools together with helper tasks for automating routines typical for JavaScript development. For example, the plugin downloads the [Yarn](#) package manager for managing `npm` dependencies in background and can build a JavaScript bundle from a Kotlin project using [webpack](#). Dependency management and configuration adjustments can be done to a large part directly from the Gradle build file, with the option to override automatically generated configurations for full control.

To create a Kotlin/JS project in IntelliJ IDEA, go to File | New | Project. Then select Kotlin and choose a Kotlin/JS target that suits you best. Don't forget to choose the language for the build script: Groovy or Kotlin.



New project wizard

Alternatively, you can apply the org.jetbrains.kotlin.js plugin to a Gradle project manually in the Gradle build file (build.gradle or build.gradle.kts).

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.js' version '1.5.20'
}
```

Kotlin

```
plugins {
    kotlin("js") version "'1.5.20"
}
```

The Kotlin/JS Gradle plugin lets you manage aspects of your project in the kotlin section of the build script.

```
kotlin {
    //...
}
```

Inside the kotlin section, you can manage the following aspects:

- [Target execution environment](#): browser or Node.js
- [Project dependencies](#): Maven and npm
- [Run configuration](#)
- [Test configuration](#)
- [Bundling and CSS support](#) for browser projects
- [Target directory and module name](#)
- [Project's package.json file](#)

Execution environments

Kotlin/JS projects can target two different execution environments:

- Browser for client-side scripting in browsers
- [Node.js](#) for running JavaScript code outside of a browser, for example, for server-side scripting.

To define the target execution environment for a Kotlin/JS project, add the js section with browser {} or nodejs {} inside.

```
kotlin {
    js {
        browser {}
        binaries.executable()
    }
}
```

The instruction binaries.executable() explicitly instructs the Kotlin compiler to emit executable .js files. This is the default behavior when using the current Kotlin/JS compiler, but the instruction is explicitly required if you are working with the [Kotlin/JS IR compiler](#), or have set kotlin.js.generate.executable.default=false in your gradle.properties. In those cases, omitting binaries.executable() will cause the compiler to only generate Kotlin-internal library files, which can be used from other projects, but not run on their own. (This is typically faster than creating executable files, and can be a possible optimization when dealing with non-leaf modules of your project.)

The Kotlin/JS plugin automatically configures its tasks for working with the selected environment. This includes downloading and installing the required environment and dependencies for running and testing the application. This allows developers to build, run and test simple projects without additional configuration.

Dependencies

Like any other Gradle projects, Kotlin/JS projects support traditional [Gradle dependency declarations](#) in the dependencies section of the build script.

Groovy

```
dependencies {  
    implementation 'org.example.myproject:1.1.0'  
}
```

Kotlin

```
dependencies {  
    implementation("org.example.myproject", "1.1.0")  
}
```

The Kotlin/JS Gradle plugin also supports dependency declarations for particular source sets in the `kotlin` section of the build script.

Groovy

```
kotlin {  
    sourceSets {  
        main {  
            dependencies {  
                implementation 'org.example.myproject:1.1.0'  
            }  
        }  
    }  
}
```

Kotlin

```
kotlin {  
    sourceSets["main"].dependencies {  
        implementation("org.example.myproject", "1.1.0")  
    }  
}
```

Please note that not all libraries available for the Kotlin programming language are available when targeting JavaScript: Only libraries that include artifacts for Kotlin/JS can be used.

If the library you are adding has dependencies on [packages from npm](#), Gradle will automatically resolve these transitive dependencies as well.

Kotlin standard libraries

The dependency on the Kotlin/JS [standard library](#) is mandatory for all Kotlin/JS projects, and as such is implicit – no artifacts need to be added.

If your project contains tests written in Kotlin, you should add a dependency on the [kotlin.test](#) library:

Groovy

```
dependencies {  
    testImplementation 'org.jetbrains.kotlin:kotlin-test-js'  
}
```

Kotlin

```
dependencies {  
    testImplementation(kotlin("test-js"))  
}
```

npm dependencies

In the JavaScript world, the most common way to manage dependencies is [npm](#). It offers the biggest public repository of JavaScript modules.

The Kotlin/JS Gradle plugin lets you declare npm dependencies in the Gradle build script, analogous to how you would declare any other dependencies.

To declare an npm dependency, pass its name and version to the `npm()` function inside a dependency declaration. You can also specify one or multiple version range based on [npm's semver syntax](#).

Groovy

```
dependencies {  
    implementation npm('react', '> 14.0.0 <=16.9.0')  
}
```

Kotlin

```
dependencies {  
    implementation(npm("react", "> 14.0.0 <=16.9.0"))  
}
```

To download and install your declared dependencies during build time, the plugin manages its own installation of the [Yarn](#) package manager.

Besides regular dependencies, there are three more types of dependencies that can be used from the Gradle DSL. To learn more about when each type of dependency can best be used, have a look at the official documentation linked from [npm](#):

- devDependencies, via `devNpm(...)`,
- optionalDependencies via `optionalNpm(...)`, and
- peerDependencies via `peerNpm(...)`.

Once an npm dependency is installed, you can use its API in your code as described in [Calling JS from Kotlin](#).

run task

The Kotlin/JS plugin provides a run task that lets you run pure Kotlin/JS projects without additional configuration.

For running Kotlin/JS projects in the browser, this task is an alias for the `browserDevelopmentRun` task (which is also available in Kotlin multiplatform projects). It uses the [webpack-dev-server](#) to serve your JavaScript artifacts. If you want to customize the configuration used by `webpack-dev-server`, for example adjust the port the server runs on, use the [webpack configuration file](#).

For running Kotlin/JS projects targeting Node.js, the run task is an alias for the `nodeRun` task (which is also available in Kotlin multiplatform projects).

To run a project, execute the standard lifecycle run task, or the alias to which it corresponds:

```
./gradlew run
```

To automatically trigger a re-build of your application after making changes to the source files, use the Gradle [continuous build](#) feature:

```
./gradlew run --continuous
```

or

```
./gradlew run -t
```

Once the build of your project has succeeded, the `webpack-dev-server` will automatically refresh the browser page.

test task

The Kotlin/JS Gradle plugin automatically sets up a test infrastructure for projects. For browser projects, it downloads and installs the [Karma](#) test runner with other required dependencies; for Node.js projects, the [Mocha](#) test framework is used.

The plugin also provides useful testing features, for example:

- Source maps generation

- Test reports generation
- Test run results in the console

For running browser tests, the plugin uses [Headless Chrome](#) by default. You can also choose other browser to run tests in, by adding the corresponding entries inside the `useKarma` section of the build script:

```
kotlin {
    js {
        browser {
            testTask {
                useKarma {
                    useIE()
                    useSafari()
                    useFirefox()
                    useChrome()
                    useChromeCanary()
                    useChromeHeadless()
                    usePhantomJS()
                    useOpera()
                }
            }
            binaries.executable()
            // ...
        }
    }
}
```

Please note that the Kotlin/JS Gradle plugin does not automatically install these browsers for you, but only uses those that are available in its execution environment. If you are executing Kotlin/JS tests on a continuous integration server, for example, make sure that the browsers you want to test against are installed.

If you want to skip tests, add the line `enabled = false` to the `testTask`.

```
kotlin {
    js {
        browser {
            testTask {
                enabled = false
            }
        }
        binaries.executable()
        // ...
    }
}
```

To run tests, execute the standard lifecycle check task:

```
./gradlew check
```

Karma configuration

The Kotlin/JS Gradle plugin automatically generates a Karma configuration file at build time which includes your settings from the `kotlin.js.browser.testTask.useKarma` block in your build.gradle(.kts). You can find the file at `build/js/packages/projectName-test/karma.conf.js`. To make adjustments to the configuration used by Karma, place your additional configuration files inside a directory called `karma.config.d` in the root of your project. All `.js` configuration files in this directory will be picked up and are automatically merged into the generated `karma.conf.js` at build time.

All karma configuration abilities are well described in Karma's [documentation](#).

webpack bundling

For browser targets, the Kotlin/JS plugin uses the widely known [webpack](#) module bundler.

webpack version

The Kotlin/JS plugin uses webpack 5.

If you have projects created with plugin versions earlier than 1.5.0, you can temporarily switch back to webpack 4 used in these versions by adding the following line to the project's `gradle.properties`:

```
kotlin.js.webpack.major.version=4
```

webpack task

The most common webpack adjustments can be made directly via the `kotlin.js.browser.webpackTask` configuration block in the Gradle build file:

- `outputFileName`- the name of the webpacked output file. It will be generated in `<projectDir>/build/distibution/` after an execution of a webpack task. The default value is the project name.
- `output.libraryTarget`- the module system for the webpacked output. Learn more about [available module systems for Kotlin/JS projects](#). The default value is `umd`.

```
webpackTask {  
    outputFileName = "mycustomfilename.js"  
    output.libraryTarget = "commonjs2"  
}
```

You can also configure common webpack settings to use in bundling, running, and testing tasks in the `commonWebpackConfig` block.

webpack configuration file

The Kotlin/JS Gradle plugin automatically generates a standard webpack configuration file at the build time. It is

located in build/js/packages/projectName/webpack.config.js.

If you want to make further adjustments to the webpack configuration, place your additional configuration files inside a directory called webpack.config.d in the root of your project. When building your project, all .js configuration files will automatically be merged into the build/js/packages/projectName/webpack.config.js file. To add a new [webpack loader](#), for example, add the following to a .js file inside the webpack.config.d:

```
config.module.rules.push({
    test: /\.extension$/,
    loader: 'loader-name'
});
```

All webpack configuration capabilities are well described in its [documentation](#).

Building executables

For building executable JavaScript artifacts through webpack, the Kotlin/JS plugin contains the browserDevelopmentWebpack and browserProductionWebpack Gradle tasks.

- browserDevelopmentWebpack creates development artifacts, which are larger in size, but take little time to create. As such, use the browserDevelopmentWebpack tasks during active development.
- browserProductionWebpack applies [dead code elimination](#) to the generated artifacts and minifies the resulting JavaScript file, which takes more time, but generates executables that are smaller in size. As such, use the browserProductionWebpack task when preparing your project for production use.

Execute either of these tasks to obtain the respective artifacts for development or production. The generated files will be available in build/distributions unless [specified otherwise](#).

```
./gradlew browserProductionWebpack
```

Note that these tasks will only be available if your target is configured to generate executable files (via binaries.executable()).

CSS

The Kotlin/JS Gradle plugin also provides support for webpack's [CSS](#) and [style](#) loaders. While all options can be changed by directly modifying the [webpack configuration files](#) that are used to build your project, the most commonly used settings are available directly from the build.gradle(.kts) file.

To turn on CSS support in your project, set the cssSupport.enabled flag in the Gradle build file in the commonWbpakConfig block. This configuration is also enabled by default when creating a new project using the wizard.

```
browser {
```

```

commonWebpackConfig {
    cssSupport.enabled = true
}
binaries.executable()
}

```

Alternatively, you can add CSS support independently for webpackTask, runTask, and testTask.

```

webpackTask {
    cssSupport.enabled = true
}
runTask {
    cssSupport.enabled = true
}
testTask {
    useKarma {
        // ...
        webpackConfig.cssSupport.enabled = true
    }
}

```

Activating CSS support in your project helps prevent common errors that occur when trying to use style sheets from an unconfigured project, such as Module parse failed: Unexpected character '@' (14:0).

You can use cssSupport.mode to specify how encountered CSS should be handled. The following values are available:

- "inline" (default): styles are added to the global <style> tag.
- "extract": styles are extracted into a separate file. They can then be included from an HTML page.
- "import": styles are processed as strings. This can be useful if you need access to the CSS from your code (such as val styles = require("main.css")).

To use different modes for the same project, use cssSupport.rules. Here, you can specify a list of KotlinWebpackCssRules, each of which define a mode, as well as include and exclude patterns.

Yarn

To configure additional Yarn features, place a .yarnrc file in the root of your project. At build time, it gets picked up automatically.

For example, to use a custom registry for npm packages, add the following line to a file called .yarnrc in the project root:

```
registry "http://my.registry/api/npm/"
```

To learn more about .yarnrc, please visit the [official Yarn documentation](#).

Distribution target directory

By default, the results of a Kotlin/JS project build reside in the /build/distribution directory within the project root.

To set another location for project distribution files, add the distribution block inside browser in the build script and assign a value to the directory property. Once you run a project build task, Gradle will save the output bundle in this location together with project resources.

Groovy

```
kotlin {  
    js {  
        browser {  
            distribution {  
                directory = file("$projectDir/output/")  
            }  
        }  
        binaries.executable()  
        // ...  
    }  
}
```

Kotlin

```
kotlin {  
    js {  
        browser {  
            distribution {  
                directory = File("$projectDir/output/")  
            }  
        }  
        binaries.executable()  
        // ...  
    }  
}
```

Module name

To adjust the name for the JavaScript module (which is generated in build/js/packages/myModuleName), including the corresponding .js and .d.ts files, use the moduleName option:

```
js {  
    moduleName = "myModuleName"  
}
```

Note that this does not affect the webpacked output in build/distributions.

package.json customization

The package.json file holds the metadata of a JavaScript package. Popular package registries such as npm require all published packages to have such a file. They use it to track and manage package publications.

The Kotlin/JS Gradle plugin automatically generates package.json for Kotlin/JS projects during build time. By default, the file contains essential data: name, version, license, and dependencies, and some other package attributes.

Aside from basic package attributes, package.json can define how a JavaScript project should behave, for example, identifying scripts that are available to run.

You can add custom entries to the project's package.json via the Gradle DSL. To add custom fields to your package.json, use the `customField` function in the compilations `packageJson` block:

```
kotlin {  
    js {  
        compilations["main"].packageJson {  
            customField("hello", mapOf("one" to 1, "two" to 2))  
        }  
    }  
}
```

When you build the project, this code will add the following block to the package.json file:

```
"hello": { "one": 1, "two": 2 }
```

Learn more about writing package.json files for npm registry in the [npm docs](#).

Troubleshooting

When building a Kotlin/JS project using Kotlin 1.3.xx, you may encounter a Gradle error if one of your dependencies (or any transitive dependency) was built using Kotlin 1.4 or higher: Could not determine the dependencies of task ':client:jsTestPackageJson'. Cannot choose between the following variants. This is a known problem, a workaround is provided [here](#).

Run Kotlin/JS

Since Kotlin/JS projects are managed with the Kotlin/JS Gradle plugin, you can run your project using the appropriate tasks. If you're starting with a blank project, ensure that you have some sample code to execute. Create the file `src/main/kotlin/App.kt` and fill it with a small "Hello, World"-type code snippet:

```
fun main() {  
    console.log("Hello, Kotlin/JS!")  
}
```

Depending on the target platform, some platform-specific extra setup might be required to run your code for the first

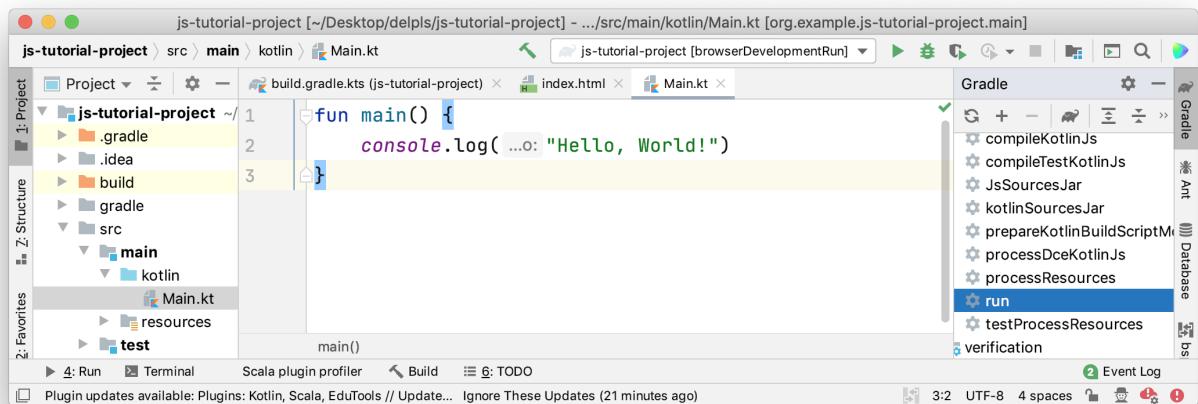
time.

Run the Node.js target

When targeting Node.js with Kotlin/JS, you can simply execute the run Gradle task. This can be done for example via the command line, using the Gradle wrapper:

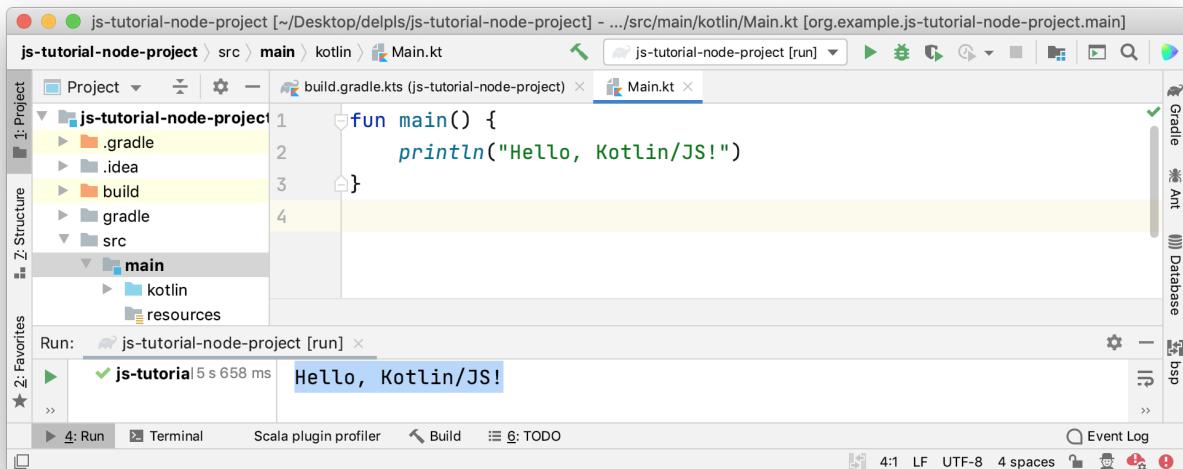
```
./gradlew run
```

If you're using IntelliJ IDEA, you can find the run action in the Gradle tool window:



Gradle Run task in IntelliJ IDEA

On first start, the kotlin.js Gradle plugin will download all required dependencies to get you up and running. After the build is completed, the program is executed, and you can see the logging output in the terminal:



Executing a Kotlin JS program in IntelliJ IDEA

Run the browser target

When targeting the browser, your project is required to have an HTML page. This page will be served by the development server while you are working on your application, and should embed your compiled Kotlin/JS file. Create and fill an HTML file /src/main/resources/index.html:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello, Kotlin/JS!</title>
</head>
<body>

</body>
<script src="jsTutorial.js"></script>
</html>
```

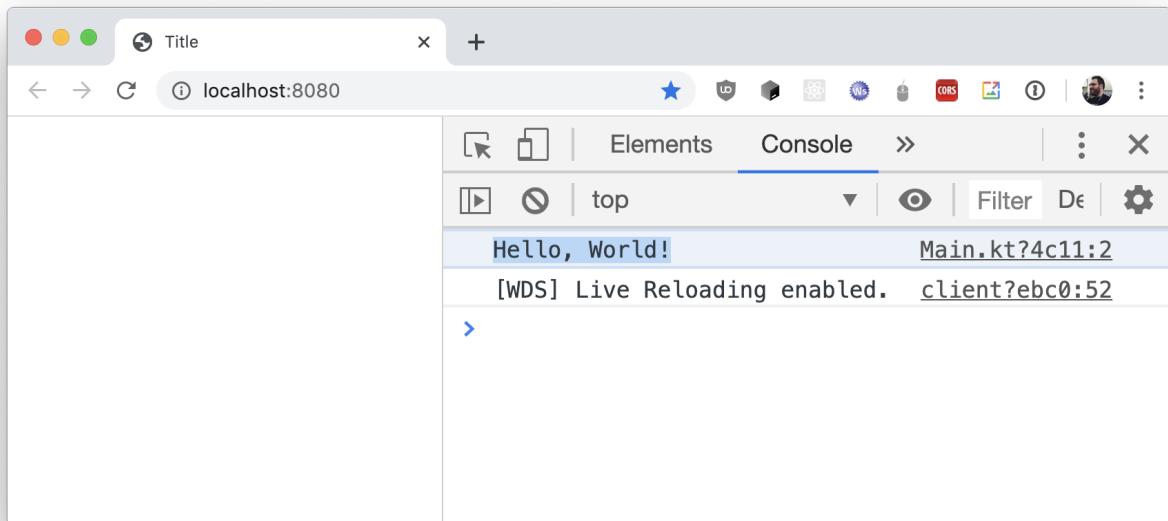
By default, the name of your project's generated artifact (which is created through webpack) that needs to be referenced is your project name (in this case, jsTutorial). If you've named your project followAlong, make sure to embed followAlong.js instead of jsTutorial.js

After making these adjustments, start the integrated development server. You can do this from the command line via the Gradle wrapper:

```
./gradlew run
```

When working from IntelliJ IDEA, you can find the run action in the Gradle tool window.

After the project has been built, the embedded webpack-dev-server will start running, and will open a (seemingly empty) browser window pointing to the HTML file you specified previously. To validate that your program is running correctly, open the developer tools of your browser (for example by right-clicking and choosing the Inspect action). Inside the developer tools, navigate to the console, where you can see the results of the executed JavaScript code:



Console output in browser developer tools

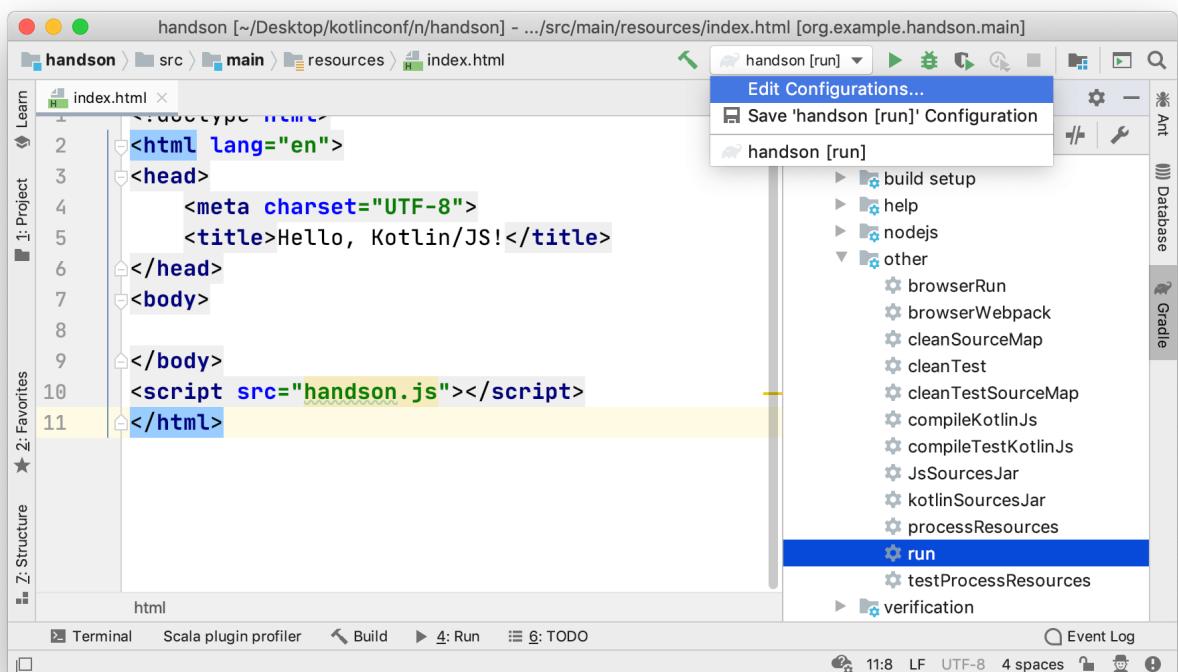
With this setup, you can recompile your project after each code change to see your changes. Kotlin/JS also supports a more convenient way of automatically rebuilding the application while you are developing it. To find out how to set up this continuous mode, check out the [corresponding tutorial](#).

Development server and continuous compilation

Instead of manually compiling and executing a Kotlin/JS project every time you want to see the changes you made, you can use the continuous compilation mode. Instead of using the regular run command, invoke the Gradle wrapper in continuous mode:

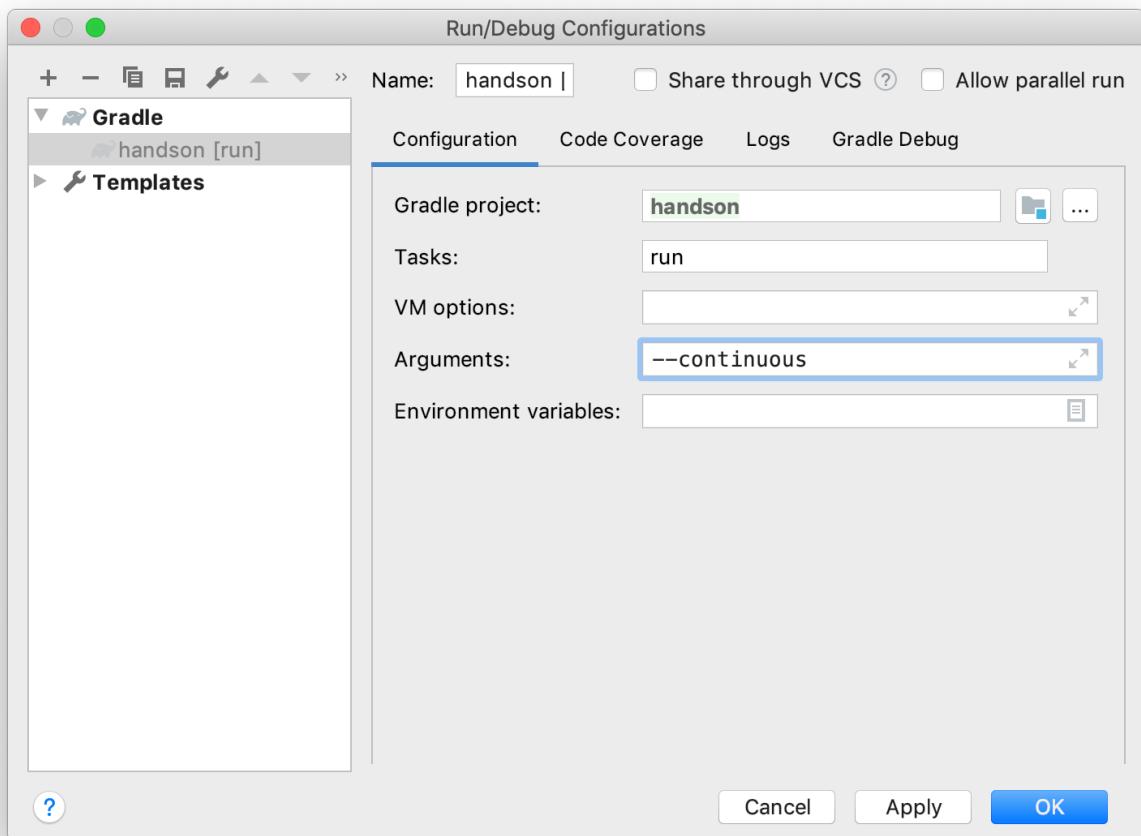
```
./gradlew run --continuous
```

If you are working in IntelliJ IDEA, you can pass the same flag via the run configuration. After running the Gradle run task for the first time from the IDE, IntelliJ IDEA automatically generates a run configuration for it, which you can edit:



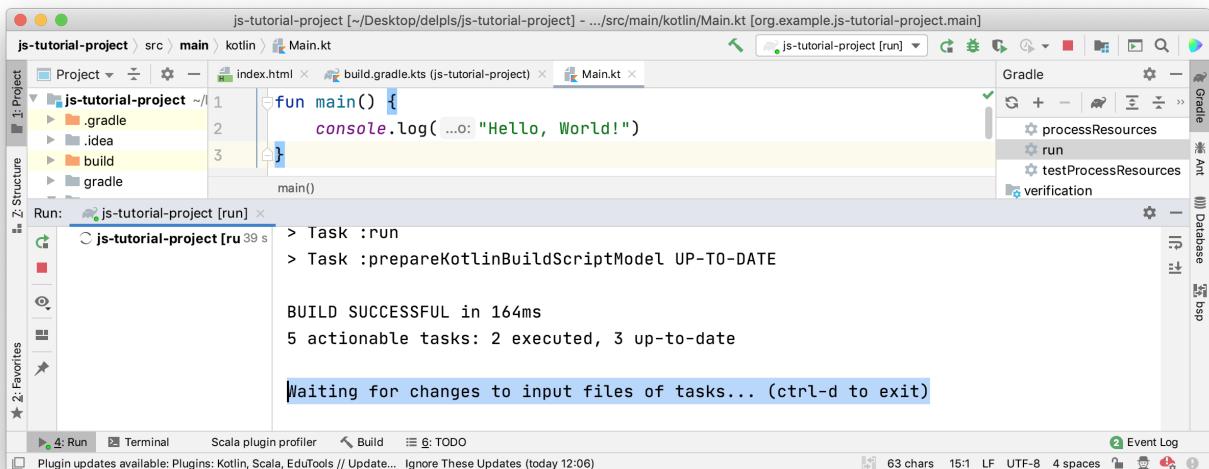
Editing run configurations in IntelliJ IDEA

Enabling continuous mode via the Run/Debug Configurations dialog is as easy as adding the --continuous flag to the arguments for the run configuration:



Adding the continuous flag to a run configuration in IntelliJ IDEA

When executing this run configuration, you can note that the Gradle process continues watching for changes to the program:



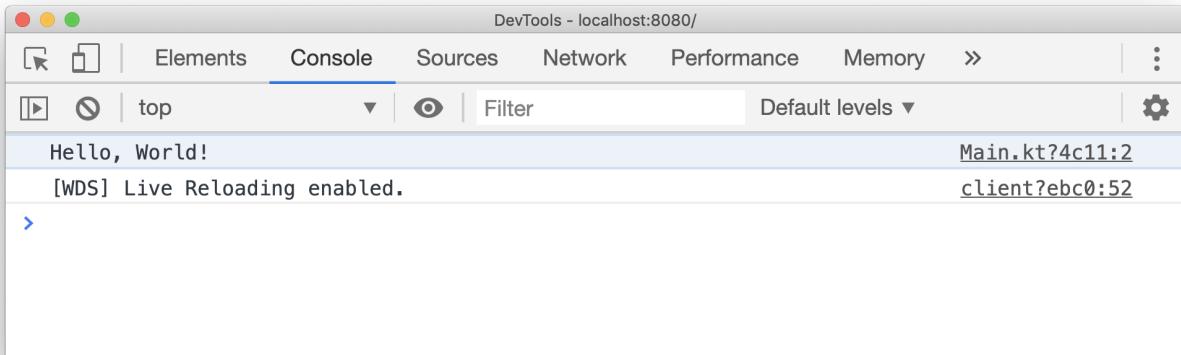
Gradle waiting for changes

Once a change has been detected, the program will be recompiled automatically. If you still have the page open in the browser, the development server will trigger an automatic reload of the page, and the changes will become visible. This is thanks to the integrated webpack-dev-server that is managed by the Kotlin/JS Gradle plugin.

Debug Kotlin/JS in the browser

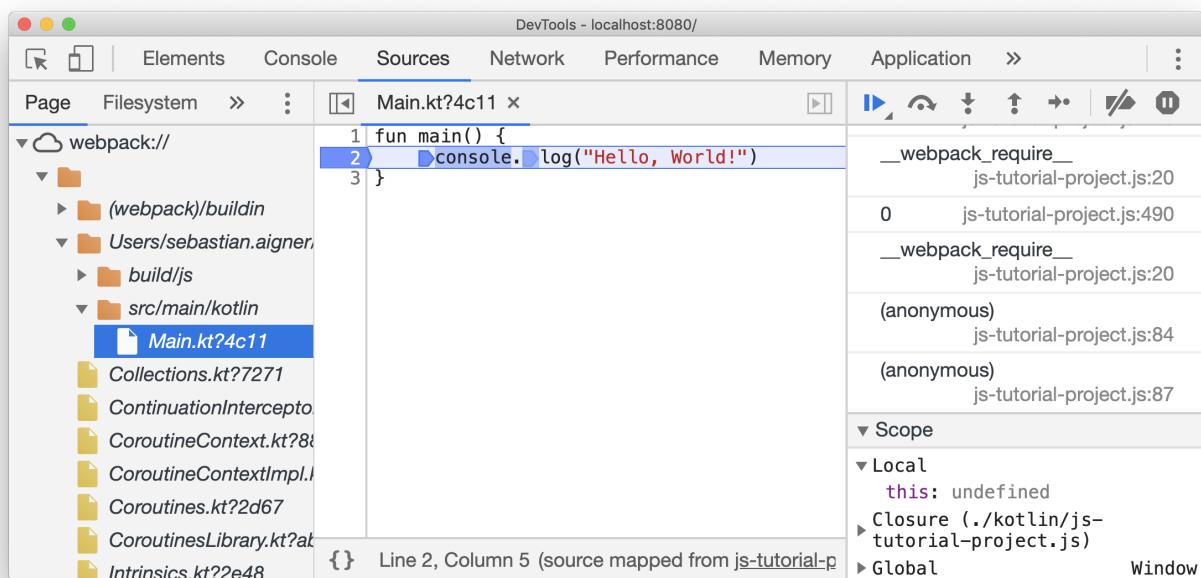
Kotlin/JS Gradle plugin automatically generates source maps for the project builds. These enable support for debugging the code using browser development tools. To get started debugging your code, execute the run task in a project using the JS Gradle plugin, or browserDevelopmentRun in a multiplatform project. Once the development server is running, you can start a debugging session from the browser.

To begin debugging, navigate to the page and launch the developer tools of the browser (for example by right-clicking and selecting the Inspect action). If your program is logging information to the console, navigate to the Console tab to see this output. Depending on your browser and its developer tools, these logs will also reference the Kotlin source files they originate from:



Chrome DevTools console

You can click the file reference on the right to navigate directly to the corresponding statement. Alternatively, you can manually switch to the Sources tab, and find the correct file in the file tree. Navigating to the Kotlin file actually shows you the regular Kotlin code (as opposed to minified JavaScript):



Debugging in Chrome DevTools

You can now start debugging the program. For example, set a breakpoint by clicking on one of the line numbers. The developer tools even support setting breakpoints within a statement. As with regular JavaScript code, any set breakpoints will persist across page reloads. This also makes it possible to debug Kotlin's main method which is executed when the script is loaded for the first time.

Run tests in Kotlin/JS

The Kotlin/JS Gradle plugin lets you run tests through a variety of test runners that can be specified via the Gradle configuration. In order to make test annotations and functionality available for the JavaScript target, add the correct platform artifact for `kotlin.test` in `build.gradle.kts`:

```
dependencies {
    // ...
    testImplementation(kotlin("test-js"))
}
```

You can tune how tests are executed in Kotlin/JS by adjusting the settings available in the `testTask` block in the Gradle build script. For example, using the Karma test runner together with a headless instance of Chrome and an instance of Firefox looks like this:

```
target {
    browser {
        testTask {
            useKarma {
                useChromeHeadless()
                useFirefox()
            }
        }
    }
}
```

For a detailed description of the available functionality, check out the Kotlin/JS reference on [configuring the test task](#).

Please note that by default, no browsers are bundled with the plugin. This means that you'll have to ensure they're available on the target system.

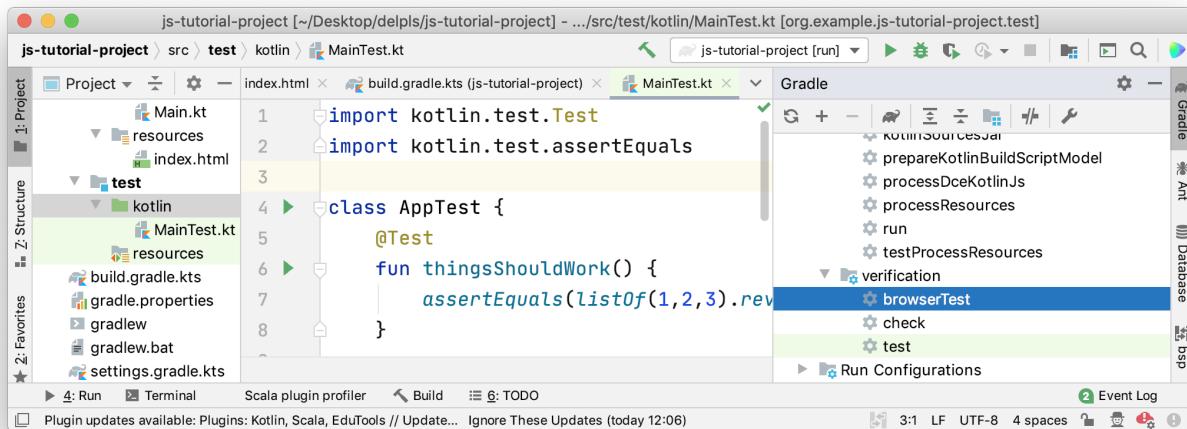
To check that tests are executed properly, add a file `src/test/kotlin/AppTest.kt` and fill it with this content:

```
import kotlin.test.Test
import kotlin.test.assertEquals

class AppTest {
    @Test
    fun thingsShouldWork() {
        assertEquals(listOf(1, 2, 3).reversed(), listOf(3, 2, 1))
    }

    @Test
    fun thingsShouldBreak() {
        assertEquals(listOf(1, 2, 3).reversed(), listOf(1, 2, 3))
    }
}
```

To run the tests in the browser, execute the `browserTest` task via IntelliJ IDEA, or use the gutter icons to execute all or individual tests:

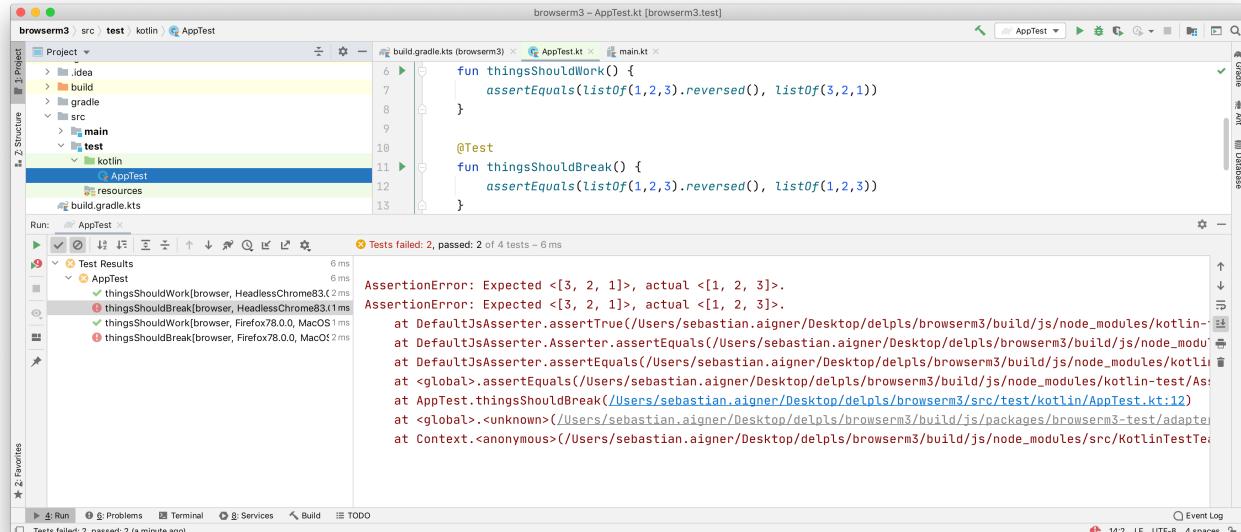


Gradle browserTest task

Alternatively, if you want to run the tests via the command line, use the Gradle wrapper:

```
./gradlew browserTest
```

After running the tests from IntelliJ IDEA, the Run tool window will show the test results. You can click failed tests to see their stack trace, and navigate to the corresponding test implementation via a double-click.



Test results in IntelliJ IDEA

After each test run, regardless of how you executed the test, you can find a properly formatted test report from Gradle in build/reports/tests/browserTest/index.html. Open this file in a browser to see another overview of the test results:

Test Summary

4	2	0	0.002s
tests	failures	ignored	duration

50%
successful

Failed tests [Packages](#) [Classes](#)

[AppTest. thingsShouldBreak\[browser, Firefox70.0.0, MacOSX10.14.0\]](#)
[AppTest. thingsShouldBreak\[browser, HeadlessChrome78.0.3904, MacOSX10.14.6\]](#)

Gradle test summary

If you are using the set of example tests shown in the snippet above, one test passes, and one test breaks, which gives the resulting total of 50% successful tests. To get more information about individual test cases, you can navigate via the provided hyperlinks:

The screenshot shows a browser window with the title "Test results - Class AppTest". The URL in the address bar is "/Users/sebastian.aigner...". The main content of the page is a red header "thingsShouldBreak[browser, HeadlessChrome78.0.3904, MacOSX10.14.6]" followed by a stacktrace of an `AssertionError`. The stacktrace is as follows:

```
AssertionError: Expected <[3, 2, 1]>, actual <[1, 2, 3]>.
    at AssertionError.init(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/kotlin/1.3.61/kotlin/exceptions.kt:102)
        at DefaultJsAssertor.failWithMessage(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/kotlin-test/1.3.61/src/main/kotlin/test/DefaultJsAssertor.kt:80)
            at DefaultJsAssertor.assertTrue(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/kotlin-test/1.3.61/src/main/kotlin/test/DefaultJsAssertor.kt:60)
                at DefaultJsAssertor.Assertor.assertEquals(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/kotlin-test/1.3.61/Asserts.kt:215)
                    at DefaultJsAssertor.assertEquals(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/kotlin-test/1.3.61/src/main/kotlin/test/DefaultJsAssertor.kt:27)
                        at <global>.assertEquals(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/kotlin-test/1.3.61/Assertions.kt:51)
                            at AppTest.thingsShouldBreak(/Users/sebastian.aigner/Desktop/kotlinconf/handson/src/test/kotlin/AppTest.kt:12)
                                at <global>.<unknown>(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/kotlin-test/adaptor_browser.js:52012)
```

Stacktrace of failed test in the Gradle summary

Kotlin/JS dead code elimination

The Kotlin/JS Gradle plugin includes a [dead code elimination](#) (DCE) tool. Dead code elimination is often also called tree shaking. It reduces the size of the resulting JavaScript code by removing unused properties, functions, and classes.

Unused declarations can appear in cases like:

- A function is inlined and never gets called directly (which happens always except for a few situations).
- A module uses a shared library. Without DCE, parts of the library that you don't use are still included in the resulting bundle. For example, the Kotlin standard library contains functions for manipulating lists, arrays, char sequences, adapters for DOM, and so on. All of this functionality would require about 1.3 MB as a JavaScript file. A simple "Hello, world" application only requires console routines, which is only a few kilobytes for the entire file.

The Kotlin/JS Gradle plugin handles DCE automatically when you build a production bundle, for example by using the `browserProductionWebpack` task. Development bundling tasks (like `browserDevelopmentWebpack`) don't include DCE.

Exclude declarations from DCE

Sometimes you may need to keep a function or a class in the resulting JavaScript code even if you don't use it in your module, for example, if you're going to use it in the client JavaScript code.

To keep certain declarations from elimination, add the `dceTask` block to your Gradle build script and list the declarations as arguments of the `keep` function. An argument must be the declaration's fully qualified name with the module name as a prefix: `moduleName.dot.separated.package.name.declarationName`

Unless specified otherwise, the names of functions and modules can be [mangled](#) in the generated JavaScript code. To keep such functions from elimination, use the mangled names in the `keep` arguments as they appear in the generated JavaScript code.

```
kotlin {  
    js {  
        browser {  
            dceTask {  
                keep("myKotlinJSModule.org.example.getName",  
"myKotlinJSModule.org.example.User")  
            }  
            binaries.executable()  
        }  
    }  
}
```

If you want to keep a whole package or module from elimination, you can use its fully qualified name as it appears in the generated JavaScript code.

Keeping whole packages or modules from elimination can prevent DCE from removing many unused declarations. Because of this, it is preferable to select individual declarations which should be excluded from DCE one by one.

Disable DCE

To turn off DCE completely, use the `devMode` option in the `dceTask`:

```
kotlin {  
    js {  
        browser {  
            dceTask {  
                dceOptions.devMode = true  
            }  
        }  
    }  
}
```

```
        }
        binaries.executable()
    }
}
```

Kotlin/JS 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](#).

The Kotlin/JS IR compiler backend is the main focus of innovation around Kotlin/JS, and paves the way forward for the technology.

Rather than directly generating JavaScript code from Kotlin source code, the Kotlin/JS IR compiler backend leverages a new approach. Kotlin source code is first transformed into a [Kotlin intermediate representation \(IR\)](#), which is subsequently compiled into JavaScript. For Kotlin/JS, this enables aggressive optimizations, and allows improvements on pain points that were present in the previous compiler, such as generated code size (through dead code elimination), and JavaScript and TypeScript ecosystem interoperability, to name some examples.

The IR compiler backend is available starting with Kotlin 1.4.0 through the Kotlin/JS Gradle plugin. To enable it in your project, pass a compiler type to the `js` function in your Gradle build script:

```
kotlin {
    js(IR) { // or: LEGACY, BOTH
        // ...
    }
    binaries.executable() // not applicable to BOTH, see details below
}
```

- IR uses the new IR compiler backend for Kotlin/JS.
- LEGACY uses the default compiler backend.
- BOTH compiles your project with the new IR compiler as well as the default compiler backend. Use this mode for [authoring libraries compatible with both backends](#).

The compiler type can also be set in the `gradle.properties` file, with the key `kotlin.js.compiler=ir`. This behaviour is overwritten by any settings in the `build.gradle(.kts)`, however.

Ignoring compilation errors

Ignore compilation errors mode 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/JS IR compiler provides a new compilation mode unavailable in the default backend – ignoring compilation errors. In this mode, you can try out your application even while its code contains errors. For example, when you're doing a complex refactoring or working on a part of the system that is completely unrelated to a compilation error in another part.

With this new compiler mode, the compiler ignores all broken code. Thus, you can run the application and try its parts that don't use the broken code. If you try to run the code that was broken during compilation, you'll get a runtime exception.

Choose between two tolerance policies for ignoring compilation errors in your code:

- SEMANTIC. The compiler will accept code that is syntactically correct but doesn't make sense semantically. For example, assigning a number to a string variable (type mismatch).
- SYNTAX. The compiler will accept any code, even if it contains syntax errors. Regardless of what you write, the compiler will still try to generate a runnable executable.

As an experimental feature, ignoring compilation errors requires an opt-in. To enable this mode, add the `-Xerror-tolerance-policy={SEMANTIC|SYNTAX}` compiler option:

```
kotlin {  
    js(IR) {  
        compilations.all {  
            compileKotlinTask.kotlinOptions.freeCompilerArgs += listOf("-Xerror-tolerance-  
policy=SYNTAX")  
        }  
    }  
}
```

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](#).

For better application startup performance, the Kotlin/JS IR compiler offers an option to initialize top-level properties lazily. This way, the application loads without initializing all the top-level properties used in its code. It initializes only the ones needed at startup; other properties receive their values later when the code that uses them actually runs.

As an experimental feature, lazy initialization of top-level properties requires an opt-in. To use the lazy initialization of top-level properties, add the `-Xir-property-lazy-initialization` option when compiling the code with the JS IR compiler:

Groovy

```
tasks.withType(Kotlin2JsCompile) {  
    kotlinOptions {  
        freeCompilerArgs += "-Xir-property-lazy-initialization"  
    }  
}
```

Kotlin

```
tasks.withType<Kotlin2JsCompile> {  
    kotlinOptions {  
        freeCompilerArgs += "-Xir-property-lazy-initialization"  
    }  
}
```

Preview: generation of TypeScript declaration files (d.ts)

The generation of TypeScript declaration files (d.ts) 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 Kotlin/JS IR compiler is capable of generating TypeScript definitions from your Kotlin code. These definitions can be used by JavaScript tools and IDEs when working on hybrid apps to provide autocompletion, support static analyzers, and make it easier to include Kotlin code in JavaScript and TypeScript projects.

Top-level declarations marked with `@JsExport` in a project that produces executable files (`binaries.executable()`) will get a .d.ts file generated, which contains the TypeScript definitions for the exported Kotlin declarations. These declarations can be found in `build/js/packages/<package_name>/kotlin` alongside the corresponding un-webpacked JavaScript code.

Current limitations of the IR compiler

A major change with the new IR compiler backend is the absence of binary compatibility with the default backend. A lack of such compatibility between the two backends for Kotlin/JS means that a library created with the new IR compiler backend can't be used from the default backend, and vice versa.

If you want to use the IR compiler backend for your project, you need to update all Kotlin dependencies to versions that support this new backend. Libraries published by JetBrains for Kotlin 1.4+ targeting Kotlin/JS already contain all

artifacts required for usage with the new IR compiler backend.

If you are a library author looking to provide compatibility with the current compiler backend as well as the new IR compiler backend, additionally check out the [section about authoring libraries for the IR compiler](#) section.

The IR compiler backend also has some discrepancies in comparison to the default backend. When trying out the new backend, it's good to be mindful of these possible pitfalls.

- Currently, the IR backend does not generate source maps for Kotlin code. You can follow the progress [on YouTrack](#).
- Some libraries that rely on specific characteristics of the default backend, such as kotlin-wrappers, can display some problems. You can follow the investigation and progress [on YouTrack](#).
- The IR backend does not make Kotlin declarations available to JavaScript by default at all. To make Kotlin declarations visible to JavaScript, they must be annotated with [@JsExport](#).

Migrating existing projects to the IR compiler

Due to significant differences between the two Kotlin/JS compilers, making your Kotlin/JS code work with the IR compiler may require some adjustments. Learn how to migrate existing Kotlin/JS projects to the IR compiler in the [Kotlin/JS IR compiler migration guide](#).

Authoring libraries for the IR compiler with backwards compatibility

If you're a library maintainer who is looking to provide compatibility with the default backend as well as the new IR compiler backend, a setting for the compiler selection is available that allows you to create artifacts for both backends, allowing you to keep compatibility for your existing users while providing support for the next generation of Kotlin compiler. This so-called both -mode can be turned on using the `kotlin.js.compiler=both` setting in your `gradle.properties` file, or can be set as one of the project-specific options inside your `js` block inside the `build.gradle(kts)` file:

```
kotlin {  
    js(BOTH) {  
        // ...  
    }  
}
```

When in both mode, the IR compiler backend and default compiler backend are both used when building a library from your sources (hence the name). This means that both `klib` files with Kotlin IR as well as `jar` files for the default compiler will be generated. When published under the same Maven coordinate, Gradle will automatically choose the right artifact depending on the use case – `js` for the old compiler, `klib` for the new one. This enables you to compile and publish your library for projects that are using either of the two compiler backends.

Migrating Kotlin/JS projects to 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](#).

We are replacing the current Kotlin/JS compiler with [the IR-based compiler](#) in order to unify Kotlin's behavior on all platforms and to make it possible to implement new JS-specific optimizations, among other reasons. You can learn more about the internal differences between the two compilers in the blog post [Migrating our Kotlin/JS app to the new IR compiler](#) by Sebastian Aigner.

Due to the significant differences between the compilers, switching your Kotlin/JS project from the old backend to the new one may require adjusting your code. On this page, we've compiled a list of known migration issues along with suggested solutions.

Note that this guide may change over time as we fix issues and find new ones. Please help us keep it complete – report any issues you encounter when switching to the IR compiler by submitting them to our issue tracker [YouTrack](#) or filling out [this form](#).

Convert JS- and React-related classes and interfaces to external interfaces

Issue: Using Kotlin interfaces and classes (including data classes) that derive from pure JS classes, such as React's RState and RProps, can cause a ClassCastException. Such exceptions appear because the compiler attempts to work with instances of these classes as if they were Kotlin objects, when they actually come from JS.

Solution: convert all classes and interfaces that derive from pure JS classes to [external interfaces](#):

```
// Replace this
interface AppState : RState { }
interface AppProps : RProps { }
data class CustomComponentState(var name: String) : RState
```

```
// With this
external interface AppState : RState { }
external interface AppProps : RProps { }
external interface CustomComponentState : RState {
    var name: String
}
```

In IntelliJ IDEA, you can use these [structural search and replace templates](#) to automatically mark interfaces as external:

- [Template for RState](#)

- [Template for RProps](#)

Convert properties of external interfaces to var

Issue: properties of external interfaces in Kotlin/JS code can't be read-only (val) properties because their values can be assigned only after the object is created with js() or jsObject() (a helper function from [kotlin-wrappers](#)):

```
val myState = js("{}") as CustomComponentState
myState.name = "name"
```

Solution: convert all properties of external interfaces to var:

```
// Replace this
external interface CustomComponentState : RState {
    val name: String
}
```

```
// With this
external interface CustomComponentState : RState {
    var name: String
}
```

Make boolean properties nullable in external interfaces

Issue: JavaScript treats the null or undefined value of a boolean variable as false. So, boolean properties can be used in expressions without being defined. This is okay in JavaScript, but not in Kotlin.

```
external interface ComponentProps: RProps {
    var isInitialized: Boolean
    var visible: Boolean
}
```

```
val props = js("{}") as ComponentProps
props.isInitialized = true
// visible is not initialized - OK in JS - means it's false
```

If you try to use such a property in a function overridden in Kotlin (for example, a React button), you'll get a ClassCastException:

```
button {
    attrs {
        autoFocus = props.visible // ClassCastException here
    }
}
```

Solution: make all Boolean properties of external interfaces nullable (Boolean?):

```
// Replace this
external interface ComponentProps: RProps {
    var visible: Boolean
}
```

```
// With this
external interface ComponentProps: RProps {
    var visible: Boolean?
}
```

Convert functions with receivers in external interfaces to regular functions

Issue: external declarations can't contain functions with receivers, such as extension functions or properties with corresponding functional types.

Solution: convert such functions and properties to regular functions by adding the receiver object as an argument:

```
// Replace this
external interface ButtonProps : RProps {
    var inside: StyledDOMBuilder<BUTTON>.() -> Unit
}
```

```
external interface ButtonProps : RProps {
    var inside: (StyledDOMBuilder<BUTTON>) -> Unit
}
```

Create plain JS objects for interoperability

Issue: properties of a Kotlin object that implements an external interface are not enumerable. This means that they are not visible for operations that iterate over the object's properties, for example:

- for (var name in obj)
- console.log(obj)
- JSON.stringify(obj)

Although they are still accessible by the name: obj.myProperty

```

external interface AppProps { var name: String }
data class AppPropsImpl(override var name: String) : AppProps
fun main() {
    val jsApp = js("{name: 'App1'}") as AppProps // plain JS object
    println("Kotlin sees: ${jsApp.name}") // "App1"
    println("JSON.stringify sees:" + JSON.stringify(jsApp)) // {"name": "App1"} - OK

    val ktApp = AppPropsImpl("App2") // Kotlin object
    println("Kotlin sees: ${ktApp.name}") // "App2"
    // JSON sees only the backing field, not the property
    println("JSON.stringify sees:" + JSON.stringify(ktApp)) // {"_name_3": "App2"}
}

```

Solution 1: create plain JavaScript objects with `js()` or `jsObject()` (a helper function from [kotlin-wrappers](#)):

```

external interface AppProps { var name: String }
data class AppPropsImpl(override var name: String) : AppProps

```

```

// Replace this
val ktApp = AppPropsImpl("App1") // Kotlin object

```

```

// With this
val jsApp = js("{name: 'App1'}") as AppProps // or jsObject {}

```

Solution 2: create objects with `kotlin.js.json()`:

```

// or with this
val jsonApp = kotlin.js.json(Pair("name", "App1")) as AppProps

```

Replace `toString()` calls on function references with `.name`

Issue: in the IR backend, calling `toString()` on function references doesn't produce unique values.

Solution: use the `name` property instead of `toString()`.

Browser and DOM API

The Kotlin/JS standard library lets you access browser-specific functionality using the `kotlinx.browser` package, which includes typical top-level objects such as `document` and `window`. The standard library provides typesafe wrappers for the functionality exposed by these objects wherever possible. As a fallback, the dynamic type is used to provide interaction with functions that do not map well into the Kotlin type system.

Interaction with the DOM

For interaction with the Document Object Model (DOM), you can use the variable document. For example, you can set the background color of our website through this object:

```
document.bgColor = "FFAA12"
```

The document object also provides you a way to retrieve a specific element by ID, name, class name, tag name and so on. All returned elements are of type Element?. To access their properties, you need to cast them to their appropriate type. For example, assume that you have an HTML page with an email <input> field:

```
<body>
  <input type="text" name="email" id="email"/>

  <script type="text/javascript" src="tutorial.js"></script>
</body>
```

Note that your script is included at the bottom of the body tag. This ensures that the DOM is fully available before the script is loaded.

With this setup, you can access elements of the DOM. To access the properties of the input field, invoke getElementById and cast it to HTMLInputElement. You can then safely access its properties, such as value:

```
val email = document.getElementById("email") as HTMLInputElement
email.value = "hadi@jetbrains.com"
```

Much like you reference this input element, you can access other elements on the page, casting them to the appropriate types.

To see how to create and structure elements in the DOM in a concise way, check out the [Typesafe HTML DSL](#).

Use JavaScript code from Kotlin

Kotlin was first designed for easy interoperation with the Java platform: it sees Java classes as Kotlin classes, and Java sees Kotlin classes as Java classes.

However, JavaScript is a dynamically typed language, which means it does not check types at compile time. You can freely talk to JavaScript from Kotlin via [dynamic](#) types. If you want to use the full power of the Kotlin type system, you can create external declarations for JavaScript libraries which will be understood by the Kotlin compiler and the surrounding tooling.

An experimental tool to automatically create Kotlin external declarations for npm dependencies which provide type definitions (TypeScript / d.ts) called [Dukat](#) is also available.

Inline JavaScript

You can inline some JavaScript code into your Kotlin code using the `js()` function. For example:

```
fun jsTypeOf(o: Any): String {
    return js("typeof o")
}
```

Because the parameter of `js` is parsed at compile time and translated to JavaScript code "as-is", it is required to be a string constant. So, the following code is incorrect:

```
fun jsTypeOf(o: Any): String {
    return js(getTypeof() + " o") // error reported here
}
fun getTypeof() = "typeof"
```

Note that invoking `js()` returns a result of type `dynamic`, which provides no type safety at the compile time.

external modifier

To tell Kotlin that a certain declaration is written in pure JavaScript, you should mark it with the `external` modifier. When the compiler sees such a declaration, it assumes that the implementation for the corresponding class, function or property is provided externally (by the developer or via an [npm dependency](#)), and therefore does not try to generate any JavaScript code from the declaration. This is also why external declarations can't have a body. For example:

```
external fun alert(message: Any?): Unit

external class Node {
    val firstChild: Node

    fun append(child: Node): Node

    fun removeChild(child: Node): Node

    // etc
}

external val window: Window
```

Note that the `external` modifier is inherited by nested declarations. This is why in the example `Node` class, there is no `external` modifier before member functions and properties.

The `external` modifier is only allowed on package-level declarations. You can't declare an `external` member of a non-`external` class.

Declare (static) members of a class

In JavaScript you can define members either on a prototype or a class itself:

```
function MyClass() { ... }
MyClass.sharedMember = function() { /* implementation */ };
MyClass.prototype.ownMember = function() { /* implementation */ };
```

There is no such syntax in Kotlin. However, in Kotlin we have [companion](#) objects. Kotlin treats companion objects of external classes in a special way: instead of expecting an object, it assumes members of companion objects to be members of the class itself. MyClass from the example above can be described as follows:

```
external class MyClass {
    companion object {
        fun sharedMember()
    }

    fun ownMember()
}
```

Declare optional parameters

If you are writing an external declaration for a JavaScript function which has an optional parameter, use `definedExternally`. This delegates the generation of the default values to the JavaScript function itself:

```
external fun myFunWithOptionalArgs(
    x: Int,
    y: String = definedExternally,
    z: String = definedExternally
)
```

With this external declaration, you can call `myFunWithOptionalArgs` with one required argument and two optional arguments, where the default values are calculated by the JavaScript implementation of `myFunWithOptionalArgs`.

Extend JavaScript classes

You can easily extend JavaScript classes as if they were Kotlin classes. Just define an external open class and extend it by a non- external class. For example:

```
open external class Foo {
    open fun run()
    fun stop()
}

class Bar: Foo() {
    override fun run() {
        window.alert("Running!")
    }

    fun restart() {
        window.alert("Restarting")
    }
}
```

```
    }  
}
```

There are some limitations:

- When a function of an external base class is overloaded by signature, you can't override it in a derived class.
- You can't override a function with default arguments.
- Non-external classes can't be extended by external classes.

external interfaces

JavaScript does not have the concept of interfaces. When a function expects its parameter to support two methods foo and bar, you would just pass in an object that actually has these methods.

You can use interfaces to express this concept in statically typed Kotlin:

```
external interface HasFooAndBar {  
    fun foo()  
  
    fun bar()  
}  
  
external fun myFunction(p: HasFooAndBar)
```

A typical use case for external interfaces is to describe settings objects. For example:

```
external interface JQueryAjaxSettings {  
    var async: Boolean  
  
    var cache: Boolean  
  
    var complete: (JQueryXHR, String) -> Unit  
  
    // etc  
}  
  
fun JQueryAjaxSettings(): JQueryAjaxSettings = js("{}")  
  
external class JQuery {  
    companion object {  
        fun get(settings: JQueryAjaxSettings): JQueryXHR  
    }  
}  
  
fun sendQuery() {  
    JQuery.get(JQueryAjaxSettings().apply {  
        complete = { (xhr, data) ->  
            window.alert("Request complete")  
        }  
    })  
}
```

External interfaces have some restrictions:

- They can't be used on the right-hand side of is checks.
- They can't be passed as reified type arguments.
- They can't be used in class literal expressions (such as I::class).
- as casts to external interfaces always succeed. Casting to external interfaces produces the "Unchecked cast to external interface" compile time warning. The warning can be suppressed with the @Suppress("UNCHECKED_CAST_TO_EXTERNAL_INTERFACE") annotation.

IntelliJ IDEA can also automatically generate the @Suppress annotation. Open the intentions menu via the light bulb icon or Alt-Enter, and click the small arrow next to the "Unchecked cast to external interface" inspection. Here, you can select the suppression scope, and your IDE will add the annotation to your file accordingly.

Casts

In addition to the "unsafe" cast operator as, which throws a ClassCastException in case a cast is not possible, Kotlin/JS also provides unsafeCast<T>(). When using unsafeCast, no type checking is done at all during runtime. For example, consider the following two methods:

```
fun usingUnsafeCast(s: Any) = s.unsafeCast<String>()
fun usingAsOperator(s: Any) = s as String
```

They will be compiled accordingly:

```
function usingUnsafeCast(s) {
    return s;
}

function usingAsOperator(s) {
    var tmp$;
    return typeof (tmp$ = s) === 'string' ? tmp$ : throwCCE();
}
```

Dynamic type

The dynamic type is not supported in code targeting the JVM.

Being a statically typed language, Kotlin still has to interoperate with untyped or loosely typed environments, such as the JavaScript ecosystem. To facilitate these use cases, the dynamic type is available in the language:

```
val dyn: dynamic = ...
```

The dynamic type basically turns off Kotlin's type checker:

- A value of the dynamic type can be assigned to any variable or passed anywhere as a parameter.
- Any value can be assigned to a variable of the dynamic type or passed to a function that takes dynamic as a parameter.
- null -checks are disabled for the dynamic type values.

The most peculiar feature of dynamic is that we are allowed to call any property or function with any parameters on a dynamic variable:

```
dyn.whatever(1, "foo", dyn) // 'whatever' is not defined anywhere
dyn.whatever(*arrayOf(1, 2, 3))
```

On the JavaScript platform this code will be compiled "as is": dyn.whatever(1) in Kotlin becomes dyn.whatever(1) in the generated JavaScript code.

When calling functions written in Kotlin on values of dynamic type, keep in mind the name mangling performed by the Kotlin to JavaScript compiler. You may need to use the [@JsName annotation](#) to assign well-defined names to the functions that you need to call.

A dynamic call always returns dynamic as a result, so you can chain such calls freely:

```
dyn.foo().bar.baz()
```

When you pass a lambda to a dynamic call, all of its parameters by default have the type dynamic:

```
dyn.foo {
    x -> x.bar() // x is dynamic
}
```

Expressions using values of dynamic type are translated to JavaScript "as is", and do not use the Kotlin operator conventions. The following operators are supported:

- binary: +, -, *, /, %, >, < >=, <=, ==, !=, ===, !==, &&, ||
- unary
 - prefix: -, +, !
 - prefix and postfix: ++, --
- assignments: +=, -=, *=, /=, %=
- indexed access:

- read: d[a], more than one argument is an error
- write: d[a1] = a2, more than one argument in [] is an error

in, !in and .. operations with values of type dynamic are forbidden.

For a more technical description, see the [spec document](#).

Use dependencies from npm

In Kotlin/JS projects, all dependencies can be managed through the Gradle plugin. This includes Kotlin/Multiplatform libraries such as kotlinx.coroutines, kotlinx.serialization, or ktor-client.

For depending on JavaScript packages from [npm](#), the Gradle DSL exposes an npm function that lets you specify packages you want to import from npm. Let's consider the import of an NPM package called [is-sorted](#).

The corresponding part in the Gradle build file looks as follows:

```
dependencies {
    // ...
    implementation(npm("is-sorted", "1.0.5"))
}
```

Because JavaScript modules are usually dynamically typed and Kotlin is a statically typed language, you need to provide a kind of adapter. In Kotlin, such adapters are called external declarations. For the is-sorted package which offers only one function, this declaration is small to write. Inside the source folder, create a new file called is-sorted.kt, and fill it with these contents:

```
@JsModule("is-sorted")
@JsNonModule
external fun <T> sorted(a: Array<T>): Boolean
```

Please note that if you're using CommonJS as a target, the @JsModule and @JsNonModule annotations need to be adjusted accordingly.

This JavaScript function can now be used just like a regular Kotlin function. Because we provided type information in the header file (as opposed to simply defining parameter and return type to be dynamic), proper compiler support and type-checking is also available.

```
console.log("Hello, Kotlin/JS!")
console.log(sorted(arrayOf(1, 2, 3)))
console.log(sorted(arrayOf(3, 1, 2)))
```

Running these three lines either in the browser or Node.js, the output shows that the call to sorted was properly mapped to the function exported by the is-sorted package:

```
Hello, Kotlin/JS!
```

```
true  
false
```

Because the JavaScript ecosystem has multiple ways of exposing functions in a package (for example through named or default exports), other npm packages might need a slightly altered structure for their external declarations.

To learn more about how to write declarations, please refer to [Calling JavaScript from Kotlin](#).

Use Kotlin code from JavaScript

Depending on the selected [JavaScript Module](#) system, the Kotlin/JS compiler generates different output. But in general, the Kotlin compiler generates normal JavaScript classes, functions and properties, which you can freely use from JavaScript code. There are some subtle things you should remember, though.

Isolating declarations in a separate JavaScript object in plain mode

If you have explicitly set your module kind to be plain, Kotlin creates an object that contains all Kotlin declarations from the current module. This is done to prevent spoiling the global object. This means that for a module myModule, all declarations are available to JavaScript via the myModule object. For example:

```
fun foo() = "Hello"
```

Can be called from JavaScript like this:

```
alert(myModule.foo());
```

This is not applicable when you compile your Kotlin module to JavaScript modules like UMD (which is the default setting for both browser and nodejs targets), CommonJS or AMD. In this case, your declarations will be exposed in the format specified by your chosen JavaScript module system. When using UMD or CommonJS, for example, your call site could look like this:

```
alert(require('myModule').foo());
```

Check the article on [JavaScript Modules](#) for more information on the topic of JavaScript module systems.

Package structure

Kotlin exposes its package structure to JavaScript, so unless you define your declarations in the root package, you have to use fully qualified names in JavaScript. For example:

```
package my.qualified.packagename

fun foo() = "Hello"
```

When using UMD or CommonJS, for example, your callsite could look like this:

```
alert(require('myModule').my.qualified.packagename.foo())
```

Or, in the case of using plain as a module system setting:

```
alert(myModule.my.qualified.packagename.foo());
```

@JsName annotation

In some cases (for example, to support overloads), the Kotlin compiler mangles the names of generated functions and attributes in JavaScript code. To control the generated names, you can use the `@JsName` annotation:

```
// Module 'kjs'
class Person(val name: String) {
    fun hello() {
        println("Hello $name!")
    }

    @JsName("helloWithGreeting")
    fun hello(greeting: String) {
        println("$greeting $name!")
    }
}
```

Now you can use this class from JavaScript in the following way:

```
// If necessary, import 'kjs' according to chosen module system
var person = new kjs.Person("Dmitry"); // refers to module 'kjs'
person.hello(); // prints "Hello Dmitry!"
person.helloWithGreeting("Servus"); // prints "Servus Dmitry!"
```

If we didn't specify the `@JsName` annotation, the name of the corresponding function would contain a suffix calculated from the function signature, for example `hello_61zpoe$`.

Note that there are some cases in which the Kotlin compiler does not apply mangling:

- external declarations are not mangled.
- Any overridden functions in non- external classes inheriting from external classes are not mangled.

The parameter of `@JsName` is required to be a constant string literal which is a valid identifier. The compiler will report an error on any attempt to pass non-identifier string to `@JsName`. The following example produces a compile-time

error:

```
@JsName("new C()") // error here
external fun newC()
```

@JsExport annotation

The @JsExport annotation is currently marked as experimental. Its design may change in future versions.

By applying the @JsExport annotation to a top-level declaration (like a class or function), you make the Kotlin declaration available from JavaScript. The annotation exports all nested declarations with the name given in Kotlin. It can also be applied on file-level using @file:JsExport.

To resolve ambiguities in exports (like overloads for functions with the same name), you can use the @JsExport annotation together with @JsName to specify the names for the generated and exported functions.

The @JsExport annotation is available in the current default compiler backend and the new [IR compiler backend](#). If you are targeting the IR compiler backend, you must use the @JsExport annotation to make your functions visible from Kotlin in the first place.

For multiplatform projects, @JsExport is available in common code as well. It only has an effect when compiling for the JavaScript target, and allows you to also export Kotlin declarations that are not platform specific.

Kotlin types in JavaScript

- Kotlin numeric types, except for kotlin.Long are mapped to JavaScript Number.
- kotlin.Char is mapped to JavaScript Number representing character code.
- Kotlin can't distinguish between numeric types at run time (except for kotlin.Long), so the following code works:

```
fun f() {
    val x: Int = 23
    val y: Any = x
    println(y as Float)
}
```

- Kotlin preserves overflow semantics for kotlin.Int, kotlin.Byte, kotlin.Short, kotlin.Char and kotlin.Long.
- kotlin.Long is not mapped to any JavaScript object, as there is no 64-bit integer number type in JavaScript. It is emulated by a Kotlin class.
- kotlin.String is mapped to JavaScript String.

- kotlin.Any is mapped to JavaScript Object (new Object(), {}, and so on).
- kotlin.Array is mapped to JavaScript Array.
- Kotlin collections (List, Set, Map, and so on) are not mapped to any specific JavaScript type.
- kotlin.Throwable is mapped to JavaScript Error.
- Kotlin preserves lazy object initialization in JavaScript.
- Kotlin does not implement lazy initialization of top-level properties in JavaScript.

Primitive arrays

Primitive array translation utilizes JavaScript TypedArray:

- kotlin.ByteArray, -.ShortArray, -.IntArray, -.FloatArray, and -.DoubleArray are mapped to JavaScript Int8Array, Int16Array, Int32Array, Float32Array, and Float64Array correspondingly.
- kotlin.BooleanArray is mapped to JavaScript Int8Array with a property \$type\$ == "BooleanArray"
- kotlin.CharArray is mapped to JavaScript Uint16Array with a property \$type\$ == "CharArray"
- kotlin.LongArray is mapped to JavaScript Array of kotlin.Long with a property \$type\$ == "LongArray".

JavaScript modules

You can compile your Kotlin projects to JavaScript modules for various popular module systems. We currently support the following configurations for JavaScript modules:

- [Unified Module Definitions \(UMD\)](#), which is compatible with both AMD and CommonJS. UMD modules are also able to be executed without being imported or when no module system is present. This is the default option for the browser and nodejs targets.
- [Asynchronous Module Definitions \(AMD\)](#), which is in particular used by the [RequireJS](#) library.
- [CommonJS](#), widely used by Node.js/npm (require function and module.exports object)
- Plain. Don't compile for any module system. You can access a module by its name in the global scope.

Browser targets

If you're targeting the browser and want to use a different module system than UMD, you can specify the desired module type in the webpackTask configuration block. For example, to switch to CommonJS, use:

```
kotlin {
```

```
js {
    browser {
        webpackTask {
            output.libraryTarget = "commonjs2"
        }
    }
    binaries.executable()
}
}
```

Webpack provides two different "flavors" of CommonJS, commonjs and commonjs2, which affect the way your declarations are made available. While in most cases, you probably want commonjs2, which adds the module.exports syntax to the generated library, you can also opt for the "pure" commonjs option, which implements the CommonJS specification exactly. To learn more about the difference between commonjs and commonjs2, check [here](#).

JavaScript libraries and Node.js files

If you are creating a library that will be consumed from JavaScript or a Node.js file, and want to use a different module system, the instructions are slightly different.

Choose the target module system

To select module kind, set the moduleKind compiler option in the Gradle build script.

Groovy

```
compileKotlinJs.kotlinOptions.moduleKind = "commonjs"
```

Kotlin

```
tasks.named<KotlinJsCompile>("compileKotlinJs").configure {
    kotlinOptions.moduleKind = "commonjs"
}
```

Available values are: umd (default), commonjs, amd, plain.

This is different from adjusting webpackTask.output.libraryTarget. The library target changes the output generated by webpack (after your code has already been compiled). kotlinOptions.moduleKind changes the output generated by the Kotlin compiler.

In the Kotlin Gradle DSL, there is also a shortcut for setting the CommonJS module kind:

```
kotlin {  
    js {  
        useCommonJs()  
        // ...  
    }  
}
```

@JsModule annotation

To tell Kotlin that an external class, package, function or property is a JavaScript module, you can use `@JsModule` annotation. Consider you have the following CommonJS module called "hello":

```
module.exports.sayHello = function(name) { alert("Hello, " + name); }
```

You should declare it like this in Kotlin:

```
@JsModule("hello")  
external fun sayHello(name: String)
```

Apply @JsModule to packages

Some JavaScript libraries export packages (namespaces) instead of functions and classes. In terms of JavaScript, it's an object that has members that are classes, functions and properties. Importing these packages as Kotlin objects often looks unnatural. The compiler can map imported JavaScript packages to Kotlin packages, using the following notation:

```
@file:JsModule("extModule")  
package ext.jspackage.name  
  
external fun foo()  
  
external class C
```

where the corresponding JavaScript module is declared like this:

```
module.exports = {  
    foo: { /* some code here */ },  
    C: { /* some code here */ }  
}
```

Files marked with `@file:JsModule` annotation can't declare non-external members. The example below produces a compile-time error:

```
@file:JsModule("extModule")
```

```

package ext.jspackage.name

external fun foo()

fun bar() = "!" + foo() + "!" // error here

```

Import deeper package hierarchies

In the previous example the JavaScript module exports a single package. However, some JavaScript libraries export multiple packages from within a module. This case is also supported by Kotlin, though you have to declare a new .kt file for each package you import.

For example, let's make the example a bit more complicated:

```

module.exports = {
  mylib: {
    pkg1: {
      foo: function() { /* some code here */ },
      bar: function() { /* some code here */ }
    },
    pkg2: {
      baz: function() { /* some code here */ }
    }
  }
}

```

To import this module in Kotlin, you have to write two Kotlin source files:

```

@file:JsModule("extModule")
@file:JsQualifier("mylib.pkg1")
package extlib.pkg1

external fun foo()

external fun bar()

```

and

```

@file:JsModule("extModule")
@file:JsQualifier("mylib.pkg2")
package extlib.pkg2

external fun baz()

```

@JsNonModule annotation

When a declaration is marked as @JsModule, you can't use it from Kotlin code when you don't compile it to a JavaScript module. Usually, developers distribute their libraries both as JavaScript modules and downloadable .js files that you can copy to your project's static resources and include via a <script> tag. To tell Kotlin that it's okay to use a

@JsModule declaration from a non-module environment, add the @JsNonModule annotation. For example, consider the following JavaScript code:

```
function topLevelSayHello(name) { alert("Hello, " + name); }
if (module && module.exports) {
    module.exports = topLevelSayHello;
}
```

You could describe it from Kotlin as follows:

```
@JsModule("hello")
@JsNonModule
@JsName("topLevelSayHello")
external fun sayHello(name: String)
```

Module system used by the Kotlin Standard Library

Kotlin is distributed with the Kotlin/JS standard library as a single file, which is itself compiled as an UMD module, so you can use it with any module system described above. While for most use cases of Kotlin/JS, it is recommended to use a Gradle dependency on kotlin-stdlib-js, it is also available on NPM as the [kotlin](#) package.

Kotlin/JS reflection

Kotlin/JS provides a limited support for the Kotlin [reflection API](#). The only supported parts of the API are:

- [class references](#) (::class).
- [KType](#) and [typeof\(\)](#) function.

Class references

The ::class syntax returns a reference to the class of an instance, or the class corresponding to the given type. In Kotlin/JS, the value of a ::class expression is a stripped-down [KClass](#) implementation that supports only:

- [simpleName](#) and [isInstance\(\)](#) members.
- [cast\(\)](#) and [safeCast\(\)](#) extension functions.

In addition to that, you can use [KClass.js](#) to access the [JsClass](#) instance corresponding to the class. The JsClass instance itself is a reference to the constructor function. This can be used to interoperate with JS functions that expect a reference to a constructor.

KType and typeOf()

The `typeOf()` function 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 `typeof()` function constructs an instance of `KType` for a given type. To use `typeof()` in your code, [opt in](#) to the experimental API of the standard library, for example, add `@OptIn(ExperimentalStdlibApi::class)` to your code.

The `KType` API is fully supported in Kotlin/JS except for Java-specific parts.

Example

Here is an example of the reflection usage in Kotlin/JS.

```
open class Shape
class Rectangle : Shape()

@OptIn(ExperimentalStdlibApi::class) // typeof() is experimental and requires an opt-in
inline fun <reified T> accessReifiedTypeArg() =
    println(typeOf<T>().toString())

fun main() {
    val s = Shape()
    val r = Rectangle()

    println(r::class.simpleName) // Prints "Rectangle"
    println(Shape::class.simpleName) // Prints "Shape"
    println(Shape::class.js.name) // Prints "Shape"

    println(Shape::class.isInstance(r)) // Prints "true"
    println(Rectangle::class.isInstance(s)) // Prints "false"
    val rShape = Shape::class.cast(r) // Casts a Rectangle "r" to Shape

    accessReifiedTypeArg<Rectangle>() // Accesses the type via typeof(). Prints "Rectangle"
}
```

Typesafe HTML DSL

The [kotlinx.html library](#) provides the ability to generate DOM elements using statically typed HTML builders (and besides JavaScript, it is even available on the JVM target!) To use the library, include the corresponding repository and dependency to our `build.gradle.kts` file:

```
repositories {
    // ...
    jcenter()
}

dependencies {
    implementation(kotlin("stdlib-js"))
```

```

implementation("org.jetbrains.kotlinx:kotlinx-html-js:0.7.1")
// ...
}

```

Once the dependency is included, you can access the different interfaces provided to generate the DOM. To render a headline, some text, and a link, the following snippet would be sufficient, for example:

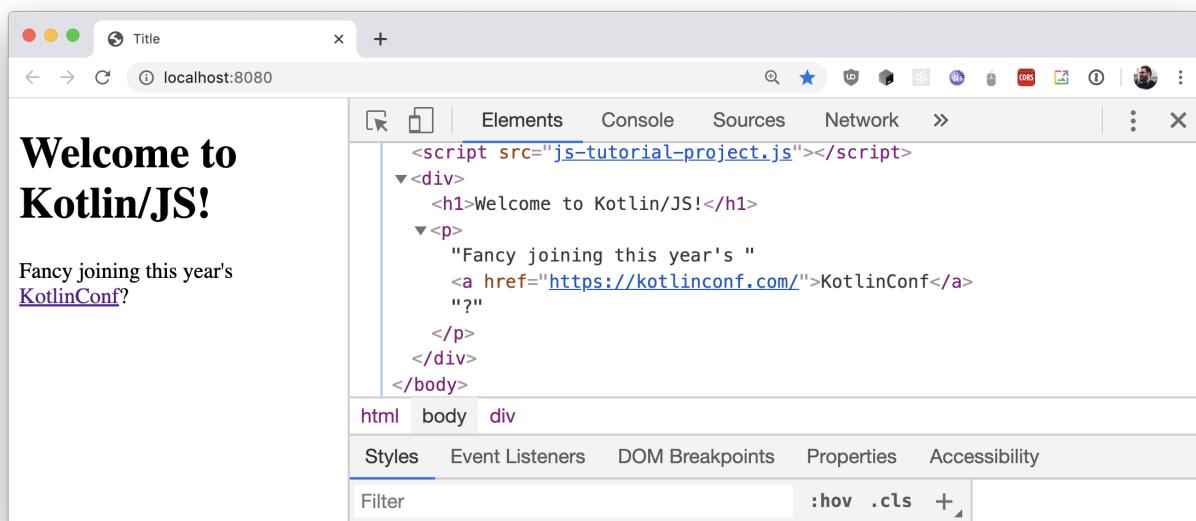
```

import kotlinx.browser.*
import kotlinx.html.*
import kotlinx.html.dom.*

fun main() {
    document.body!!.append.div {
        h1 {
            +"Welcome to Kotlin/JS!"
        }
        p {
            +"Fancy joining this year's "
            a("https://kotlinconf.com/") {
                +"KotlinConf"
            }
            ?
        }
    }
}

```

When running this example in the browser, the DOM will be assembled in a straightforward way. This is easily confirmed by checking the Elements of the website using the developer tools of our browser:



Rendering a website from kotlinx.html

To learn more about the kotlinx.html library, check out the [GitHub Wiki](#), where you can find more information about how

to [create elements](#) without adding them to the DOM, [binding to events](#) like onClick, and examples on how to [apply CSS classes](#) to your HTML elements, to name just a few.

Generation of external declarations with Dukat

Dukat is still [experimental](#). If you encounter any problems, please report them in Dukat's [issue tracker](#).

[Dukat](#) is a tool currently in development which allows the automatic conversion of TypeScript declaration files (.d.ts) into Kotlin external declarations. This aims to make it more comfortable to use libraries from the JavaScript ecosystem in a type-safe manner in Kotlin, reducing the need for manually writing external declarations and wrappers for JS libraries.

The Kotlin/JS Gradle plugin provides an integration with Dukat. When enabled, type-safe Kotlin external declarations are automatically generated for npm dependencies that provide TypeScript definitions. You have two different ways of selecting if and when Dukat should generate declarations: at build time, and manually via a Gradle task.

Generate external declarations at build time

The npm dependency function takes a third parameter after the package name and version: generateExternals. This allows you to control whether Dukat should generate declarations for a specific dependency:

Groovy

```
dependencies {  
    implementation(npm('decamelize', '4.0.0', true))  
}
```

Kotlin

```
dependencies {  
    implementation(npm("decamelize", "4.0.0", generateExternals = true))  
}
```

If the repository of the dependency you wish to use does not provide TypeScript definitions, you can also use types provided via the [DefinitelyTyped](#) repository. In this case, make sure you add npm dependencies for both your-package and @types/your-package (with generateExternals = true).

You can use the flag kotlin.js.generate.externals in your gradle.properties file to set the generator's behavior for all npm dependencies simultaneously. As usual, individual explicit settings take precedence over this general flag.

Manually generate external declarations via Gradle task

If you want to have full control over the declarations generated by Dukat, want to apply manual adjustments, or if you're running into trouble with the auto-generated externals, you can also trigger the creation of the declarations for all your npm dependencies manually via the Gradle task `generateExternals` (`jsGenerateExternals` with the multiplatform plugin). This will generate declarations in a directory titled `externals` in your project root. Here, you can review the generated code and copy any parts you would like to use to your source directories.

It is recommended to only provide external declarations manually in your source folder or enabling the generation of external declarations at build time for any single dependency. Doing both can result in resolution issues.

Hands-on labs for Kotlin/JS

Hands-on labs are long-form tutorials that help you get to know a technology by guiding you through a self-contained project related to a specific topic.

They include sample projects, which can serve as jumping-off points for your own projects, and contain useful snippets and patterns.

For Kotlin/JS, the following hands-on labs are currently available:

- [Building Web Applications with React and Kotlin/JS](#) guides you through the process of building a simple web application using the React framework, shows how a typesafe Kotlin DSL for HTML makes it convenient to build reactive DOM elements, and illustrates how to use third-party React components, and how to obtain information from APIs, while writing the whole application logic in pure Kotlin/JS.
- [Building a Full Stack Web App with Kotlin Multiplatform](#) teaches the concepts behind building an application that targets Kotlin/JVM and Kotlin/JS by building a client-server application that makes use of common code, serialization, and other multiplatform paradigms. It also provides a brief introduction into working with Ktor both as a server- and client-side framework.

We are continuously working on expanding the set of hands-on labs to make it as easy as possible for you to learn more about Kotlin/JS and adjacent technologies.

Get started with Kotlin/Native in IntelliJ IDEA

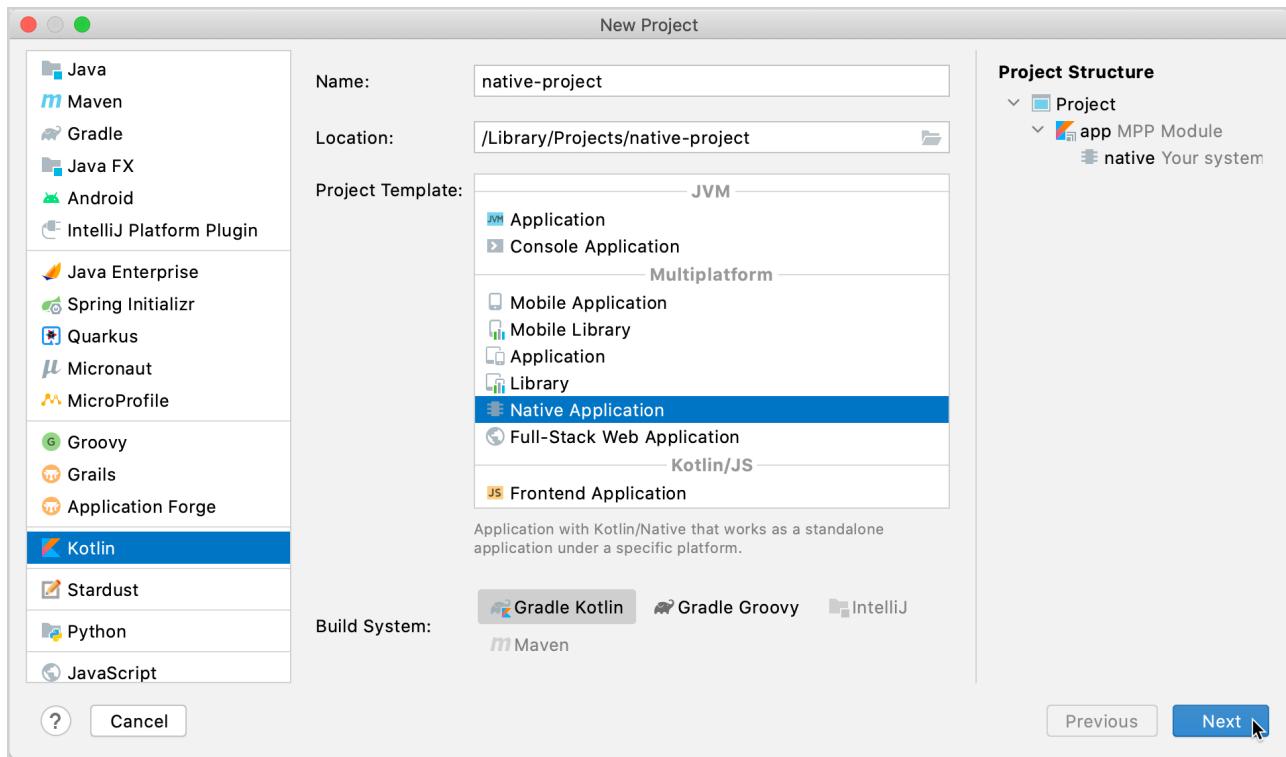
This tutorial demonstrates how to use IntelliJ IDEA for creating a Kotlin/Native application.

To get started, install the latest version of [IntelliJ IDEA](#). The tutorial is applicable to both IntelliJ IDEA Community Edition and the Ultimate Edition.

Create a new Kotlin/Native project in IntelliJ IDEA

1. In IntelliJ IDEA, select File | New | Project.

2. In the panel on the left, select Kotlin.
3. Enter a project name, select Native Application as the project template, and click Next.

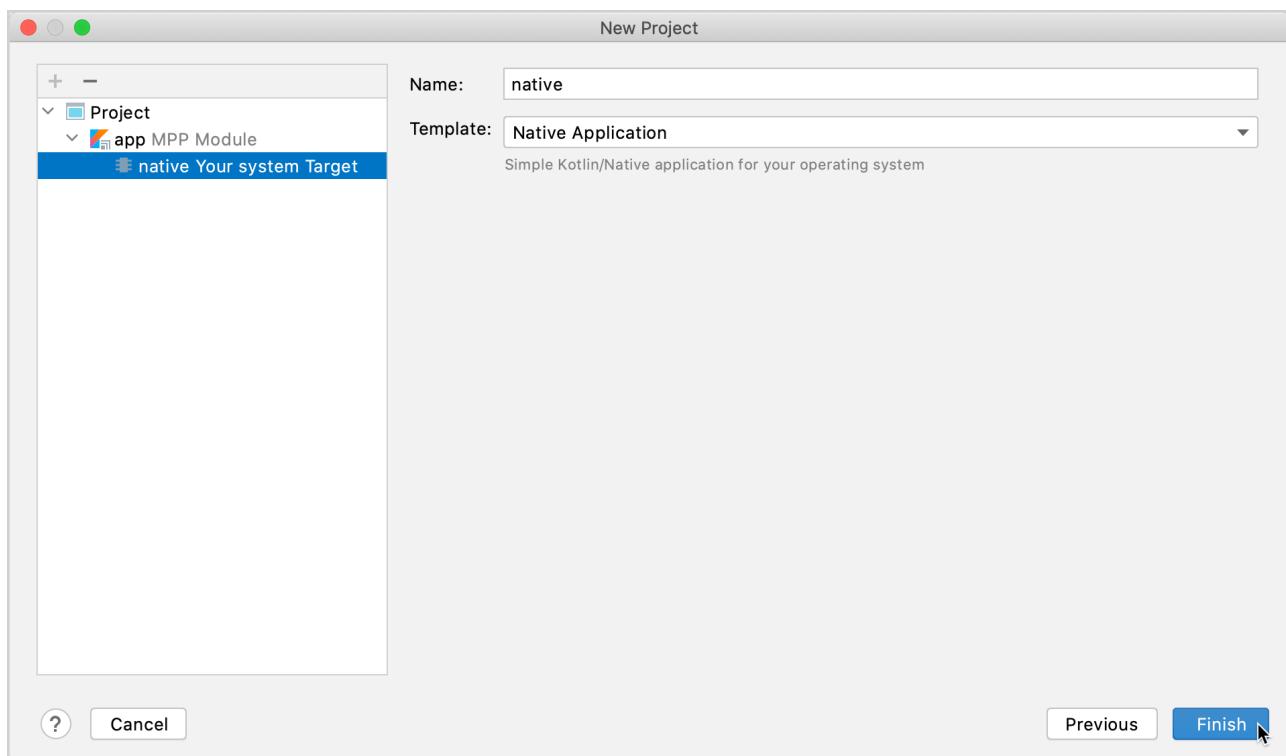


Create a native application

By default, your project will use Gradle with Kotlin DSL as the build system.

Kotlin/Native doesn't support Maven and IntelliJ IDEA native builder.

4. Accept the default configuration on the next screen and click Finish.



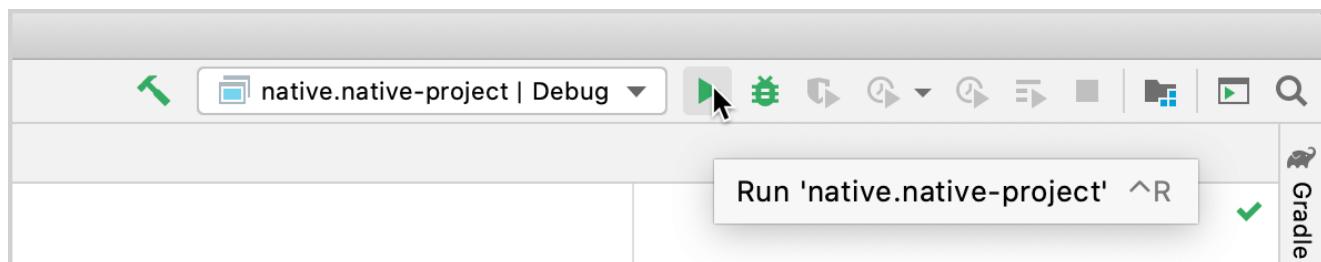
Configure a native application

Your project will open. By default, the wizard creates the necessary main.kt file with code that prints "Hello, Kotlin/Native!" to the standard output.

The build.gradle.kts file contains the project settings. Read more about these settings in the [Kotlin Multiplatform Gradle DSL reference](#).

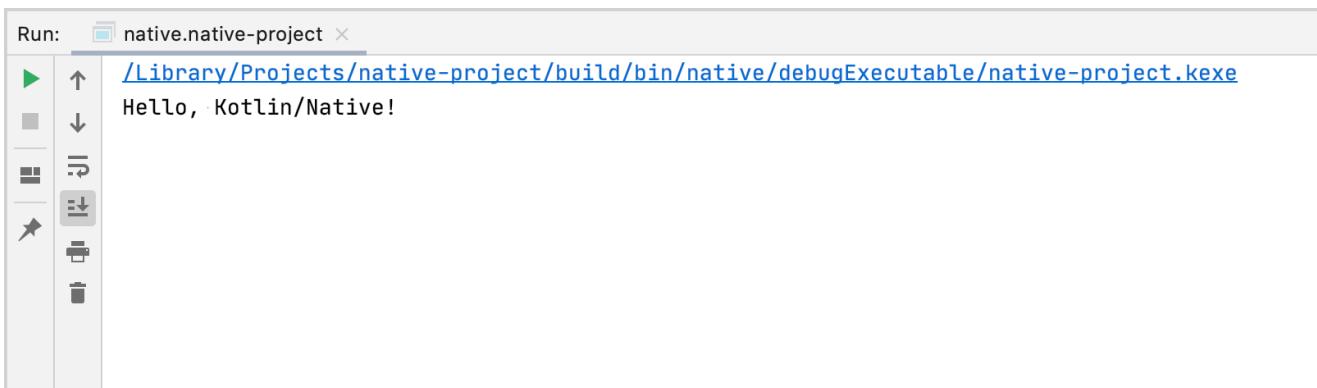
Run the application

Start the application by clicking Run next to the run configuration at the top of the screen.



Run the application

IntelliJ IDEA opens the Run tab and shows the output:



```
Run: native.native-project ×
/Libraries/Projects/native-project/build/bin/native/debugExecutable/native-project.kexe
Hello, Kotlin/Native!
```

Application output

Update the application

Count the letters in your name

1. Open the file main.kt in src/<your_app_name>Main/kotlin.

The src directory contains the Kotlin source files and resources. The file main.kt includes sample code that prints "Hello, Kotlin/Native!" using the `println()` function.

2. Add code to read the input. Use the `readLine()` function to read the input value and assign it to the name variable.

```
fun main() {
    // Read the input value.
    println("Hello, enter your name:")
    val name = readLine()
}
```

3. Eliminate the whitespaces and count the letters:

- Check that the provided name is not null with the `safe call operator ?..`
- Use the `replace()` function to remove the empty spaces in the name.
- Use the scope function `let` to run the function within the object context.
- Use a `string template` to insert your name length into the string by adding a dollar sign \$ and enclosing it in curly braces – `${it.length}`. It is the default name of a `lambda parameter`.

```
fun main() {
    // Read the input value.
    println("Hello, enter your name:")
    val name = readLine()
    // Count the letters in the name.
    name?.replace(" ", "")?.let {
        println("Your name contains ${it.length} letters")
    }
}
```

```
}
```

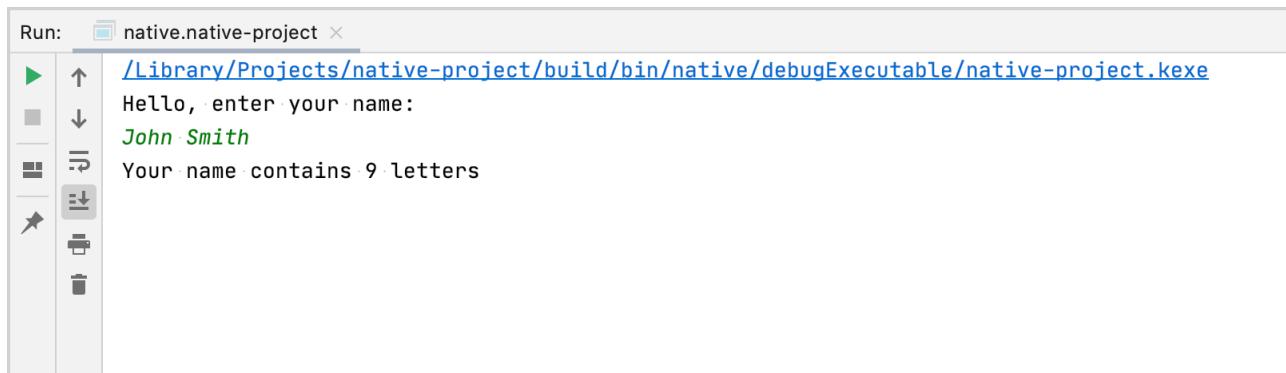
- Report a null value using the `error()` function after the Elvis operator ?:.

```
fun main() {  
    // Read the input value.  
    println("Hello, enter your name:")  
    val name = readLine()  
    // Count the letters in the name.  
    name?.replace(" ", "")?.let {  
        println("Your name contains ${it.length} letters")  
    } ?: error("Error while reading input from the terminal: the value can't be null.")  
}
```

- Save the changes and run the application.

IntelliJ IDEA opens the Run tab and shows the output.

- Enter your name and enjoy the result:



Application output

Count the unique letters in your name

- Open the file `main.kt` in `src/<your_app_name>Main/kotlin`.
- Declare the new extension function `countDistinctCharacters()` for `String`:
 - Convert the name to lowercase using the `lowercase()` function.
 - Convert the input string to a list of characters using the `toList()` function.
 - Select only the distinct characters in your name using the `distinct()` function.
 - Count the distinct characters using the `count()` function.

```
fun String.countDistinctCharacters() = lowercase().toList().distinct().count()
```

3. Use the `countDistinctCharacters()` function to count the unique letters in your name.

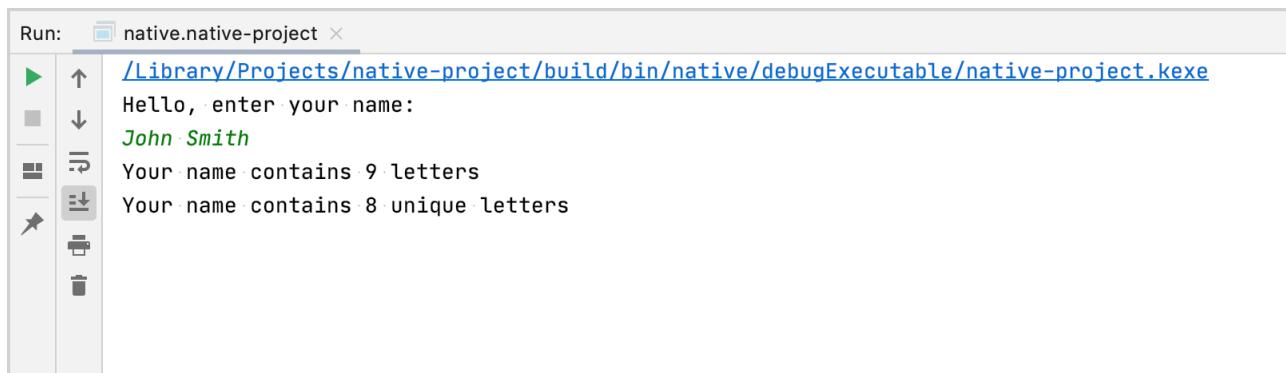
```
fun String.countDistinctCharacters() = lowercase().toList().distinct().count()

fun main() {
    // Read the input value.
    println("Hello, enter your name:")
    val name = readLine()
    // Count the letters in the name.
    name?.replace(" ", "")?.let {
        println("Your name contains ${it.length} letters")
        // Print the number of unique letters.
        println("Your name contains ${it.countDistinctCharacters()} unique letters")
    } ?: error("Error while reading input from the terminal: the value can't be null.")
}
```

4. Save the changes and run the application.

IntelliJ IDEA opens the Run tab and shows the output.

5. Enter your name and enjoy the result:



```
Run: native-native-project ×
/Library/Projects/native-native-project/build/bin/native/debugExecutable/native-project.kexe
Hello, enter your name:
John Smith
Your name contains 9 letters
Your name contains 8 unique letters
```

Application output

What's next?

Once you have created your first application, you can go to Kotlin hands-on labs and complete long-form tutorials on Kotlin/Native.

For Kotlin/Native, the following hands-on labs are currently available:

- [Learn about the concurrency model in Kotlin/Native](#) shows you how to build a command-line application and work with states in a multi-threaded environment.
- [Creating an HTTP Client in Kotlin/Native](#) explains how to create a native HTTP client and interoperate with C libraries.

Get started with Kotlin/Native using Gradle

Gradle is a build system that is very commonly used in the Java, Android, and other ecosystems. It is the default choice for Kotlin/Native and Multiplatform when it comes to build systems.

While most IDE's including [IntelliJ IDEA](#) can generate the corresponding Gradle file, we're going to take a look at how to create this manually, to have a better understanding of how things work under the covers. If you'd like to use the IDE, check out [Using IntelliJ IDEA](#).

Gradle supports two languages for build scripts:

- Groovy scripts in build.gradle files
- Kotlin scripts in build.gradle.kts files

The Groovy language is the first supported scripting language for Gradle, it leverages the power of dynamic typing and runtime features of the language. It is also possible to use Kotlin in Gradle scripts. Being a statically-typed language, it plays better with IDEs when it comes to compilation and error detection.

Either can be used and samples will show the syntax for both languages.

Create project files

First, create a project directory. Inside it, create build.gradle or build.gradle.kts Gradle build file with the following contents:

Groovy

```
plugins {  
    id 'org.jetbrains.kotlin.multiplatform' version '1.5.20'  
}  
  
repositories {  
    mavenCentral()  
}  
  
kotlin {  
    macosX64('native') { // on macOS  
        // linuxX64('native') // on Linux  
        // mingwX64('native') // on Windows  
        binaries {  
            executable()  
        }  
    }  
}  
  
wrapper {  
    gradleVersion = '6.7.1'  
    distributionType = 'BIN'  
}
```

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.5.20"
}

repositories {
    mavenCentral()
}

kotlin {
    macosX64("native") { // on macOS
        // linuxX64("native") // on Linux
        // mingwX64("native") // on Windows
        binaries {
            executable()
        }
    }
}

tasks.withType<Wrapper> {
    gradleVersion = "6.7.1"
    distributionType = Wrapper.DistributionType.BIN
}
```

The prepared project sources can be directly downloaded from Github:

- for macOS: [Groovy](#), [Kotlin](#)
- for Linux: [Groovy](#), [Kotlin](#)
- for Windows: [Groovy](#), [Kotlin](#)

Next, create an empty settings.gradle or settings.gradle.kts file in the project folder.

Depending on the target platform, different functions, such as macosX64, mingwX64, linuxX64, iosX64, are used for creating the Kotlin target. The function name is the platform for which you are compiling your code. These functions optionally take the target name as a parameter, which is "native" in our case. The specified target name is used to generate the source paths and task names in the project.

By convention, all sources are located in the src/<target name>[Main|Test]/kotlin folders, where main is for the source code and test is for tests. <target name> corresponds to the target platform (in this case native), as specified in the build file.

Create a folder src/nativeMain/kotlin and inside it place the file hello.kt with the following contents:

```
fun main() {
    println("Hello Kotlin/Native!")
}
```

Build the project

From the root project folder, execute the build by running

```
gradle nativeBinaries
```

This should create a folder build/bin/native with two subfolders debugExecutable and releaseExecutable with the corresponding binary. By default, the binary's name is the same as the project folder.

Open the project in an IDE

Any IDE that supports Gradle should allow for opening the project in the IDE. In the case of [IntelliJ IDEA](#), just open the project folder, and it will automatically detect it as Kotlin/Native project.

What's next?

Learn how to [write Gradle build scripts for real-life Kotlin/Native projects](#).

Get started with Kotlin/Native using the command-line compiler

Obtain the compiler

The Kotlin/Native compiler is available for macOS, Linux, and Windows. It is available as a command line tool and ships as part of the standard Kotlin distribution and can be downloaded from [GitHub Releases](#). It supports different targets including iOS (arm32, arm64, simulator x86_64), Windows (mingw32 and x86_64), Linux (x86_64, arm64, MIPS), macOS (x86_64), Raspberry PI, SMT32, WASM. [See the full list of targets here](#). While cross-platform compilation is possible, which means using one platform to compile for a different one, in this Kotlin case we'll be targeting the same platform we're compiling on.

While the output of the compiler does not have any dependencies or virtual machine requirements, the compiler itself requires [Java 1.8 or higher runtime](#).

Write "Hello Kotlin/Native" program

The application will print "Hello Kotlin/Native" on the standard output. In a working directory of choice, create a file named hello.kt and enter the following contents:

```
fun main() {
```

```
    println("Hello Kotlin/Native!")
}
```

Compile the code from the console

To compile the application use the [downloaded](#) compiler to execute the following command:

```
kotlinc-native hello.kt -o hello
```

The value of `-o` option specifies the name of the output file, so this call should generate a `hello.kexe` (Linux and macOS) or `hello.exe` (Windows) binary file. For the full list of available compiler options, see the [compiler options reference](#).

While compilation from the console seems to be easy and clear, it does not scale well for larger projects with hundreds of files and libraries. For real-world projects, it is recommended to use a [build system](#) and [IDE](#).

Concurrency in Kotlin/Native

Kotlin/Native runtime doesn't encourage a classical thread-oriented concurrency model with mutually exclusive code blocks and conditional variables, as this model is known to be error-prone and unreliable. Instead, we suggest a collection of alternative approaches, allowing you to use hardware concurrency and implement blocking IO. Those approaches are as follows, and they will be elaborated on in further sections:

- Workers with message passing
- Object subgraph ownership transfer
- Object subgraph freezing
- Object subgraph detachment
- Raw shared memory using C globals
- Atomic primitives and references
- Coroutines for blocking operations (not covered in this document)

Workers

Instead of threads Kotlin/Native runtime offers the concept of workers: concurrently executed control flow streams with an associated request queue. Workers are very similar to the actors in the Actor Model. A worker can exchange Kotlin objects with another worker, so that at any moment each mutable object is owned by a single worker, but ownership can be transferred. See section [Object transfer and freezing](#).

Once a worker is started with the Worker.start function call, it can be addressed with its own unique integer worker id. Other workers, or non-worker concurrency primitives, such as OS threads, can send a message to the worker with the execute call.

```
val future = execute(TransferMode.SAFE, { SomeDataForWorker() }) {
    // data returned by the second function argument comes to the
    // worker routine as 'input' parameter.
    input ->
    // Here we create an instance to be returned when someone consumes result future.
    WorkerResult(input.stringParam + " result")
}

future.consume {
    // Here we see result returned from routine above. Note that future object or
    // id could be transferred to another worker, so we don't have to consume future
    // in same execution context it was obtained.
    result -> println("result is $result")
}
```

The call to execute uses a function passed as its second parameter to produce an object subgraph (i.e. set of mutually referring objects) which is then passed as a whole to that worker, it is then no longer available to the thread that initiated the request. This property is checked if the first parameter is TransferMode.SAFE by graph traversal and is just assumed to be true, if it is TransferMode.UNSAFE. The last parameter to execute is a special Kotlin lambda, which is not allowed to capture any state, and is actually invoked in the target worker's context. Once processed, the result is transferred to whatever consumes it in the future, and it is attached to the object graph of that worker/thread.

If an object is transferred in UNSAFE mode and is still accessible from multiple concurrent executors, program will likely crash unexpectedly, so consider that last resort in optimizing, not a general purpose mechanism.

For a more complete example please refer to the [workers example](#) in the Kotlin/Native repository.

Object transfer and freezing

An important invariant that Kotlin/Native runtime maintains is that the object is either owned by a single thread/worker, or it is immutable (shared XOR mutable). This ensures that the same data has a single mutator, and so there is no need for locking to exist. To achieve such an invariant, we use the concept of not externally referred object subgraphs. This is a subgraph which has no external references from outside of the subgraph, which could be checked algorithmically with O(N) complexity (in ARC systems), where N is the number of elements in such a subgraph. Such subgraphs are usually produced as a result of a lambda expression, for example some builder, and may not contain objects, referred to externally.

Freezing is a runtime operation making a given object subgraph immutable, by modifying the object header so that future mutation attempts throw an InvalidMutabilityException. It is deep, so if an object has a pointer to other objects - transitive closure of such objects will be frozen. Freezing is a one way transformation, frozen objects cannot be unfrozen. Frozen objects have a nice property that due to their immutability, they can be freely shared between multiple workers/threads without breaking the "mutable XOR shared" invariant.

If an object is frozen it can be checked with an extension property isFrozen, and if it is, object sharing is allowed. Currently, Kotlin/Native runtime only freezes the enum objects after creation, although additional autofreezing of certain

provably immutable objects could be implemented in the future.

Object subgraph detachment

An object subgraph without external references can be disconnected using `DetachedObjectGraph<T>` to a `COpaquePointer` value, which could be stored in `void*` data, so the disconnected object subgraphs can be stored in a C data structure, and later attached back with `DetachedObjectGraph<T>.attach()` in an arbitrary thread or a worker. Combining it with raw memory sharing it allows side channel object transfer between concurrent threads, if the worker mechanisms are insufficient for a particular task. Note, that object detachment may require explicit leaving function holding object references and then performing cyclic garbage collection. For example, code like:

```
val graph = DetachedObjectGraph {
    val map = mutableMapOf<String, String>()
    for (entry in map.entries) {
        // ...
    }
    map
}
```

will not work as expected and will throw runtime exception, as there are uncollected cycles in the detached graph, while:

```
val graph = DetachedObjectGraph {
    val map = mutableMapOf<String, String>()
    for (entry in map.entries) {
        // ...
    }
    map
}().also {
    kotlin.native.internal.GC.collect()
}
```

will work properly, as holding references will be released, and then cyclic garbage affecting reference counter is collected.

Raw shared memory

Considering the strong ties between Kotlin/Native and C via interoperability, in conjunction with the other mechanisms mentioned above it is possible to build popular data structures, like concurrent hashmap or shared cache with Kotlin/Native. It is possible to rely upon shared C data, and store in it references to detached object subgraphs.

Consider the following .def file:

```
package = global
```

```

---  

typedef struct {  

    int version;  

    void* kotlinObject;  

} SharedData;  

  
SharedData sharedData;

```

After running the cinterop tool it can share Kotlin data in a versionized global structure, and interact with it from Kotlin transparently via autogenerated Kotlin like this:

```

class SharedData(rawPtr: NativePtr) : CStructVar(rawPtr) {  

    var version: Int  

    var kotlinObject: COpaquePointer?  

}

```

So in combination with the top-level variable declared above, it can allow looking at the same memory from different threads and building traditional concurrent structures with platform-specific synchronization primitives.

Global variables and singletons

Frequently, global variables are a source of unintended concurrency issues, so Kotlin/Native implements the following mechanisms to prevent the unintended sharing of state via global objects:

- global variables, unless specially marked, can be only accessed from the main thread (that is, the thread Kotlin/Native runtime was first initialized), if other thread access such a global, `IncorrectDereferenceException` is thrown
- for global variables marked with the `@kotlin.native.ThreadLocal` annotation each threads keeps thread-local copy, so changes are not visible between threads
- for global variables marked with the `@kotlin.native.SharedImmutable` annotation value is shared, but frozen before publishing, so each threads sees the same value
- singleton objects unless marked with `@kotlin.native.ThreadLocal` are frozen and shared, lazy values allowed, unless cyclic frozen structures were attempted to be created
- enums are always frozen

Combined, these mechanisms allow natural race-free programming with code reuse across platforms in MPP projects.

Atomic primitives and references

Kotlin/Native standard library provides primitives for safe working with concurrently mutable data, namely `AtomicInt`, `AtomicLong`, `AtomicNativePtr`, `AtomicReference` and `FreezableAtomicReference` in the package `kotlin.native.concurrent`. Atomic primitives allows concurrency-safe update operations, such as increment, decrement and compare-and-swap, along with value setters and getters. Atomic primitives are considered always frozen by the

runtime, and while their fields can be updated with the regular `field.value += 1`, it is not concurrency safe. Value must be changed using dedicated operations, so it is possible to perform concurrent-safe global counters and similar data structures.

Some algorithms require shared mutable references across the multiple workers, for example global mutable configuration could be implemented as an immutable instance of properties list atomically replaced with the new version on configuration update as the whole in a single transaction. This way no inconsistent configuration could be seen, and at the same time configuration could be updated as needed. To achieve such functionality Kotlin/Native runtime provides two related classes: `kotlin.native.concurrent.AtomicReference` and `kotlin.native.concurrent.FreezeableAtomicReference`. Atomic reference holds reference to a frozen or immutable object, and its value could be updated by set or compare-and-swap operation. Thus, dedicated set of objects could be used to create mutable shared object graphs (of immutable objects). Cycles in the shared memory could be created using atomic references. Kotlin/Native runtime doesn't support garbage collecting cyclic data when reference cycle goes through `AtomicReference` or frozen `FreezeableAtomicReference`. So to avoid memory leaks atomic references that are potentially parts of shared cyclic data should be zeroed out once no longer needed.

If atomic reference value is attempted to be set to non-frozen value runtime exception is thrown.

Freezable atomic reference is similar to the regular atomic reference, but until frozen behaves like regular box for a reference. After freezing it behaves like an atomic reference, and can only hold a reference to a frozen object.

Immutability in Kotlin/Native

Kotlin/Native implements strict mutability checks, ensuring the important invariant that the object is either immutable or accessible from the single thread at that moment in time (mutable XOR global).

Immutability is a runtime property in Kotlin/Native, and can be applied to an arbitrary object subgraph using the `kotlin.native.concurrent.freeze` function. It makes all the objects reachable from the given one immutable, such a transition is a one-way operation (i.e., objects cannot be unfrozen later). Some naturally immutable objects such as `kotlin.String`, `kotlin.Int`, and other primitive types, along with `AtomicInt` and `AtomicReference` are frozen by default. If a mutating operation is applied to a frozen object, an `InvalidMutabilityException` is thrown.

To achieve mutable XOR global invariant, all globally visible state (currently, object singletons and enums) are automatically frozen. If object freezing is not desired, a `kotlin.native.ThreadLocal` annotation can be used, which will make the object state thread local, and so, mutable (but the changed state is not visible to other threads).

Top level/global variables of non-primitive types are by default accessible in the main thread (i.e., the thread which initialized Kotlin/Native runtime first) only. Access from another thread will lead to an `IncorrectDereferenceException` being thrown. To make such variables accessible in other threads, you can use either the `@ThreadLocal` annotation, and mark the value thread local or `@SharedImmutable`, which will make the value frozen and accessible from other threads.

Class `AtomicReference` can be used to publish the changed frozen state to other threads, and so build patterns like shared caches.

Kotlin/Native libraries

Kotlin compiler specifics

To produce a library with the Kotlin/Native compiler use the `-produce library` or `-p library` flag. For example:

```
$ kotlinc-native foo.kt -p library -o bar
```

This command will produce a `bar.klib` with the compiled contents of `foo.kt`.

To link to a library use the `-library <name>` or `-l <name>` flag. For example:

```
$ kotlinc-native qux.kt -l bar
```

This command will produce a `program.kexe` out of `qux.kt` and `bar.klib`

cinterop tool specifics

The cinterop tool produces `.klib` wrappers for native libraries as its main output. For example, using the simple `libgit2.def` native library definition file provided in your Kotlin/Native distribution

```
$ cinterop -def samples/gitchurn/src/nativeInterop/cinterop/libgit2.def -compiler-option -I/usr/local/include -o libgit2
```

we will obtain `libgit2.klib`.

See more details in [C Interop](#)

klib utility

The `klib` library management utility allows you to inspect and install the libraries.

The following commands are available:

- `content` – list library contents:

```
$ klib contents <name>
```

- `info` – inspect the bookkeeping details of the library

```
$ klib info <name>
```

- install – install the library to the default location use

```
$ klib install <name>
```

- remove – remove the library from the default repository use

```
$ klib remove <name>
```

All of the above commands accept an additional -repository <directory> argument for specifying a repository different to the default one.

```
$ klib <command> <name> -repository <directory>
```

Several examples

First let's create a library. Place the tiny library source code into kotlinizer.kt:

```
package kotlinizer
val String.kotlinized
    get() = "Kotlin $this"
```

```
$ kotlinc-native kotlinizer.kt -p library -o kotlinizer
```

The library has been created in the current directory:

```
$ ls kotlinizer.klib
kotlinizer.klib
```

Now let's check out the contents of the library:

```
$ klib contents kotlinizer
```

You can install kotlinizer to the default repository:

```
$ klib install kotlinizer
```

Remove any traces of it from the current directory:

```
$ rm kotlinizer.klib
```

Create a very short program and place it into a use.kt:

```
import kotlinizer.*  
  
fun main(args: Array<String>) {  
    println("Hello, ${"world".kotlinized}!")  
}
```

Now compile the program linking with the library you have just created:

```
$ kotlinc-native use.kt -l kotlinizer -o kohello
```

And run the program:

```
$ ./kohello.kexe  
Hello, Kotlin world!
```

Have fun!

Advanced topics

Library search sequence

When given a -library foo flag, the compiler searches the foo library in the following order:

- Current compilation directory or an absolute path.
- All repositories specified with -repo flag.
- Libraries installed in the default repository (For now the default is ~/.konan, however it could be changed by setting KONAN_DATA_DIR environment variable).
- Libraries installed in \$installation/klib directory.

Library format

Kotlin/Native libraries are zip files containing a predefined directory structure, with the following layout:

foo.klib when unpacked as foo/ gives us:

```
- foo/  
- $component_name/  
- ir/  
  - Serialized Kotlin IR.  
- targets/  
  - $platform/
```

```
- kotlin/
  - Kotlin compiled to LLVM bitcode.
- native/
  - Bitcode files of additional native objects.
- $another_platform/
  - There can be several platform specific kotlin and native pairs.
- linkdata/
  - A set of ProtoBuf files with serialized linkage metadata.
- resources/
  - General resources such as images. (Not used yet).
- manifest - A file in the java property format describing the library.
```

An example layout can be found in klib/stdcil directory of your installation.

Platform libraries

To provide access to user's native operating system services, Kotlin/Native distribution includes a set of prebuilt libraries specific to each target. We call them Platform Libraries.

POSIX bindings

For all Unix- or Windows-based targets (including Android and iOS targets) we provide the posix platform lib. It contains bindings to platform's implementation of the [POSIX standard](#).

To use the library, just import it:

```
import platform.posix.*
```

The only target for which it is not available is [WebAssembly](#).

Note that the content of platform.posix is NOT identical on different platforms, in the same way as different POSIX implementations are a little different.

Popular native libraries

There are many more platform libraries available for host and cross-compilation targets. Kotlin/Native distribution provides access to OpenGL, zlib and other popular native libraries on applicable platforms.

On Apple platforms, objc library is provided for interoperability with [Objective-C](#).

Inspect the contents of dist/klib/platform/\$target of the distribution for the details.

Availability by default

The packages from platform libraries are available by default. No special link flags need to be specified to use them. Kotlin/Native compiler automatically detects which of the platform libraries have been accessed and automatically links the needed libraries.

On the other hand, the platform libs in the distribution are merely just wrappers and bindings to the native libraries. That means the native libraries themselves (.so, .a, .dylib, .dll etc) should be installed on the machine.

Examples

The Kotlin/Native repository provides a wide spectrum of examples demonstrating the use of platform libraries. See [samples](#) for details.

Kotlin/Native as a dynamic library – tutorial

Learn how you can use the Kotlin/Native code from existing native applications or libraries. For this, you need to compile the Kotlin code into a dynamic library, .so, .dylib, and .dll.

Kotlin/Native also has tight integration with Apple technologies. The [Kotlin/Native as an Apple Framework](#) tutorial explains how to compile Kotlin code into a framework for Swift and Objective-C.

In this tutorial, you will:

- [Compile a Kotlin code to a dynamic library](#)
- [Examine generated C headers](#)
- [Use the Kotlin dynamic library from C](#)
- Compile and run the example on [Linux and Mac](#) and [Windows](#)

Create a Kotlin library

Kotlin/Native compiler can produce a dynamic library out of the Kotlin code. A dynamic library often comes with a header file, a .h file, which you will use to call the compiled code from C.

The best way to understand these techniques is to try them out. First, create a first tiny Kotlin library and use it from a C program.

Start by creating a library file in Kotlin and save it as hello.kt:

```
package example

object Object {
    val field = "A"
}
```

```

class Clazz {
    fun memberFunction(p: Int): ULong = 42UL
}

fun forIntegers(b: Byte, s: Short, i: UInt, l: Long) { }
fun forFloats(f: Float, d: Double) { }

fun strings(str: String) : String? {
    return "That is '$str' from C"
}

val globalString = "A global String"

```

While it is possible to use the command line, either directly or by combining it with a script file (such as .sh or .bat file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the [A Basic Kotlin/Native Application](#) tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and [multiplatform](#) builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following build.gradle.kts Gradle build file:

Groovy

```

plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.5.20'
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64("native") { // on Linux
        // macosX64("native") { // on macOS
        // mingwX64("native") { //on Windows
            binaries {
                sharedLib {
                    baseName = "native" // on Linux and macOS
                    // baseName = "libnative" //on Windows
                }
            }
        }
    }
}

```

```

wrapper {
    gradleVersion = "6.7.1"
    distributionType = "ALL"
}

```

Kotlin

```

plugins {
    kotlin("multiplatform") version "1.5.20"
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64("native") { // on Linux
        // macosX64("native") { // on macOS
        // mingwX64("native") { // on Windows
            binaries {
                sharedLib {
                    baseName = "native" // on Linux and macOS
                    // baseName = "libnative" // on Windows
                }
            }
        }
    }
}

tasks.wrapper {
    gradleVersion = "6.7.1"
    distributionType = Wrapper.DistributionType.ALL
}

```

The prepared project sources can be directly downloaded from Github:

- for macOS: [Groovy](#), [Kotlin](#)
- for Linux: [Groovy](#), [Kotlin](#)
- for Windows: [Groovy](#), [Kotlin](#)

Move the sources file into the src/nativeMain/kotlin folder under the project. This is the default path, for where sources are located, when the [kotlin-multiplatform](#) plugin is used. Use the following block to instruct and configure the project to generate a dynamic or shared library:

```

binaries {
    sharedLib {
        baseName = "native" // on Linux and macOS
        // baseName = "libnative" // on Windows
    }
}

```

The libnative is used as the library name, the generated header file name prefix. It is also prefixes all declarations in the header file.

Now you can [open the project in IntelliJ IDEA](#) and see how to fix the example project. While doing this, we'll examine how C functions are mapped into Kotlin/Native declarations.

Run the linkNative Gradle task to build the library in the IDE or by calling the following console command:

```
./gradlew linkNative
```

The build generates the following files under the build/bin/native/debugShared folder, depending on the host OS:

- macOS: libnative_api.h and libnative.dylib
- Linux: libnative_api.h and libnative.so
- Windows: libnative_api.h, libnative_symbols.def and libnative.dll

The same rules are used by the Kotlin/Native compiler to generate the .h file for all platforms.

Let's check out the C API of our Kotlin library.

Generated headers file

In the libnative_api.h, you'll find the following code. Let's discuss the code in parts to make it easier to understand.

The way Kotlin/Native exports symbols is subject to change without notice.

The very first part contains the standard C/C++ header and footer:

```
#ifndef KONAN_DEMO_H
#define KONAN_DEMO_H
#ifndef __cplusplus
extern "C" {
#endif

/// THE REST OF THE GENERATED CODE GOES HERE

#ifndef __cplusplus
} /* extern "C" */
#endif
#endif /* KONAN_DEMO_H */
```

After the rituals in the libnative_api.h, there is a block with the common type definitions:

```
#ifndef __cplusplus
typedef bool libnative_KBoolean;
```

```

#else
typedef _Bool libnative_KBoolean;
#endif
typedef unsigned short libnative_KChar;
typedef signed char libnative_KByte;
typedef short libnative_KShort;
typedef int libnative_KInt;
typedef long long libnative_KLong;
typedef unsigned char libnative_KUByte;
typedef unsigned short libnative_KUShort;
typedef unsigned int libnative_KUInt;
typedef unsigned long long libnative_KULong;
typedef float libnative_KFloat;
typedef double libnative_KDouble;
typedef void* libnative_KNativePtr;

```

Kotlin uses the libnative_ prefix for all declarations in the created libnative_api.h file. Let's present the mapping of the types in a more readable way:

Kotlin Define	C Type
libnative_KBoolean	bool or _Bool
libnative_KChar	unsigned short
libnative_KByte	signed char
libnative_KShort	short
libnative_KInt	int
libnative_KLong	long long
libnative_KUByte	unsigned char
libnative_KUShort	unsigned short
libnative_KUInt	unsigned int
libnative_KULong	unsigned long long
libnative_KFloat	float
libnative_KDouble	double
libnative_KNativePtr	void*

The definitions part shows how Kotlin primitive types map into C primitive types. The reverse mapping is described in the [Mapping primitive data types from C](#) tutorial.

The next part of the libnative_api.h file contains definitions of the types that are used in the library:

```
struct libnative_KType;
typedef struct libnative_KType libnative_KType;

typedef struct {
    libnative_KNativePtr pinned;
} libnative_kref_example_Object;

typedef struct {
    libnative_KNativePtr pinned;
} libnative_kref_example_Clazz;
```

The `typedef struct { .. } TYPE_NAME` syntax is used in C language to declare a structure. [This thread](#) on Stackoverflow provides more explanations of that pattern.

As you can see from these definitions, the Kotlin object Object is mapped into libnative_kref_example_Object, and Clazz is mapped into libnative_kref_example_Clazz. Both structs contain nothing but the pinned field with a pointer, the field type libnative_KNativePtr is defined as `void*` above.

There is no namespaces support in C, so the Kotlin/Native compiler generates long names to avoid any possible clashes with other symbols in the existing native project.

A significant part of the definitions goes in the libnative_api.h file. It includes the definition of our Kotlin/Native library world:

```
typedef struct {
    /* Service functions. */
    void (*DisposeStablePointer)(libnative_KNativePtr ptr);
    void (*DisposeString)(const char* string);
    libnative_KBoolean (*IsInstance)(libnative_KNativePtr ref, const libnative_KType* type);

    /* User functions. */
    struct {
        struct {
            struct {
                void (*forIntegers)(libnative_KByte b, libnative_KShort s, libnative_KUInt i,
libnative_KLong l);
                void (*forFloats)(libnative_KFloat f, libnative_KDouble d);
                const char* (*strings)(const char* str);
                const char* (*get_globalString)();
                struct {
                    libnative_KType* (*_type)(void);
                    libnative_kref_example_Object (*_instance)();
                    const char* (*get_field)(libnative_kref_example_Object thiz);
                } Object;
                struct {
                    libnative_KType* (*_type)(void);
                    libnative_kref_example_Clazz (*Clazz)();
                    libnative_KULong (*memberFunction)(libnative_kref_example_Clazz thiz, libnative_KInt
p);
                }
            }
        }
    }
}
```

```

        } Clazz;
    } example;
} root;
} kotlin;
} libnative_ExportedSymbols;

```

The code uses anonymous structure declarations. The code struct { .. } foo declares a field in the outer struct of that anonymous structure type, the type with no name.

C does not support objects either. People use function pointers to mimic object semantics. A function pointer is declared as follows RETURN_TYPE (* FIELD_NAME)(PARAMETERS). It is tricky to read, but we should be able to see function pointer fields in the structures above.

Runtime functions

The code reads as follows. You have the libnative_ExportedSymbols structure, which defines all the functions that Kotlin/Native and our library provides us. It uses nested anonymous structures heavily to mimic packages. The libnative_ prefix comes from the library name.

The libnative_ExportedSymbols structure contains several helper functions:

```

void (*DisposeStablePointer)(libnative_KNativePtr ptr);
void (*DisposeString)(const char* string);
libnative_KBoolean (*IsInstance)(libnative_KNativePtr ref, const libnative_KType* type);

```

These functions deal with Kotlin/Native objects. Call the DisposeStablePointer to release a Kotlin object and DisposeString to release a Kotlin String, which has the char* type in C. It is possible to use the IsInstance function to check if a Kotlin type or a libnative_KNativePtr is an instance of another type. The actual set of operations generated depends on the actual usages.

Kotlin/Native has garbage collection, but it does not help us deal with Kotlin objects from the C language. Kotlin/Native has interop with Objective-C and Swift and integrates with their reference counters. The [Objective-C Interop](#) documentation article contains more details on it. Also, there is the tutorial [Kotlin/Native as an Apple Framework](#).

Your library functions

Let's take a look at the kotlin.root.example field, it mimics the package structure of our Kotlin code with a kotlin.root. prefix.

There is a kotlin.root.example.Clazz field that represents the Clazz from Kotlin. The Clazz#memberFunction is accessible with the memberFunction field. The only difference is that the memberFunction accepts a this reference as the first parameter. The C language does not support objects, and this is the reason to pass a this pointer explicitly.

There is a constructor in the Clazz field (aka kotlin.root.example.Clazz.Clazz), which is the constructor function to create an instance of the Clazz.

Kotlin object Object is accessible as kotlin.root.example.Object. There is the _instance function to get the only instance of the object.

Properties are translated into functions. The get_ and set_ prefix is used to name the getter and the setter functions respectively. For example, the read-only property globalString from Kotlin is turned into a get_globalString function in C.

Global functions forInts, forFloats, or strings are turned into the functions pointers in the kotlin.root.example anonymous struct.

Entry point

You can see how the API is created. To start with, you need to initialize the libnative_ExportedSymbols structure. Let's take a look at the latest part of the libnative_api.h for this:

```
extern libnative_ExportedSymbols* libnative_symbols(void);
```

The function libnative_symbols allows you to open the way from the native code to the Kotlin/Native library. This is the entry point you'll use. The library name is used as a prefix for the function name.

Kotlin/Native object references do not support multi-threaded access. Hosting the returned libnative_ExportedSymbols* pointer per thread might be necessary.

Use generated headers from C

The usage from C is straightforward and uncomplicated. Create a main.c file with the following code:

```
#include "libnative_api.h"
#include "stdio.h"

int main(int argc, char** argv) {
    //obtain reference for calling Kotlin/Native functions
    libnative_ExportedSymbols* lib = libnative_symbols();

    lib->kotlin.root.example.forIntegers(1, 2, 3, 4);
    lib->kotlin.root.example.forFloats(1.0f, 2.0);

    //use C and Kotlin/Native strings
    const char* str = "Hello from Native!";
    const char* response = lib->kotlin.root.example.strings(str);
    printf("in: %s\nout:%s\n", str, response);
    lib->DisposeString(response);

    //create Kotlin object instance
    libnative_kref_example_Class newInstance = lib->kotlin.root.example.Class.Class();
    long x = lib->kotlin.root.example.Class.memberFunction(newInstance, 42);
    lib->DisposeStablePointer(newInstance.pinned);

    printf("DemoClass returned %ld\n", x);
```

```
    return 0;  
}
```

Compile and run the example on Linux and macOS

On macOS 10.13 with Xcode, compile the C code and link it with the dynamic library with the following command:

```
clang main.c libnative.dylib
```

On Linux call a similar command:

```
gcc main.c libnative.so
```

The compiler generates an executable called a.out. Run it to see in action the Kotlin code being executed from C library. On Linux, you'll need to include . into the LD_LIBRARY_PATH to let the application know to load the libnative.so library from the current folder.

Compile and run the example on Windows

To start with, you'll need a Microsoft Visual C++ compiler installed that supports a x64_64 target. The easiest way to do this is to have a version of Microsoft Visual Studio installed on a Windows machine.

In this example, you'll be using the x64 Native Tools Command Prompt <VERSION> console. You'll see the shortcut to open the console in the start menu. It comes with a Microsoft Visual Studio package.

On Windows, Dynamic libraries are included either via a generated static library wrapper or with manual code, which deals with the [LoadLibrary](#) or similar Win32API functions. Follow the first option and generate the static wrapper library for the libnative.dll as described below.

Call lib.exe from the toolchain to generate the static library wrapper libnative.lib that automates the DLL usage from the code:

```
lib /def:libnative_symbols.def /out:libnative.lib
```

Now you are ready to compile our main.c into an executable. Include the generated libnative.lib into the build command and start:

```
cl.exe main.c libnative.lib
```

The command produces the main.exe file, which you can run.

Next steps

Dynamic libraries are the main way to use Kotlin code from existing programs. You can use them to share your code with many platforms or languages, including JVM, [Python](#), iOS, Android, and others.

Kotlin/Native also has tight integration with Objective-C and Swift. It is covered in the [Kotlin/Native as an Apple Framework](#) tutorial.

Interoperability with C

Kotlin/Native follows the general tradition of Kotlin to provide excellent existing platform software interoperability. In the case of a native platform, the most important interoperability target is a C library. So Kotlin/Native comes with a cinterop tool, which can be used to quickly generate everything needed to interact with an external library.

The following workflow is expected when interacting with the native library:

1. Create a .def file describing what to include into bindings.
2. Use the cinterop tool to produce Kotlin bindings.
3. Run the Kotlin/Native compiler on an application to produce the final executable.

The interoperability tool analyses C headers and produces a "natural" mapping of the types, functions, and constants into the Kotlin world. The generated stubs can be imported into an IDE for the purpose of code completion and navigation.

Interoperability with Swift/Objective-C is provided too and covered in [Objective-C interop](#).

Platform libraries

Note that in many cases there's no need to use custom interoperability library creation mechanisms described below, as for APIs available on the platform standardized bindings called [platform libraries](#) could be used. For example, POSIX on Linux/macOS platforms, Win32 on Windows platform, or Apple frameworks on macOS/iOS are available this way.

Simple example

Install libgit2 and prepare stubs for the git library:

```
cd samples/gitchurn  
../../dist/bin/cinterop -def src/nativeInterop/cinterop/libgit2.def \  
-compiler-option -I/usr/local/include -o libgit2
```

Compile the client:

```
../../dist/bin/kotlinc src/gitChurnMain/kotlin \
    -library libgit2 -o GitChurn
```

Run the client:

```
./GitChurn.kexe ...
```

Create bindings for a new library

To create bindings for a new library, start from creating a .def file. Structurally it's a simple property file, which looks like this:

```
headers = png.h
headerFilter = png.h
package = png
```

Then run the cinterop tool with something like this (note that for host libraries that are not included in the sysroot search paths, headers may be needed):

```
cinterop -def png.def -compiler-option -I/usr/local/include -o png
```

This command will produce a png.klib compiled library and png-build/kotlin directory containing Kotlin source code for the library.

If the behavior for a certain platform needs to be modified, you can use a format like compilerOpts.osx or compilerOpts.linux to provide platform-specific values to the options.

Note that the generated bindings are generally platform-specific, so if you are developing for multiple targets, the bindings need to be regenerated.

After the generation of bindings, they can be used by the IDE as a proxy view of the native library.

For a typical Unix library with a config script, the compilerOpts will likely contain the output of a config script with the --cflags flag (maybe without exact paths).

The output of a config script with --libs will be passed as a -linkedArgs kotlinc flag value (quoted) when compiling.

Select library headers

When library headers are imported to a C program with the #include directive, all of the headers included by these headers are also included in the program. So all header dependencies are included in generated stubs as well.

This behavior is correct but it can be very inconvenient for some libraries. So it is possible to specify in the .def file

which of the included headers are to be imported. The separate declarations from other headers can also be imported in case of direct dependencies.

Filter headers by globs

It is possible to filter headers by globs. The headerFilter property value from the .def file is treated as a space-separated list of globs. If the included header matches any of the globs, then the declarations from this header are included into the bindings.

The globs are applied to the header paths relative to the appropriate include path elements, e.g. time.h or curl/curl.h. So if the library is usually included with #include <SomeLibrary/Header.h>, then it would probably be correct to filter headers with

```
headerFilter = SomeLibrary/**
```

If a headerFilter is not specified, then all headers are included.

Filter headers by module maps

Some libraries have proper module.modulemap or module.map files in their headers. For example, macOS and iOS system libraries and frameworks do. The [module map file](#) describes the correspondence between header files and modules. When the module maps are available, the headers from the modules that are not included directly can be filtered out using the experimental excludeDependentModules option of the .def file:

```
headers = OpenGL/gl.h OpenGL/glu.h GLUT/glut.h
compilerOpts = -framework OpenGL -framework GLUT
excludeDependentModules = true
```

When both excludeDependentModules and headerFilter are used, they are applied as an intersection.

C compiler and linker options

Options passed to the C compiler (used to analyze headers, such as preprocessor definitions) and the linker (used to link final executables) can be passed in the definition file as compilerOpts and linkerOpts respectively. For example:

```
compilerOpts = -DFOO=bar
linkerOpts = -lpng
```

Target-specific options only applicable to the certain target can be specified as well:

```
compilerOpts = -DBAR=bar
compilerOpts.linux_x64 = -DFOO=foo1
compilerOpts.mac_x64 = -DFOO=foo2
```

With such a configuration, C headers will be analyzed with -DBAR=bar -DFOO=foo1 on Linux and with -DBAR=bar -

DFOO=foo2 on macOS . Note that any definition file option can have both common and the platform-specific part.

Add custom declarations

Sometimes it is required to add custom C declarations to the library before generating bindings (e.g., for [macros](#)).

Instead of creating an additional header file with these declarations, you can include them directly to the end of the .def file, after a separating line, containing only the separator sequence ---:

```
headers = errno.h

---

static inline int getErrno() {
    return errno;
}
```

Note that this part of the .def file is treated as part of the header file, so functions with the body should be declared as static. The declarations are parsed after including the files from the headers list.

Include a static library in your klib

Sometimes it is more convenient to ship a static library with your product, rather than assume it is available within the user's environment. To include a static library into .klib use staticLibraries and libraryPaths clauses. For example:

```
headers = foo.h
staticLibraries = libfoo.a
libraryPaths = /opt/local/lib /usr/local/opt/curl/lib
```

When given the above snippet the cinterop tool will search libfoo.a in /opt/local/lib and /usr/local/opt/curl/lib, and if it is found include the library binary into klib.

When using such klib in your program, the library is linked automatically.

Bindings

Basic interop types

All the supported C types have corresponding representations in Kotlin:

- Signed, unsigned integral, and floating point types are mapped to their Kotlin counterpart with the same width.
- Pointers and arrays are mapped to CPointer<T>?.
- Enums can be mapped to either Kotlin enum or integral values, depending on heuristics and the [definition file hints](#).
- Structs and unions are mapped to types having fields available via the dot notation, i.e. someStructInstance.field1.

- `typedef` are represented as `typealias`.

Also, any C type has the Kotlin type representing the lvalue of this type, i.e., the value located in memory rather than a simple immutable self-contained value. Think C++ references, as a similar concept. For structs (and `typedef`s to structs) this representation is the main one and has the same name as the struct itself, for Kotlin enums it is named `${type}Var`, for `CPointer<T>` it is `CPointerVar<T>`, and for most other types it is `${type}Var`.

For types that have both representations, the one with a "lvalue" has a mutable `.value` property for accessing the value.

Pointer types

The type argument `T` of `CPointer<T>` must be one of the "lvalue" types described above, e.g., the C type struct `S*` is mapped to `CPointer<S>`, `int8_t*` is mapped to `CPointer<int_8tVar>`, and `char**` is mapped to `CPointer<CPointerVar<ByteVar>>`.

C null pointer is represented as Kotlin's `null`, and the pointer type `CPointer<T>` is not nullable, but the `CPointer<T>?` is. The values of this type support all the Kotlin operations related to handling null, e.g. `?:`, `?,`, `!!` etc.:

```
val path = getenv("PATH")?.toKString() ?: ""
```

Since the arrays are also mapped to `CPointer<T>`, it supports the `[]` operator for accessing values by index:

```
fun shift(ptr: CPointer<BytePtr>, length: Int) {
    for (index in 0 .. length - 2) {
        ptr[index] = ptr[index + 1]
    }
}
```

The `.pointed` property for `CPointer<T>` returns the lvalue of type `T`, pointed by this pointer. The reverse operation is `.ptr`: it takes the lvalue and returns the pointer to it.

`void*` is mapped to `COpaquePointer` – the special pointer type which is the supertype for any other pointer type. So if the C function takes `void*`, then the Kotlin binding accepts any `CPointer`.

Casting a pointer (including `COpaquePointer`) can be done with `.reinterpret<T>`, e.g.:

```
val intPtr = bytePtr.reinterpret<IntVar>()
```

or

```
val intPtr: CPointer<IntVar> = bytePtr.reinterpret()
```

As is with C, these reinterpret casts are unsafe and can potentially lead to subtle memory problems in the application.

Also there are unsafe casts between `CPointer<T>?` and `Long` available, provided by the `.toLong()` and `.toCPointer<T>()` extension methods:

```
val longValue = ptr.toLong()
val originalPtr = longValue.toCPointer<T>()
```

Note that if the type of the result is known from the context, the type argument can be omitted as usual due to the type inference.

Memory allocation

The native memory can be allocated using the NativePlacement interface, e.g.

```
val byteVar = placement.alloc<ByteVar>()
```

or

```
val bytePtr = placement.allocArray<ByteVar>(5)
```

The most "natural" placement is in the object nativeHeap. It corresponds to allocating native memory with malloc and provides an additional .free() operation to free allocated memory:

```
val buffer = nativeHeap.allocArray<ByteVar>(size)
<use buffer>
nativeHeap.free(buffer)
```

However, the lifetime of allocated memory is often bound to the lexical scope. It is possible to define such scope with memScoped { ... }. Inside the braces, the temporary placement is available as an implicit receiver, so it is possible to allocate native memory with alloc and allocArray, and the allocated memory will be automatically freed after leaving the scope.

For example, the C function returning values through pointer parameters can be used like

```
val fileSize = memScoped {
    val statBuf = alloc<stat>()
    val error = stat("/", statBuf.ptr)
    statBuf.st_size
}
```

Pass pointers to bindings

Although C pointers are mapped to the CPointer<T> type, the C function pointer-typed parameters are mapped to CValuesRef<T>. When passing CPointer<T> as the value of such a parameter, it is passed to the C function as is. However, the sequence of values can be passed instead of a pointer. In this case the sequence is passed "by value", i.e., the C function receives the pointer to the temporary copy of that sequence, which is valid only until the function returns.

The CValuesRef<T> representation of pointer parameters is designed to support C array literals without explicit native

memory allocation. To construct the immutable self-contained sequence of C values, the following methods are provided:

- \${type}Array.toCValues(), where type is the Kotlin primitive type
- Array<CPointer<T>?>.toCValues(), List<CPointer<T>?>.toCValues()
- cValuesOf(vararg elements: \${type}), where type is a primitive or pointer

For example:

C:

```
void foo(int* elements, int count);
...
int elements[] = {1, 2, 3};
foo(elements, 3);
```

Kotlin:

```
foo(cValuesOf(1, 2, 3), 3)
```

Strings

Unlike other pointers, the parameters of type const char* are represented as a Kotlin String. So it is possible to pass any Kotlin string to a binding expecting a C string.

There are also some tools available to convert between Kotlin and C strings manually:

- fun CPointer<ByteVar>.toKString(): String
- val String.cstr: CValuesRef<ByteVar>.

To get the pointer, .cstr should be allocated in native memory, e.g.

```
val cString = kotlinString.cstr.getPointer(nativeHeap)
```

In all cases, the C string is supposed to be encoded as UTF-8.

To skip automatic conversion and ensure raw pointers are used in the bindings, a noStringConversion statement in the .def file could be used, i.e.

```
noStringConversion = LoadCursorA LoadCursorW
```

This way any value of type CPointer<ByteVar> can be passed as an argument of const char* type. If a Kotlin string should be passed, code like this could be used:

```
memScoped {
    LoadCursorA(null, "cursor.bmp".cstr.ptr) // for ASCII version
```

```
        LoadCursorW(null, "cursor.bmp".wcstr.ptr) // for Unicode version
    }
```

Scope-local pointers

It is possible to create a scope-stable pointer of C representation of CValues<T> instance using the CValues<T>.ptr extension property, available under memScoped { ... }. It allows using the APIs which require C pointers with a lifetime bound to a certain MemScope. For example:

```
memScoped {
    items = arrayOfNulls<CPointer<ITEM>?>(6)
    arrayOf("one", "two").forEachIndexed { index, value -> items[index] = value.cstr.ptr }
    menu = new_menu("Menu".cstr.ptr, items.toCValues().ptr)
    ...
}
```

In this example, all values passed to the C API new_menu() have a lifetime of the innermost memScope it belongs to. Once the control flow leaves the memScoped scope the C pointers become invalid.

Pass and receive structs by value

When a C function takes or returns a struct / union T by value, the corresponding argument type or return type is represented as CValue<T>.

CValue<T> is an opaque type, so the structure fields cannot be accessed with the appropriate Kotlin properties. It should be possible, if an API uses structures as handles, but if field access is required, there are the following conversion methods available:

- fun T.readValue(): CValue<T>. Converts (the lvalue) T to a CValue<T>. So to construct the CValue<T>, T can be allocated, filled, and then converted to CValue<T>.
- CValue<T>.useContents(block: T.() -> R): R. Temporarily places the CValue<T> to memory, and then runs the passed lambda with this placed value T as receiver. So to read a single field, the following code can be used:

```
val fieldValue = structValue.useContents { field }
```

Callbacks

To convert a Kotlin function to a pointer to a C function, staticCFunction(::kotlinFunction) can be used. It is also able to provide the lambda instead of a function reference. The function or lambda must not capture any values.

If the callback doesn't run in the main thread, it is mandatory to init the Kotlin/Native runtime by calling kotlin.native.initRuntimeIfNeeded().

Pass user data to callbacks

Often C APIs allow passing some user data to callbacks. Such data is usually provided by the user when configuring the callback. It is passed to some C function (or written to the struct) as e.g. void*. However, references to Kotlin objects can't be directly passed to C. So they require wrapping before configuring the callback and then unwrapping in the callback itself, to safely swim from Kotlin to Kotlin through the C world. Such wrapping is possible with StableRef class.

To wrap the reference:

```
val stableRef = StableRef.create(kotlinReference)
val voidPtr = stableRef.asCPointer()
```

where the voidPtr is a COpaquePointer and can be passed to the C function.

To unwrap the reference:

```
val stableRef = voidPtr.asStableRef<KotlinClass>()
val kotlinReference = stableRef.get()
```

where kotlinReference is the original wrapped reference.

The created StableRef should eventually be manually disposed using the .dispose() method to prevent memory leaks:

```
stableRef.dispose()
```

After that it becomes invalid, so voidPtr can't be unwrapped anymore.

See the samples/libcurl for more details.

Macros

Every C macro that expands to a constant is represented as a Kotlin property. Other macros are not supported. However, they can be exposed manually by wrapping them with supported declarations. E.g. function-like macro FOO can be exposed as function foo by adding the custom declaration to the library:

```
headers = library/base.h

---

static inline int foo(int arg) {
    return FOO(arg);
}
```

Definition file hints

The .def file supports several options for adjusting the generated bindings.

- excludedFunctions property value specifies a space-separated list of the names of functions that should be ignored. This may be required because a function declared in the C header is not generally guaranteed to be really callable, and it is often hard or impossible to figure this out automatically. This option can also be used to workaround a bug in the interop itself.
- strictEnums and nonStrictEnums properties values are space-separated lists of the enums that should be generated as a Kotlin enum or as integral values correspondingly. If the enum is not included into any of these lists, then it is generated according to the heuristics.
- noStringConversion property value is space-separated lists of the functions whose const char* parameters shall not be autoconverted as Kotlin string

Portability

Sometimes the C libraries have function parameters or struct fields of a platform-dependent type, e.g. long or size_t. Kotlin itself doesn't provide neither implicit integer casts nor C-style integer casts (e.g. (size_t) intValue), so to make writing portable code in such cases easier, the convert method is provided:

```
fun ${type1}.convert<${type2}>(): ${type2}
```

where each of type1 and type2 must be an integral type, either signed or unsigned.

.convert<\${type}> has the same semantics as one of the .toByte, .toShort, .toInt, .toLong, .toUByte, .toUShort, .toUInt or .toULong methods, depending on type.

The example of using convert:

```
fun zeroMemory(buffer: C0paquePointer, size: Int) {
    memset(buffer, 0, size.convert<size_t>())
}
```

Also, the type parameter can be inferred automatically and so may be omitted in some cases.

Object pinning

Kotlin objects could be pinned, i.e. their position in memory is guaranteed to be stable until unpinned, and pointers to such objects inner data could be passed to the C functions. For example

```
fun readData(fd: Int): String {
    val buffer = ByteArray(1024)
    buffer.usePinned { pinned ->
        while (true) {
            val length = recv(fd, pinned.addressOf(0), buffer.size.convert(), 0).toInt()

            if (length <= 0) {
                break
            }
        }
        // Now `buffer` has raw data obtained from the `recv()` call.
    }
}
```

```
    }
}
}
```

Here we use service function `usePinned`, which pins an object, executes block and unpins it on normal and exception paths.

Mapping primitive data types from C – tutorial

In this tutorial, you will learn what C data types are visible in Kotlin/Native and vice versa. You will:

- See what [Data types are in C language](#).
- Create a [tiny C Library](#) that uses those types in exports.
- [Inspect generated Kotlin APIs from a C library](#).
- Find how [Primitive types in Kotlin](#) are mapped to C.

Types in C language

What types are there in the C language? Let's take the [C data types](#) article from Wikipedia as a basis. There are following types in the C programming language:

- basic types `char`, `int`, `float`, `double` with modifiers `signed`, `unsigned`, `short`, `long`
- structures, unions, arrays
- pointers
- function pointers

There are also more specific types:

- boolean type (from [C99](#))
- `size_t` and `ptrdiff_t` (also `ssize_t`)
- fixed width integer types, such as `int32_t` or `uint64_t` (from [C99](#))

There are also the following type qualifiers in the C language: `const`, `volatile`, `restrict`, `atomic`.

The best way to see what C data types are visible in Kotlin is to try it.

Example C library

Create a lib.h file to see how C functions are mapped into Kotlin:

```
#ifndef LIB2_H_INCLUDED
#define LIB2_H_INCLUDED

void ints(char c, short d, int e, long f);
void uints(unsigned char c, unsigned short d, unsigned int e, unsigned long f);
void doubles(float a, double b);

#endif
```

The file is missing the `extern "C"` block, which is not needed for this example, but may be necessary if you use C++ and overloaded functions. The [C++ compatibility thread](#) on Stackoverflow contains more details on this.

For every set of .h files, you will be using the [cinterop tool](#) from Kotlin/Native to generate a Kotlin/Native library, or .klib. The generated library will bridge calls from Kotlin/Native to C. It includes respective Kotlin declarations for the definitions from the .h files. It is only necessary to have a .h file to run the cinterop tool. And you do not need to create a lib.c file, unless you want to compile and run the example. More details on this are covered in the [C interop](#) page. It is enough for the tutorial to create the lib.def file with the following content:

```
headers = lib.h
```

You may include all declarations directly into the .def file after a --- separator. It can be helpful to include macros or other C defines into the code generated by the cinterop tool. Method bodies are compiled and fully included into the binary too. Use that feature to have a runnable example without a need for a C compiler. To implement that, you need to add implementations to the C functions from the lib.h file, and place these functions into a .def file. You will have the following interop.def result:

```
---

void ints(char c, short d, int e, long f) { }
void uints(unsigned char c, unsigned short d, unsigned int e, unsigned long f) { }
void doubles(float a, double b) { }
```

The interop.def file is enough to compile and run the application or open it in an IDE. Now it is time to create project files, open the project in [IntelliJ IDEA](#) and run it.

Inspect generated Kotlin APIs for a C library

While it is possible to use the command line, either directly or by combining it with a script file (such as .sh or .bat file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the [A Basic Kotlin/Native Application](#) tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and multiplatform builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following build.gradle(.kts) Gradle build file:

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.5.20'
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64('native') { // on Linux
        // macosX64('native') { // on macOS
        // mingwX64('native') { // on Windows
            compilations.main.cinterops {
                interop
            }

            binaries {
                executable()
            }
        }
    }
}

wrapper {
    gradleVersion = '6.7.1'
    distributionType = 'BIN'
}
```

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.5.20"
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64("native") { // on Linux
        // macosX64("native") { // on macOS
        // mingwX64("native") { // on Windows
```

```

    val main by compilations.getting
    val interop by main.cinterop.create()

    binaries {
        executable()
    }
}

tasks.wrapper {
    gradleVersion = "6.7.1"
    distributionType = Wrapper.DistributionType.BIN
}

```

The prepared project sources can be directly downloaded from Github:

- for macOS: [Groovy, Kotlin](#)
- for Linux: [Groovy, Kotlin](#)
- for Windows: [Groovy, Kotlin](#)

The project file configures the C interop as an additional step of the build. Let's move the interop.def file to the src/nativeInterop/cinterop directory. Gradle recommends using conventions instead of configurations, for example, the source files are expected to be in the src/nativeMain/kotlin folder. By default, all the symbols from C are imported to the interop package, you may want to import the whole package in our .kt files. Check out the [kotlin-multiplatform](#) plugin documentation to learn about all the different ways you could configure it.

Create a src/nativeMain/kotlin/hello.kt stub file with the following content to see how C primitive type declarations are visible from Kotlin:

```

import interop.*

fun main() {
    println("Hello Kotlin/Native!")

    ints(/* fix me*/)
    uints(/* fix me*/)
    doubles(/* fix me*/)
}

```

Now you are ready to [open the project in IntelliJ IDEA](#) and to see how to fix the example project. While doing that, see how C primitive types are mapped into Kotlin/Native.

Primitive types in kotlin

With the help of IntelliJ IDEA's Go to | Declaration or compiler errors, you see the following generated API for the C functions:

```
fun ints(c: Byte, d: Short, e: Int, f: Long)
fun uints(c: UByte, d: UShort, e: UInt, f: ULong)
fun doubles(a: Float, b: Double)
```

C types are mapped in the way we would expect, note that char type is mapped to kotlin.Byte as it is usually an 8-bit signed value.

C	Kotlin
char	kotlin.Byte
unsigned char	kotlin.UByte
short	kotlin.Short
unsigned short	kotlin.UShort
int	kotlin.Int
unsigned int	kotlin.UInt
long long	kotlin.Long
unsigned long long	kotlin.ULong
float	kotlin.Float
double	kotlin.Double

Fix the code

You've seen all definitions and it is the time to fix the code. Run the runDebugExecutableNative Gradle task [in IDE](#) or use the following command to run the code:

```
./gradlew runDebugExecutableNative
```

The final code in the hello.kt file may look like that:

```
import interop.*

fun main() {
    println("Hello Kotlin/Native!")
```

```
    ints(1, 2, 3, 4)
    uints(5, 6, 7, 8)
    doubles(9.0f, 10.0)
}
```

Next steps

Continue to explore more complicated C language types and their representation in Kotlin/Native in the next tutorials:

- [Mapping struct and union types from C](#)
- [Mapping function pointers from C](#)
- [Mapping strings from C](#)

The [C interop documentation](#) covers more advanced scenarios of the interop.

Mapping struct and union types from C – tutorial

This is the second post in the series. The very first tutorial of the series is [Mapping primitive data types from C](#). There are also the [Mapping function pointers from C](#) and [Mapping Strings from C](#) tutorials.

In the tutorial, you will learn:

- [How struct and union types are mapped](#)
- [How to use struct and union type from Kotlin](#)

Mapping struct and union C types

The best way to understand the mapping between Kotlin and C is to try a tiny example. We will declare a struct and a union in the C language, to see how they are mapped into Kotlin.

Kotlin/Native comes with the cinterop tool, the tool generates bindings between the C language and Kotlin. It uses a .def file to specify a C library to import. More details are discussed in the [Interop with C Libraries](#) tutorial.

In [the previous tutorial](#), you've created a lib.h file. This time, include those declarations directly into the interop.def file, after the --- separator line:

```
---
```

```

typedef struct {
    int a;
    double b;
} MyStruct;

void struct_by_value(MyStruct s) {}
void struct_by_pointer(MyStruct* s) {}

typedef union {
    int a;
    MyStruct b;
    float c;
} MyUnion;

void union_by_value(MyUnion u) {}
void union_by_pointer(MyUnion* u) {}

```

The interop.def file is enough to compile and run the application or open it in an IDE. Now it is time to create project files, open the project in [IntelliJ IDEA](#) and run it.

Inspect Generated Kotlin APIs for a C library

While it is possible to use the command line, either directly or by combining it with a script file (such as .sh or .bat file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the [A Basic Kotlin/Native Application](#) tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and [multiplatform](#) builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following build.gradle(kts) Gradle build file:

Groovy

```

plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.5.20'
}

repositories {
    mavenCentral()
}

kotlin {

```

```

linuxX64('native') { // on Linux
// macosX64('native') { // on macOS
// mingwX64('native') { //on Windows
    compilations.main.cinterops {
        interop
    }

    binaries {
        executable()
    }
}
}

wrapper {
    gradleVersion = '6.7.1'
    distributionType = 'BIN'
}

```

Kotlin

```

plugins {
    kotlin("multiplatform") version "1.5.20"
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64("native") { // on Linux
        // macosX64("native") { // on macOS
        // mingwX64("native") { // on Windows
            val main by compilations.getting
            val interop by main.cinterops.creating

            binaries {
                executable()
            }
        }
    }

    tasks.wrapper {
        gradleVersion = "6.7.1"
        distributionType = Wrapper.DistributionType.BIN
    }
}

```

The prepared project sources can be directly downloaded from GitHub:

- for macOS: [Groovy](#), [Kotlin](#)
- for Linux: [Groovy](#), [Kotlin](#)
- for Windows: [Groovy](#), [Kotlin](#)

The project file configures the C interop as an additional step of the build. Let's move the interop.def file to the

src/nativeInterop/cinterop directory. Gradle recommends using conventions instead of configurations, for example, the source files are expected to be in the src/nativeMain/kotlin folder. By default, all the symbols from C are imported to the interop package, you may want to import the whole package in our .kt files. Check out the [kotlin-multiplatform](#) plugin documentation to learn about all the different ways you could configure it.

Create a src/nativeMain/kotlin/hello.kt stub file with the following content to see how C declarations are visible from Kotlin:

```
import interop.*

fun main() {
    println("Hello Kotlin/Native!")

    struct_by_value(/* fix me*/)
    struct_by_pointer(/* fix me*/)
    union_by_value(/* fix me*/)
    union_by_pointer(/* fix me*/)
}
```

Now you are ready to [open the project in IntelliJ IDEA](#) and to see how to fix the example project. While doing that, see how C primitive types are mapped into Kotlin/Native.

Primitive types in Kotlin

With the help of IntelliJ IDEA's Go to | Declaration or compiler errors, you see the following generated API for the C functions, struct, and union:

```
fun struct_by_value(s: CValue<MyStruct>)
fun struct_by_pointer(s: CValuesRef<MyStruct>?)

fun union_by_value(u: CValue<MyUnion>)
fun union_by_pointer(u: CValuesRef<MyUnion>?)

class MyStruct constructor(rawPtr: NativePtr /* = NativePtr */) : CStructVar {
    var a: Int
    var b: Double
    companion object : CStructVar.Type
}

class MyUnion constructor(rawPtr: NativePtr /* = NativePtr */) : CStructVar {
    var a: Int
    val b: MyStruct
    var c: Float
    companion object : CStructVar.Type
}
```

You see that cinterop generated wrapper types for our struct and union types. For MyStruct and MyUnion type declarations in C, there are the Kotlin classes MyStruct and MyUnion generated respectively. The wrappers inherit from the CStructVar base class and declare all fields as Kotlin properties. It uses CValue<T> to represent a by-value structure parameter and CValuesRef<T>? to represent passing a pointer to a structure or a union.

Technically, there is no difference between struct and union types on the Kotlin side. Note that a, b, and c properties of MyUnion class in Kotlin use the same memory location to read/write their value just like union does in C language.

More details and advanced use-cases are presented in the [C Interop documentation](#)

Use struct and union types from Kotlin

It is easy to use the generated wrapper classes for C struct and union types from Kotlin. Thanks to the generated properties, it feels natural to use them in Kotlin code. The only question, so far, is how to create a new instance on those classes. As you see from the declarations of MyStruct and MyUnion, their constructors require a NativePtr. Of course, you are not willing to deal with pointers manually. Instead, you can use Kotlin API to have those objects instantiated for us.

Let's take a look at the generated functions that take our MyStruct and MyUnion as parameters. You see that by-value parameters are represented as kotlinx.cinterop.CValue<T>. And for typed pointer parameters you see kotlinx.cinterop.CValuesRef<T>. Kotlin provides us with an API to deal with both types easily, let's try it and see.

Create a CValue

CValue<T> type is used to pass by-value parameters to a C function call. Use cValue function to create CValue<T> object instance. The function requires a [lambda function with a receiver](#) to initialize the underlying C type in-place. The function is declared as follows:

```
fun <reified T : CStructVar> cValue(initialize: T.() -> Unit): CValue<T>
```

Now it is time to see how to use cValue and pass by-value parameters:

```
fun callValue() {
    val cStruct = cValue<MyStruct> {
        a = 42
        b = 3.14
    }
    struct_by_value(cStruct)

    val cUnion = cValue<MyUnion> {
        b.a = 5
        b.b = 2.7182
    }

    union_by_value(cUnion)
}
```

Create struct and union as CValuesRef

CValuesRef<T> type is used in Kotlin to pass a typed pointer parameter of a C function. First, you need an instance of

MyStruct and MyUnion classes. Create them directly in the native memory. Use the

```
fun <reified T : kotlinx.cinterop.CVariable> alloc(): T
```

extension function on kotlinx.cinterop.NativePlacement type for this.

NativePlacement represents native memory with functions similar to malloc and free. There are several implementations of NativePlacement. The global one is called with kotlinx.cinterop.nativeHeap and don't forget to call the nativeHeap.free(..) function to free the memory after use.

Another option is to use the

```
fun <R> memScoped(block: kotlinx.cinterop.MemScope.() -> R): R
```

function. It creates a short-lived memory allocation scope, and all allocations will be cleaned up automatically at the end of the block.

Your code to call functions with pointers will look like this:

```
fun callRef() {
    memScoped {
        val cStruct = alloc<MyStruct>()
        cStruct.a = 42
        cStruct.b = 3.14

        struct_by_pointer(cStruct.ptr)

        val cUnion = alloc<MyUnion>()
        cUnion.b.a = 5
        cUnion.b.b = 2.7182

        union_by_pointer(cUnion.ptr)
    }
}
```

Note that this code uses the extension property ptr which comes from a memScoped lambda receiver type, to turn MyStruct and MyUnion instances into native pointers.

The MyStruct and MyUnion classes have the pointer to the native memory underneath. The memory will be released when a memScoped function ends, which is equal to the end of its block. Make sure that a pointer is not used outside of the memScoped call. You may use Arena() or nativeHeap for pointers that should be available longer, or are cached inside a C library.

Conversion between CValue and CValuesRef

Of course, there are use cases when you need to pass a struct as a value to one call, and then, to pass the same struct as a reference to another call. This is possible in Kotlin/Native too. A NativePlacement will be needed here.

Let's see now CValue<T> is turned to a pointer first:

```
fun callMix_ref() {
    val cStruct = cValue<MyStruct> {
        a = 42
        b = 3.14
    }

    memScoped {
        struct_by_pointer(cStruct.ptr)
    }
}
```

This code uses the extension property ptr which comes from memScoped lambda receiver type to turn MyStruct and MyUnion instances into native pointers. Those pointers are only valid inside the memScoped block.

For the opposite conversion, to turn a pointer into a by-value variable, we call the readValue() extension function:

```
fun callMix_value() {
    memScoped {
        val cStruct = alloc<MyStruct>()
        cStruct.a = 42
        cStruct.b = 3.14

        struct_by_value(cStruct.readValue())
    }
}
```

Run the code

Now when you have learned how to use C declarations in your code, you are ready to try it out on a real example. Let's fix the code and see how it runs by calling the runDebugExecutableNative Gradle task [in the IDE](#) or by using the following console command:

```
./gradlew runDebugExecutableNative
```

The final code in the hello.kt file may look like this:

```
import interop./*
import kotlinc.cinterop.alloc
import kotlinc.cinterop.cValue
import kotlinc.cinterop.memScoped
import kotlinc.cinterop.ptr
import kotlinc.cinterop.readValue

fun main() {
    println("Hello Kotlin/Native!")

    val cUnion = cValue<MyUnion> {
```

```

        b.a = 5
        b.b = 2.7182
    }

    memScoped {
        union_by_value(cUnion)
        union_by_pointer(cUnion.ptr)
    }

    memScoped {
        val cStruct = alloc<MyStruct> {
            a = 42
            b = 3.14
        }

        struct_by_value(cStruct.readValue())
        struct_by_pointer(cStruct.ptr)
    }
}

```

Next steps

Continue exploring the C language types and their representation in Kotlin/Native in the related tutorials:

- [Mapping primitive data types from C](#)
- [Mapping function pointers from C](#)
- [Mapping strings from C](#)

The [C Interop documentation](#) covers more advanced scenarios of the interop.

Mapping function pointers from C – tutorial

This is the third post in the series. The very first tutorial is [Mapping primitive data types from C](#). There are also [Mapping struct](#) and [union types from C](#) and [Mapping strings from C](#) tutorials.

In this tutorial We will learn how to:

- [Pass Kotlin function as C function pointer](#)
- [Use C function pointer from Kotlin](#)

Mapping function pointer types from C

The best way to understand the mapping between Kotlin and C is to try a tiny example. Declare a function that accepts a function pointer as a parameter and another function that returns a function pointer.

Kotlin/Native comes with the cinterop tool; the tool generates bindings between the C language and Kotlin. It uses a .def file to specify a C library to import. More details on this are in [Interop with C Libraries](#).

The quickest way to try out C API mapping is to have all C declarations in the interop.def file, without creating any .h or .c files at all. Then place the C declarations in a .def file after the special --- separator line:

```
---

int myFun(int i) {
    return i+1;
}

typedef int (*MyFun)(int);

void accept_fun(MyFun f) {
    f(42);
}

MyFun supply_fun() {
    return myFun;
}
```

The interop.def file is enough to compile and run the application or open it in an IDE. Now it is time to create project files, open the project in [IntelliJ IDEA](#) and run it.

Inspect generated Kotlin APIs for a C library

While it is possible to use the command line, either directly or by combining it with a script file (such as .sh or .bat file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the [A Basic Kotlin/Native Application](#) tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and [multiplatform](#) builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following build.gradle.kts Gradle build file:

Groovy

```
plugins {
```

```

    id 'org.jetbrains.kotlin.multiplatform' version '1.5.20'
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64('native') { // on Linux
    // macosX64('native') { // on macOS
    // mingwX64('native') { //on Windows
        compilations.main.cinterops {
            interop
        }

        binaries {
            executable()
        }
    }
}

wrapper {
    gradleVersion = '6.7.1'
    distributionType = 'BIN'
}

```

Kotlin

```

plugins {
    kotlin("multiplatform") version "1.5.20"
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64("native") { // on Linux
    // macosX64("native") { // on macOS
    // mingwX64("native") { // on Windows
        val main by compilations.getting
        val interop by main.cinterops.creating

        binaries {
            executable()
        }
    }
}

tasks.wrapper {
    gradleVersion = "6.7.1"
    distributionType = Wrapper.DistributionType.BIN
}

```

The prepared project sources can be directly downloaded from Github:

- for macOS: [Groovy](#), [Kotlin](#)
- for Linux: [Groovy](#), [Kotlin](#)
- for Windows: [Groovy](#), [Kotlin](#)

The project file configures the C interop as an additional step of the build. Let's move the interop.def file to the src/nativeInterop/cinterop directory. Gradle recommends using conventions instead of configurations, for example, the source files are expected to be in the src/nativeMain/kotlin folder. By default, all the symbols from C are imported to the interop package, you may want to import the whole package in our .kt files. Check out the [kotlin-multiplatform](#) plugin documentation to learn about all the different ways you could configure it.

Let's create a src/nativeMain/kotlin/hello.kt stub file with the following content to see how C primitive type declarations are visible from Kotlin:

```
import interop.*

fun main() {
    println("Hello Kotlin/Native!")

    accept_fun(https://kotlinlang.org/\*fix me \*/)
    val useMe = supply_fun()
}
```

Now you are ready to [open the project in IntelliJ IDEA](#) and to see how to fix the example project. While doing that, see how C functions are mapped into Kotlin/Native declarations.

C function pointers in Kotlin

With the help of IntelliJ IDEA's Go to | Declaration or compiler errors, see the following declarations for the C functions:

```
fun accept_fun(f: MyFun? /* = CPointer<CFunction<(Int) -> Int>>? */)
fun supply_fun(): MyFun? /* = CPointer<CFunction<(Int) -> Int>>? */

fun myFun(i: kotlin.Int): kotlin.Int

typealias MyFun = kotlndx.cinterop.CPointer<kotlndx.cinterop.CFunction<(kotlin.Int) -> kotlin.Int>>

typealias MyFunVar = kotlndx.cinterop.CPointerVarOf<lib.MyFun>
```

You see that the function's typedef from C has been turned into Kotlin typealias. It uses CPointer<..> type to represent the pointer parameters, and CFunction<(Int)->Int> to represent the function signature. There is an invoke operator extension function available for all CPointer<CFunction<..>> types, so that it is possible to call it as you would call any other function in Kotlin.

Pass Kotlin function as C function pointer

It is the time to try using C functions from the Kotlin program. Call the accept_fun function and pass the C function pointer to a Kotlin lambda:

```
fun myFun() {
    accept_fun(staticCFunction<Int, Int> { it + 1 })
}
```

This call uses the staticCFunction{..} helper function from Kotlin/Native to wrap a Kotlin lambda function into a C function pointer. It only allows having unbound and non-capturing lambda functions. For example, it is not able to use a local variable from the function. You may only use globally visible declarations. Throwing exceptions from a staticCFunction{..} will end up in non-deterministic side-effects. It is vital to make sure that your code is not throwing any sudden exceptions from it.

Use the C function pointer from Kotlin

The next step is to call a C function pointer from a C pointer that you have from the supply_fun() call:

```
fun myFun2() {
    val functionFromC = supply_fun() ?: error("No function is returned")

    functionFromC(42)
}
```

Kotlin turns the function pointer return type into a nullable CPointer<CFunction<..>> object. There is the need to explicitly check for null first. The [elvis operator](#) for that in the code above. The cinterop tool helps us to turn a C function pointer into an easy to call object in Kotlin. This is what we did on the last line.

Fix the code

You've seen all definitions and it is time to fix and run the code. Run the runDebugExecutableNative Gradle task [in the IDE](#) or use the following command to run the code:

```
./gradlew runDebugExecutableNative
```

The code in the hello.kt file may look like this:

```
import interop.*
import kotlincinterop.*

fun main() {
```

```

    println("Hello Kotlin/Native!")

    val cFunctionPointer = staticCFunction<Int, Int> { it + 1 }
    accept_fun(cFunctionPointer)

    val funFromC = supply_fun() ?: error("No function is returned")
    funFromC(42)
}

```

Next Steps

Continue exploring more C language types and their representation in Kotlin/Native in next tutorials:

- [Mapping primitive data types from C](#)
- [Mapping struct and union types from C](#)
- [Mapping strings from C](#)

The [C Interop documentation](#) covers more advanced scenarios of the interop.

Mapping Strings from C – tutorial

This is the last tutorial in the series. The first tutorial of the series is [Mapping primitive data types from C](#). There are also [Mapping struct and union types from C](#) and [Mapping function pointers from C](#) tutorials.

In this tutorial, you'll see how to deal with C strings in Kotlin/Native. You will learn how to:

- [Pass a Kotlin string to C](#)
- [Read a C string in Kotlin](#)
- [Receive C string bytes into a Kotlin string](#)

Working with C strings

There is no dedicated type in C language for strings. A developer knows from a method signature or the documentation, whether a given `char *` means a C string in the context. Strings in the C language are null-terminated, a trailing zero character `\0` is added at the end of a bytes sequence to mark a string termination. Usually, [UTF-8 encoded strings](#) are used. The UTF-8 encoding uses variable width characters, and it is backward compatible with [ASCII](#). Kotlin/Native uses UTF-8 character encoding by default.

The best way to understand the mapping between C and Kotlin languages is to try it out on a small example. Create a small library headers for that. First, create a lib.h file with the following declaration of functions that deal with the C strings:

```

#ifndef LIB2_H_INCLUDED
#define LIB2_H_INCLUDED

void pass_string(char* str);
char* return_string();
int copy_string(char* str, int size);

#endif

```

In the example, you see the most popular ways to pass or receive a string in the C language. Take the return of `return_string` with care. In general, it is best to make sure you use the right function to dispose the returned `char*` with the right `free(..)` function call.

Kotlin/Native comes with the cinterop tool; the tool generates bindings between the C language and Kotlin. It uses a `.def` file to specify a C library to import. More details on this are in the [Interop with C Libraries](#) tutorial. The quickest way to try out C API mapping is to have all C declarations in the `interop.def` file, without creating any `.h` or `.c` files at all. Then place the C declarations in a `interop.def` file after the special `---` separator line:

```

headers = lib.h
---

void pass_string(char* str) {

char* return_string() {
    return "C stirng";
}

int copy_string(char* str, int size) {
    *str++ = 'C';
    *str++ = ' ';
    *str++ = 'K';
    *str++ = '/';
    *str++ = 'N';
    *str++ = 0;
    return 0;
}

```

The `interop.def` file is enough to compile and run the application or open it in an IDE. Now it is time to create project files, open the project in [IntelliJ IDEA](#) and run it.

Inspect generated Kotlin APIs for a C library

While it is possible to use the command line, either directly or by combining it with a script file (such as `.sh` or `.bat` file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the [A Basic Kotlin/Native Application](#) tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and multiplatform builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following build.gradle(.kts) Gradle build file:

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.5.20'
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64('native') { // on Linux
        // macosX64('native') { // on macOS
        // mingwX64('native') { //on Windows
            compilations.main.cinterops {
                interop
            }

            binaries {
                executable()
            }
        }
    }
}

wrapper {
    gradleVersion = '6.7.1'
    distributionType = 'BIN'
}
```

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.5.20"
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64("native") { // on Linux
        // macosX64("native") { // on macOS
        // mingwX64("native") { // on Windows
```

```

    val main by compilations.getting
    val interop by main.cinterop.create()

    binaries {
        executable()
    }
}

tasks.wrapper {
    gradleVersion = "6.7.1"
    distributionType = Wrapper.DistributionType.BIN
}

```

The prepared project sources can be directly downloaded from GitHub:

- for macOS: [Groovy, Kotlin](#)
- for Linux: [Groovy, Kotlin](#)
- for Windows: [Groovy, Kotlin](#)

The project file configures the C interop as an additional step of the build. Let's move the interop.def file to the src/nativeInterop/cinterop directory. Gradle recommends using conventions instead of configurations, for example, the source files are expected to be in the src/nativeMain/kotlin folder. By default, all the symbols from C are imported to the interop package, you may want to import the whole package in our .kt files. Check out the [kotlin-multiplatform](#) plugin documentation to learn about all the different ways you could configure it.

Let's create a src/nativeMain/kotlin/hello.kt stub file with the following content to see how C primitive type declarations are visible from Kotlin:

```

import interop.*

fun main() {
    println("Hello Kotlin/Native!")

    pass_string(/*fix me*/)
    val useMe = return_string()
    val useMe2 = copy_string(/*fix me*/)
}

```

Now you are ready to [open the project in IntelliJ IDEA](#) and to see how to fix the example project. While doing that, see how C primitive types are mapped into Kotlin/Native.

Primitive types in Kotlin

With the help of IntelliJ IDEA's Go to | Declaration or compiler errors, you see the following generated API for the C functions:

```
fun pass_string(str: CValuesRef<ByteVar> /* = ByteVarOf<Byte> */?)  
fun return_string(): CPointer<ByteVar> /* = ByteVarOf<Byte> */?  
fun copy_string(str: CValuesRef<ByteVar> /* = ByteVarOf<Byte> */?, size: Int): Int
```

These declarations look clear. All char * pointers are turned into str: CValuesRef<ByteVar>? for parameters and to CPointer<ByteVar>? in return types. Kotlin turns char type into kotlin.Byte type, as it is usually an 8-bit signed value.

In the generated Kotlin declarations, you see that str is represented as CValuesRef<ByteVar>?. The type is nullable, and you can simply pass Kotlin null as the parameter value.

Pass Kotlin string to C

Let's try to use the API from Kotlin. Call pass_string first:

```
fun passStringToC() {  
    val str = "this is a Kotlin String"  
    pass_string(str.cstr)  
}
```

Passing a Kotlin string to C is easy, thanks to the fact that there is String.cstr [extension property](#) in Kotlin for it. There is also String.wcstr for cases when you need UTF-16 wide characters.

Read C Strings in Kotlin

This time you'll take a returned char * from the return_string function and turn it into a Kotlin string. For that, do the following in Kotlin:

```
fun passStringToC() {  
    val stringFromC = return_string()?.toKString()  
  
    println("Returned from C: $stringFromC")  
}
```

This code uses the toKString() extension function above. Please do not miss out the toString() function. The toKString() has two overloaded extension functions in Kotlin:

```
fun CPointer<ByteVar>.toKString(): String  
fun CPointer<ShortVar>.toKString(): String
```

The first extension takes a char * as a UTF-8 string and turns it into a String. The second function does the same but for wide UTF-16 strings.

Receive C string bytes from Kotlin

This time we will ask a C function to write us a C string to a given buffer. The function is called `copy_string`. It takes a pointer to the location writing characters and the allowed buffer size. The function returns something to indicate if it has succeeded or failed. Let's assume 0 means it succeeded, and the supplied buffer was big enough:

```
fun sendString() {
    val buf = ByteArray(255)
    buf.usePinned { pinned ->
        if (copy_string(pinned.addressOf(0), buf.size - 1) != 0) {
            throw Error("Failed to read string from C")
        }
    }

    val copiedStringFromC = buf.stringFromUtf8()
    println("Message from C: $copiedStringFromC")
}
```

First of all, you need to have a native pointer to pass to the C function. Use the `usePinned` extension function to temporarily pin the native memory address of the byte array. The C function fills in the byte array with data. Use another extension function `ByteArray.stringFromUtf8()` to turn the byte array into a Kotlin String, assuming UTF-8 encoding.

Fix the Code

You've now seen all the definitions and it is time to fix the code. Run the `runDebugExecutableNative` Gradle task [in the IDE](#) or use the following command to run the code:

```
./gradlew runDebugExecutableNative
```

The code in the final `hello.kt` file may look like this:

```
import interop.*
import kotlinx.cinterop.*

fun main() {
    println("Hello Kotlin/Native!")

    val str = "this is a Kotlin String"
    pass_string(str.cstr)

    val useMe = return_string()?.toString() ?: error("null pointer returned")
    println(useMe)

    val copyFromC = ByteArray(255).usePinned { pinned ->

        val useMe2 = copy_string(pinned.addressOf(0), pinned.get().size - 1)
        if (useMe2 != 0) throw Error("Failed to read string from C")
    }
}
```

```
    pinned.get().stringFromUtf8()  
}  
  
    println(copyFromC)  
}
```

Next steps

Continue to explore more C language types and their representation in Kotlin/Native in our other tutorials:

- [Mapping primitive data types from C](#)
- [Mapping struct and union types from C](#)
- [Mapping function pointers from C](#)

The [C Interop documentation](#) documentation covers more advanced scenarios of the interop.

Using C Interop and libcurl for an app – tutorial

When writing native applications, oftentimes you need to access certain functionality that is not included in the Kotlin standard library, such as making HTTP requests, reading and writing from disk, etc.

Kotlin/Native provides you with the ability to consume standard C libraries, opening up an entire ecosystem of functionality that exists for pretty much anything you could need. In fact, Kotlin/Native already ships with a set of prebuilt [platform libraries](#) which provide some additional common functionality to that of the standard library.

In this tutorial, you'll see how to use some specific libraries, such as libcurl. You'll learn to:

- [Create Kotlin bindings](#)
- [Consume a generated Kotlin API](#)
- [Link the library into the application](#)

Generate bindings

An ideal scenario for interop is to call C functions as if you were calling Kotlin functions, that is, following the same signature and conventions. This is precisely what the cinterop tool provides. It takes a C library and generates the corresponding Kotlin bindings for it, which then allows you to use the library as if it were Kotlin code.

In order to generate these bindings, you need to create a library definition .def file that contains some information about the headers you need to generate. In this case, you'll use the famous libcurl library to make some HTTP calls, so create a file named libcurl.def with the following contents:

```
headers = curl/curl.h
headerFilter = curl/*

compilerOpts.linux = -I/usr/include -I/usr/include/x86_64-linux-gnu
linkerOpts.osx = -L/opt/local/lib -L/usr/local/opt/curl/lib -lcurl
linkerOpts.linux = -L/usr/lib/x86_64-linux-gnu -lcurl
```

A few things are going on in this file, let's go through them one by one. The first entry is headers which is the list of header files that you want to generate Kotlin stubs for. You can add multiple files to this entry, separating each one with a \ on a new line. For this tutorial, you'll only need curl.h. The files we are referencing need to be relative to the folder where the definition file is, or be available on the system path (/usr/include/curl).

The second line is the headerFilter. This is used to denote what exactly we want included. In C, when one file references another file with the #include directive, all the headers are also included. Sometimes this may not be needed, and you can use this parameter, [using glob patterns](#), to fine tune things. Note, that headerFilter is an optional argument and mostly only used when the library you're using is being installed as a system library, and you do not want to fetch external dependencies (such as system stdint.h header) into your interop library. It may be important for both optimizing the library size and fixing potential conflicts between the system and the Kotlin/Native provided compilation environment.

The next lines are about providing linker and compiler options, which can vary depending on different target platforms. In this tutorial, we are defining it for macOS (the .osx suffix) and Linux (the .linux suffix). Parameters without a suffix is also possible (e.g. linkerOpts=) and will be applied to all platforms.

The convention that is followed is that each library gets its own definition file, usually named the same as the library. For more information on all the options available to cinterop, see [the Interop documentation](#)

Once you have the definition file ready, create project files and open the project in an IDE.

While it is possible to use the command line, either directly or by combining it with a script file (such as .sh or .bat file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the [A Basic Kotlin/Native Application](#) tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and [multiplatform](#) builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following build.gradle.kts Gradle build file:

Groovy

```
plugins {
```

```

    id 'org.jetbrains.kotlin.multiplatform' version '1.5.20'
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64("native") { // on Linux
    // macosX64("native") { // on macOS
    // mingwX64("native") { //on Windows
        compilations.main.cinterops {
            interop
        }

        binaries {
            executable()
        }
    }
}

wrapper {
    gradleVersion = "6.7.1"
    distributionType = "ALL"
}

```

Kotlin

```

plugins {
    kotlin("multiplatform") version "1.5.20"
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64("native") { // on Linux
    // macosX64("native") { // on macOS
    // mingwX64("native") { //on Windows
        val main by compilations.getting
        val interop by main.cinterops.creating

        binaries {
            executable()
        }
    }
}

tasks.wrapper {
    gradleVersion = "6.7.1"
    distributionType = Wrapper.DistributionType.ALL
}

```

The prepared project sources can be directly downloaded from Github:

- for macOS: [Groovy](#), [Kotlin](#)
- for Linux: [Groovy](#), [Kotlin](#)
- for Windows: [Groovy](#), [Kotlin](#)

The project file configures the C interop as an additional step of the build. Let's move the interop.def file to the src/nativeInterop/cinterop directory. Gradle recommends using conventions instead of configurations, for example, the source files are expected to be in the src/nativeMain/kotlin folder. By default, all the symbols from C are imported to the interop package, you may want to import the whole package in our .kt files. Check out the [kotlin-multiplatform](#) plugin documentation to learn about all the different ways you could configure it.

curl on Windows

You should have the curl library binaries on Windows to make the sample work. You may build curl from [sources](#) on Windows (you'll need Visual Studio or Windows SDK Commandline tools), for more details, see the [related blog post](#). Alternatively, you may also want to consider a [MinGW/MSYS2](#) curl binary.

Consume the Kotlin API

Now you have the library and Kotlin stubs and can consume them from our application. To keep things simple, in this tutorial you're going to convert one of the simplest libcurl examples over to Kotlin.

The code in question is from the [simple](#) example (comments removed for brevity):

```
#include <stdio.h>
#include <curl/curl.h>

int main(void)
{
    CURL *curl;
    CURLcode res;

    curl = curl_easy_init();
    if(curl) {
        curl_easy_setopt(curl, CURLOPT_URL, "https://example.com");
        curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);

        res = curl_easy_perform(curl);
        if(res != CURLE_OK)
            fprintf(stderr, "curl_easy_perform() failed: %s\n",
                    curl_easy_strerror(res));
        curl_easy_cleanup(curl);
    }
    return 0;
}
```

The first thing you'll need is a Kotlin file called src/nativeMain/kotlin/hello.kt with the main function defined in it and then proceed to translate each line.

```

import interop.*
import kotlinc.cinterop.*

fun main(args: Array<String>) {
    val curl = curl_easy_init()
    if (curl != null) {
        curl_easy_setopt(curl, CURLOPT_URL, "https://example.com")
        curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L)
        val res = curl_easy_perform(curl)
        if (res != CURLE_OK) {
            println("curl_easy_perform() failed ${curl_easy_strerror(res)?.toString()}")
        }
        curl_easy_cleanup(curl)
    }
}

```

As you can see, you've eliminated the explicit variable declarations in the Kotlin version, but everything else is pretty much verbatim to the C version. All the calls you'd expect in the libcurl library are available in their Kotlin equivalent.

Note that for the purpose of this tutorial, we've done a line by line literal translation. Obviously you could write this in a more Kotlin idiomatic way.

Compile and link the library

The next step is to compile the application. We already covered the basics of compiling a Kotlin/Native application from the command line in the [A Basic Kotlin/Native application](#) tutorial. The only difference in this case is that the cinterop generated part is implicitly included into the build: Call the following command:

```
./gradlew runDebugExecutableNative
```

If there are no errors during compilation, you should see the result of the execution of the program, which on execution should output the contents of the site https://example.com

```
hadihariri@Falcon:~/work/kotlin-native/httpclient » ./program.kexe
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
body {
    background-color: #f0f0f2;
    margin: 0;
    padding: 0;
    font-family: "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;
}

div {
    width: 600px;
    margin: 5em auto;
    padding: 50px;
    background-color: #fff;
    border-radius: 1em;
}
a:link, a:visited {
    color: #38488f;
    text-decoration: none;
}
@media (max-width: 700px) {
```

Output

The reason you're seeing the actual output is because the call `curl_easy_perform` prints the result to the standard output. You could hide this using `curl_easy_setopt`.

For a more complete example of using libcurl, the [libcurl sample of the Kotlin/Native project](#) shows how to abstract the code into Kotlin classes as well as display headers. It also demonstrates how to make the steps a little easier by combining them into a shell script or Gradle build.

Interoperability with Swift/Objective-C

This document covers some details of Kotlin/Native interoperability with Swift/Objective-C.

Usage

Kotlin/Native provides bidirectional interoperability with Objective-C. Objective-C frameworks and libraries can be used in Kotlin code if properly imported to the build (system frameworks are imported by default). See [compilation configurations](#) for more details. A Swift library can be used in Kotlin code if its API is exported to Objective-C with `@objc`. Pure Swift modules are not yet supported.

Kotlin modules can be used in Swift/Objective-C code if compiled into a framework ([see here for how to declare binaries](#)). See [calculator sample](#) for an example.

Mappings

The table below shows how Kotlin concepts are mapped to Swift/Objective-C and vice versa.

"->" and "<- " indicate that mapping only goes one way.

Kotlin	Swift	Objective-C	Notes
class	class	@interface	note
interface	protocol	@protocol	
constructor/ create	Initializer	Initializer	note
Property	Property	Property	note note
Method	Method	Method	note note
suspend ->	completionHandler:		note
@Throws fun	throws	error:(NSError**)error	note
Extension	Extension	Category member	note
companion member <-	Class method or property	Class method or property	
null	nil	nil	
Singleton	Singleton()	[Singleton singleton]	note
Primitive type	Primitive type / NSNumber		note
Unit return type	Void	void	
String	String	NSString	
String	NSMutableString	NSMutableString	note
List	Array	NSArray	
MutableList	NSMutableArray	NSMutableArray	
Set	Set	NSSet	

Kotlin	Swift	Objective-C	Notes
MutableSet	NSMutableSet	NSMutableSet	note
Map	Dictionary	NSDictionary	
MutableMap	NSMutableDictionary	NSMutableDictionary	note
Function type	Function type	Block pointer type	note
Inline classes	Unsupported	Unsupported	note

Name translation

Objective-C classes are imported into Kotlin with their original names. Protocols are imported as interfaces with Protocol name suffix, i.e. @protocol Foo -> interface FooProtocol. These classes and interfaces are placed into a package [specified in build configuration](#) (platform.* packages for preconfigured system frameworks).

The names of Kotlin classes and interfaces are prefixed when imported to Objective-C. The prefix is derived from the framework name.

Initializers

Swift/Objective-C initializers are imported to Kotlin as constructors and factory methods named create. The latter happens with initializers declared in the Objective-C category or as a Swift extension, because Kotlin has no concept of extension constructors.

Kotlin constructors are imported as initializers to Swift/Objective-C.

Setters

Writeable Objective-C properties overriding read-only properties of the superclass are represented as setFoo() method for the property foo. Same goes for a protocol's read-only properties that are implemented as mutable.

Top-level functions and properties

Top-level Kotlin functions and properties are accessible as members of special classes. Each Kotlin file is translated into such a class. E.g.

```
// MyLibraryUtils.kt
package my.library

fun foo() {}
```

can be called from Swift like

```
MyLibraryUtilsKt.foo()
```

Method names translation

Generally Swift argument labels and Objective-C selector pieces are mapped to Kotlin parameter names. Anyway these two concepts have different semantics, so sometimes Swift/Objective-C methods can be imported with a clashing Kotlin signature. In this case the clashing methods can be called from Kotlin using named arguments, e.g.:

```
[player moveTo:LEFT byMeters:17]
[player moveTo:UP byInches:42]
```

in Kotlin it would be:

```
player.moveTo(LEFT, byMeters = 17)
player.moveTo(UP, byInches = 42)
```

Errors and exceptions

Kotlin has no concept of checked exceptions, all Kotlin exceptions are unchecked. Swift has only checked errors. So if Swift or Objective-C code calls a Kotlin method which throws an exception to be handled, then the Kotlin method should be marked with a @Throws annotation specifying a list of "expected" exception classes.

When compiling to Objective-C/Swift framework, non-suspend functions having or inheriting @Throws annotation are represented as NSError* -producing methods in Objective-C and as throws methods in Swift. Representations for suspend functions always have NSError*/ Error parameter in completion handler.

When Kotlin function called from Swift/Objective-C code throws an exception which is an instance of one of the @Throws -specified classes or their subclasses, it is propagated as NSError. Other Kotlin exceptions reaching Swift/Objective-C are considered unhandled and cause program termination.

suspend functions without @Throws propagate only CancellationException as NSError. Non-suspend functions without @Throws don't propagate Kotlin exceptions at all.

Note that the opposite reversed translation is not implemented yet: Swift/Objective-C error-throwing methods aren't imported to Kotlin as exception-throwing.

Extensions and category members

Members of Objective-C categories and Swift extensions are imported to Kotlin as extensions. That's why these declarations can't be overridden in Kotlin. And the extension initializers aren't available as Kotlin constructors.

Kotlin extensions to "regular" Kotlin classes are imported to Swift and Objective-C as extensions and category members respectively. Kotlin extensions to other types are treated as top-level declarations with an additional receiver

parameter. These types include:

- Kotlin String type
- Kotlin collection types and subtypes
- Kotlin interface types
- Kotlin primitive types
- Kotlin inline classes
- Kotlin Any type
- Kotlin function types and subtypes
- Objective-C classes and protocols

Kotlin singletons

Kotlin singleton (made with an object declaration, including companion object) is imported to Swift/Objective-C as a class with a single instance. The instance is available through the factory method, i.e. as [MySingleton mySingleton] in Objective-C and MySingleton() in Swift.

NSNumber

Kotlin primitive type boxes are mapped to special Swift/Objective-C classes. For example, kotlin.Int box is represented as KotlinInt class instance in Swift (or \${prefix}Int instance in Objective-C, where prefix is the framework names prefix). These classes are derived from NSNumber, so the instances are proper NSNumber s supporting all corresponding operations.

NSNumber type is not automatically translated to Kotlin primitive types when used as a Swift/Objective-C parameter type or return value. The reason is that NSNumber type doesn't provide enough information about a wrapped primitive value type, i.e. NSNumber is statically not known to be a e.g. Byte, Boolean, or Double. So Kotlin primitive values should be cast to/from NSNumber manually ([see below](#)).

NSMutableString

NSMutableString Objective-C class is not available from Kotlin. All instances of NSMutableString are copied when passed to Kotlin.

Collections

Kotlin collections are converted to Swift/Objective-C collections as described in the table above. Swift/Objective-C collections are mapped to Kotlin in the same way, except for NSMutableSet and NSMutableDictionary. NSMutableSet isn't converted to a Kotlin MutableSet. To pass an object for Kotlin MutableSet, you can create this kind of Kotlin collection explicitly by either creating it in Kotlin with e.g. mutableSetOf(), or using the KotlinMutableSet class in Swift (or \${prefix}MutableSet in Objective-C, where prefix is the framework names prefix). The same holds for MutableMap.

Function types

Kotlin function-typed objects (e.g. lambdas) are converted to Swift functions / Objective-C blocks. However there is a difference in how types of parameters and return values are mapped when translating a function and a function type. In the latter case primitive types are mapped to their boxed representation. Kotlin Unit return value is represented as a corresponding Unit singleton in Swift/Objective-C. The value of this singleton can be retrieved in the same way as it is for any other Kotlin object (see singletons in the table above). To sum the things up:

```
fun foo(block: (Int) -> Unit) { ... }
```

would be represented in Swift as

```
func foo(block: (KotlinInt) -> KotlinUnit)
```

and can be called like

```
foo {  
    bar($0 as! Int32)  
    return KotlinUnit()  
}
```

Generics

Objective-C supports "lightweight generics" defined on classes, with a relatively limited feature set. Swift can import generics defined on classes to help provide additional type information to the compiler.

Generic feature support for Objective-C and Swift differ from Kotlin, so the translation will inevitably lose some information, but the features supported retain meaningful information.

Limitations

Objective-C generics do not support all features of either Kotlin or Swift, so there will be some information lost in the translation.

Generics can only be defined on classes, not on interfaces (protocols in Objective-C and Swift) or functions.

Nullability

Kotlin and Swift both define nullability as part of the type specification, while Objective-C defines nullability on methods and properties of a type. As such, the following:

```
class Sample<T>() {  
    fun myVal(): T  
}
```

will (logically) look like this:

```
class Sample<T>() {  
    fun myVal(): T?  
}
```

In order to support a potentially nullable type, the Objective-C header needs to define `myVal` with a nullable return value.

To mitigate this, when defining your generic classes, if the generic type should never be null, provide a non-null type constraint:

```
class Sample<T : Any>() {  
    fun myVal(): T  
}
```

That will force the Objective-C header to mark `myVal` as non-null.

Variance

Objective-C allows generics to be declared covariant or contravariant. Swift has no support for variance. Generic classes coming from Objective-C can be force-cast as needed.

```
data class SomeData(val num: Int = 42) : BaseData()  
class GenVarOut<out T : Any>(val arg: T)  
  
let vari0ut = GenVarOut<SomeData>(arg: sd)  
let vari0utAny : GenVarOut<BaseData> = vari0ut as! GenVarOut<BaseData>
```

Constraints

In Kotlin you can provide upper bounds for a generic type. Objective-C also supports this, but that support is unavailable in more complex cases, and is currently not supported in the Kotlin - Objective-C interop. The exception here being a non-null upper bound will make Objective-C methods/properties non-null.

To disable

To have the framework header written without generics, add the flag to the compiler config:

```
binaries.framework {  
    freeCompilerArgs += "-Xno-objc-generics"  
}
```

Casting between mapped types

When writing Kotlin code, an object may need to be converted from a Kotlin type to the equivalent Swift/Objective-C type (or vice versa). In this case a plain old Kotlin cast can be used, e.g.

```
val nsArray = listOf(1, 2, 3) as NSArray
val string = nsString as String
val nsNumber = 42 as NSNumber
```

Subclassing

Subclassing Kotlin classes and interfaces from Swift/Objective-C

Kotlin classes and interfaces can be subclassed by Swift/Objective-C classes and protocols.

Subclassing Swift/Objective-C classes and protocols from Kotlin

Swift/Objective-C classes and protocols can be subclassed with a Kotlin final class. Non- final Kotlin classes inheriting Swift/Objective-C types aren't supported yet, so it is not possible to declare a complex class hierarchy inheriting Swift/Objective-C types.

Normal methods can be overridden using the `override` Kotlin keyword. In this case the overriding method must have the same parameter names as the overridden one.

Sometimes it is required to override initializers, e.g. when subclassing `UIViewController`. Initializers imported as Kotlin constructors can be overridden by Kotlin constructors marked with the `@OverrideInit` annotation:

```
class ViewController : UIViewController {
    @OverrideInit constructor(coder: NSCoder) : super(coder)

    ...
}
```

The overriding constructor must have the same parameter names and types as the overridden one.

To override different methods with clashing Kotlin signatures, you can add a `@Suppress("CONFLICTING_OVERLOADS")` annotation to the class.

By default the Kotlin/Native compiler doesn't allow calling a non-designated Objective-C initializer as a `super(...)` constructor. This behaviour can be inconvenient if the designated initializers aren't marked properly in the Objective-C library. Adding a `disableDesignatedInitializerChecks = true` to the `.def` file for this library would disable these compiler checks.

C features

See [Interoperability with C](#) for an example case where the library uses some plain C features, such as unsafe pointers, structs, and so on.

Unsupported

Some features of Kotlin programming language are not yet mapped into respective features of Objective-C or Swift. Currently, following features are not properly exposed in generated framework headers:

- inline classes (arguments are mapped as either underlying primitive type or id)
- custom classes implementing standard Kotlin collection interfaces (List, Map, Set) and other special classes
- Kotlin subclasses of Objective-C classes

CocoaPods integration

Kotlin/Native provides integration with the [CocoaPods dependency manager](#). You can add dependencies on Pod libraries as well as use a multiplatform project with native targets as a CocoaPods dependency (Kotlin Pod).

You can manage Pod dependencies directly in IntelliJ IDEA and enjoy all the additional features such as code highlighting and completion. You can build the whole Kotlin project with Gradle and not ever have to switch to Xcode.

Use Xcode only when you need to write Swift/Objective-C code or run your application on a simulator or device. To work correctly with Xcode, you should [update your Podfile](#).

Depending on your project and purposes, you can add dependencies between a [Kotlin project and a Pod library](#) as well as [a Kotlin Pod and an Xcode project](#).

You can also add dependencies between a Kotlin Pod and multiple Xcode projects. However, in this case you need to add a dependency by calling pod install manually for each Xcode project. In other cases, it's done automatically.

Install the CocoaPods dependency manager and plugin

1. Install the [CocoaPods dependency manager](#).

```
$ sudo gem install cocoapods
```

2. Install the [cocoapods-generate](#) plugin.

```
$ sudo gem install cocoapods-generate
```

3. In build.gradle.kts (or build.gradle) of your IDEA project, apply the CocoaPods plugin as well as the Kotlin Multiplatform plugin.

```
plugins {
    kotlin("multiplatform") version "1.5.20"
    kotlin("native.cocoapods") version "1.5.20"
}
```

4. Configure summary, homepage, and frameworkName of the Podspec file in the cocoapods block. version is a version of the Gradle project.

```
plugins {
    kotlin("multiplatform") version "1.5.20"
    kotlin("native.cocoapods") version "1.5.20"
}

// CocoaPods requires the podspec to have a version.
version = "1.0"

kotlin {
    cocoapods {
        // Configure fields required by CocoaPods.
        summary = "Some description for a Kotlin/Native module"
        homepage = "Link to a Kotlin/Native module homepage"

        // You can change the name of the produced framework.
        // By default, it is the name of the Gradle project.
        frameworkName = "my_framework"
    }
}
```

5. Re-import the project.

6. Generate the [Gradle wrapper](#) to avoid compatibility issues during an Xcode build.

When applied, the CocoaPods plugin does the following:

- Adds both debug and release frameworks as output binaries for all macOS, iOS, tvOS, and watchOS targets.
- Creates a podspec task which generates a [Podspec](#) file for the project.

The Podspec file includes a path to an output framework and script phases that automate building this framework during the build process of an Xcode project.

Add dependencies on Pod libraries

To add dependencies between a Kotlin project and a Pod library, you should [complete the initial configuration](#). This

allows you to add dependencies on the following types of Pod libraries:

- [A Pod library from the CocoaPods repository](#)
- [A Pod library stored locally](#)
- [A Pod library from a Git repository](#)
- [A Pod library from an archive](#)
- [A Pod library from a custom Podspec repository](#)
- [A Pod library with custom cinterop options](#)
- [A static Pod library](#)

A Kotlin project requires the `pod()` function call in `build.gradle.kts` (`build.gradle`) for adding a Pod dependency. Each dependency requires its own separate function call. You can specify the parameters for the dependency in the configuration block of the function.

When you add a new dependency and re-import the project in IntelliJ IDEA, the new dependency will be added automatically. No additional steps are required.

To use your Kotlin project with Xcode, you should [make changes in your project Podfile](#).

Add a dependency on a Pod library from the CocoaPods repository

You can add dependencies on a Pod library from the CocoaPods repository with `pod()` to `build.gradle.kts` (`build.gradle`) of your project:

1. Specify the name of a Pod library in the `pod()` function. In the configuration block you can specify the version of the library using the `version` parameter. To use the latest version of the library, you can just omit this parameter all-together.

You can add dependencies on subspecs.

2. Specify the minimum deployment target version for the Pod library.

If you don't specify the minimum deployment target version and a dependency Pod requires a higher deployment target, you will get an error.

```
kotlin {  
    ios()  
  
    cocoapods {  
        ios.deploymentTarget = "13.5"
```

```
summary = "CocoaPods test library"
homepage = "https://github.com/JetBrains/kotlin"

pod("AFNetworking") {
    version = "~> 4.0.1"
}
}
```

3. Re-import the project.

To use these dependencies from the Kotlin code, import the packages `cocoapods.<library-name>`.

```
import cocoapods.AFNetworking.*
```

You can find a sample project [here](#).

Add a dependency on a Pod library stored locally

You can add a dependency on a Pod library stored locally with `pod()` to `build.gradle.kts` (`build.gradle`) of your project:

1. Specify the name of a Pod library in the `pod()` function. In the configuration block specify the path to the local Pod library: use the `path()` function in the `source` parameter value.

You can add local dependencies on subspecs as well. The `cocoapods` block can include dependencies to Pods stored locally and Pods from the CocoaPods repository at the same time.

2. Specify the minimum deployment target version for the Pod library.

If you don't specify the minimum deployment target version and a dependency Pod requires a higher deployment target, you will get an error.

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("pod_dependency") {
            version = "1.0"
            source = path(project.file("../pod_dependency/pod_dependency.podspec"))
        }
    }
}
```

```

    pod("subspec_dependency/Core") {
        version = "1.0"
        source = path(project.file("../subspec_dependency/subspec_dependency.podspec"))
    }
    pod("AFNetworking") {
        version = "~> 4.0.1"
    }
}
}

```

You can also specify the version of the library using version parameter in the configuration block. To use the latest version of the library, omit the parameter.

3. Re-import the project.

To use these dependencies from the Kotlin code, import the packages `cocoapods.<library-name>`.

```

import cocoapods.pod_dependency.*
import cocoapods.subspec_dependency.*
import cocoapods.AFNetworking.*

```

You can find a sample project [here](#).

Add a dependency on a Pod library from the Git repository

You can add dependencies on a Pod library from a custom Git repository with `pod()` to `build.gradle.kts` (`build.gradle`) of your project:

1. Specify the name of a Pod library in the `pod()` function. In the configuration block specify the path to the git repository: use the `git()` function in the `source` parameter value.

Additionally, you can specify the following parameters in the block after `git()`:

- `commit` – to use a specific commit from the repository
- `tag` – to use a specific tag from the repository
- `branch` – to use a specific branch from the repository

The `git()` function prioritizes passed parameters in the following order: `commit`, `tag`, `branch`. If you don't specify a parameter, the Kotlin plugin uses `HEAD` from the master branch.

You can combine `branch`, `commit`, and `tag` parameters to get the specific version of a Pod.

2. Specify the minimum deployment target version for the Pod library.

If you don't specify the minimum deployment target version and a dependency Pod requires a higher deployment target, you will get an error.

```
kotlin {  
    ios()  
  
    cocoapods {  
        summary = "CocoaPods test library"  
        homepage = "https://github.com/JetBrains/kotlin"  
  
        ios.deploymentTarget = "13.5"  
  
        pod("AFNetworking") {  
            source = git("https://github.com/AFNetworking/AFNetworking") {  
                tag = "4.0.0"  
            }  
        }  
  
        pod("JSONModel") {  
            source = git("https://github.com/jsonmodel/jsonmodel.git") {  
                branch = "key-mapper-class"  
            }  
        }  
  
        pod("CocoaLumberjack") {  
            source = git("https://github.com/CocoaLumberjack/CocoaLumberjack.git") {  
                commit = "3e7f595e3a459c39b917aacf9856cd2a48c4dbf3"  
            }  
        }  
    }  
}
```

3. Re-import the project.

To work correctly with Xcode, you should specify the path to the Podspec in your Podfile. For example:

```
target 'ios-app' do  
    # ... other pod dependencies ...  
    pod 'JSONModel', :path => '../cocoapods/kotlin-with-cocoapods-sample/kotlin-  
library/build/cocoapods/externalSources/git/JSONModel'  
end
```

To use these dependencies from the Kotlin code, import the packages `cocoapods.<library-name>`.

```
import cocoapods.AFNetworking.*  
import cocoapods.JSONModel.*  
import cocoapods.CocoaLumberjack.*
```

You can find a sample project [here](#).

Add a dependency on a Pod library from an archive

You can add dependencies on a Pod library from zip, tar, or jar archive with pod() to build.gradle.kts (build.gradle) of your project:

1. Specify the name of a Pod library in the pod() function. In the configuration block specify the path to the archive: use the url() function with an arbitrary HTTP address in the source parameter value.

Additionally, you can specify the boolean flatten parameter as a second argument for the url() function. This parameter indicates that all the Pod files are located in the root directory of the archive.

2. Specify the minimum deployment target version for the Pod library.

If you don't specify the minimum deployment target version and a dependency Pod requires a higher deployment target, you will get an error.

```
kotlin {  
    ios()  
  
    cocoapods {  
        summary = "CocoaPods test library"  
        homepage = "https://github.com/JetBrains/kotlin"  
  
        ios.deploymentTarget = "13.5"  
  
        pod("pod_dependency") {  
            source = url("https://github.com/Kotlin/kotlin-with-cocoapods-  
sample/raw/cocoapods-zip/cocoapodSourcesZip.zip", flatten = true)  
        }  
    }  
}
```

3. Re-import the project.

To work correctly with Xcode, you should specify the path to the Podspec in your Podfile. For example:

```
target 'ios-app' do  
    # ... other pod dependencies ...  
    pod 'podspecWithFilesExample', :path => '../cocoapods/kotlin-with-cocoapods-  
sample/pod_dependency'  
end
```

To use these dependencies from the Kotlin code, import the packages `cocoapods.<library-name>`.

```
import cocoapods.pod_dependency.*
```

You can find a sample project [here](#).

Add a dependency on a Pod library from a custom Podspec repository

You can add dependencies on a Pod library from a custom Podspec repository with pod() and specRepos to build.gradle.kts (build.gradle) of your project:

1. Specify the HTTP address to the custom Podspec repository using the url() inside the specRepos block.
2. Specify the name of a Pod library in the pod() function.
3. Specify the minimum deployment target version for the Pod library.

If you don't specify the minimum deployment target version and a dependency Pod requires a higher deployment target, you will get an error.

```
kotlin {  
    ios()  
  
    cocoapods {  
        summary = "CocoaPods test library"  
        homepage = "https://github.com/JetBrains/kotlin"  
  
        ios.deploymentTarget = "13.5"  
  
        specRepos {  
            url("https://github.com/Kotlin/kotlin-cocoapods-spec.git")  
        }  
        pod("example")  
    }  
}
```

4. Re-import the project.

To work correctly with Xcode, you should specify the location of specs at the beginning of your Podfile. For example,

```
source 'https://github.com/Kotlin/kotlin-cocoapods-spec.git'
```

You should also specify the path to the Podspec in your Podfile. For example:

```
target 'ios-app' do
    # ... other pod dependencies ...
    pod 'podspecWithFilesExample', :path => '../cocoapods/kotlin-with-cocoapods-
sample/pod_dependency'
end
```

To use these dependencies from the Kotlin code, import the packages cocoapods.<library-name>.

```
import cocoapods.example.*
```

You can find a sample project [here](#).

Add a dependency on a Pod library with custom cinterop options

You can add dependencies on a Pod library with custom cinterop options with pod() to build.gradle.kts (build.gradle) of your project:

1. Specify the name of a Pod library in the pod() function. In the configuration block specify the cinterop options:

- extraOpts – to specify the list of options for a Pod library. For example, specific flags: extraOpts = listOf("-compiler-option")
- packageName – to specify the package name. If you specify this, you can import the library using the package name: import <packageName>.

2. Specify the minimum deployment target version for the Pod library.

If you don't specify the minimum deployment target version and a dependency Pod requires a higher deployment target, you will get an error.

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"
```

```
ios.deploymentTarget = "13.5"

useLibraries()

pod("YandexMapKit") {
    packageName = "YandexMK"
}
}
```

3. Re-import the project.

To use these dependencies from the Kotlin code, import the packages `cocoapods.<library-name>`.

```
import cocoapods.YandexMapKit.*
```

If you use the `packageName` parameter, you can import the library using the package name `import <packageName>`:

```
import YandexMK.YMKPoint
import YandexMK.YMKDistance
```

Add a dependency on a static Pod library

You can add dependencies on a static Pod library with `pod()` and `useLibraries()` to `build.gradle.kts` (`build.gradle`) of your project:

1. Specify the name of the library using the `pod()` function.
2. Call the `useLibraries()` function – it enables a special flag for static libraries.
3. Specify the minimum deployment target version for the Pod library.

If you don't specify the minimum deployment target version and a dependency Pod requires a higher deployment target, you will get an error.

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("YandexMapKit") {
            version = "~> 3.2"
        }
    }
}
```

```
        useLibraries()  
    }  
}
```

4. Re-import the project.

To use these dependencies from the Kotlin code, import the packages `cocoapods.<library-name>`.

```
import cocoapods.YandexMapKit.*
```

Update Podfile for Xcode

If you want to import your Kotlin project in an Xcode project, you'll need to make some changes to your Podfile for it to work correctly:

- If your project has any Git, HTTP, or custom Podspec repository dependencies, you should also specify the path to the Podspec in the Podfile.

For example, if you add a dependency on `podspecWithFilesExample`, declare the path to the Podspec in the Podfile:

```
target 'ios-app' do  
    # ... other dependencies ...  
    pod 'podspecWithFilesExample', :path =>  
    'cocoapods/externalSources/url/podspecWithFilesExample'  
end
```

The `:path` should contain the filepath to the Pod.

- When you add a library from the custom Podspec repository, you should also specify the location of specs at the beginning of your Podfile:

```
source 'https://github.com/Kotlin/kotlin-cocoapods-spec.git'  
  
target 'kotlin-cocoapods-xcproj' do  
    # ... other dependencies ...  
    pod 'example'  
end
```

Re-import the project after making changes in Podfile.

If you don't make these changes to the Podfile, the `podInstall` task will fail and the CocoaPods plugin will show an error message in the log.

Check out the `withXcproject` branch of the [sample project](#), which contains an example of Xcode integration with an existing Xcode project named `kotlin-cocoapods-xcproj`.

Use a Kotlin Gradle project as a CocoaPods dependency

You can use a Kotlin Multiplatform project with native targets as a CocoaPods dependency (Kotlin Pod). You can include such a dependency in the Podfile of the Xcode project by its name and path to the project directory containing the generated Podspec. This dependency will be automatically built (and rebuilt) along with this project. Such an approach simplifies importing to Xcode by removing a need to write the corresponding Gradle tasks and Xcode build steps manually.

You can add dependencies between:

- [A Kotlin Pod and an Xcode project with one target](#)
- [A Kotlin Pod and an Xcode project with several targets](#)

To correctly import the dependencies into the Kotlin/Native module, the Podfile must contain either `use_modular_headers!` or `use_frameworks!` directive.

Add a dependency between a Kotlin Pod and Xcode project with one target

1. Create an Xcode project with a Podfile if you haven't done so yet.
2. Add the path to your Xcode project Podfile with `podfile = project.file(..)` to `build.gradle.kts` (`build.gradle`) of your Kotlin project. This step helps synchronize your Xcode project with Kotlin Pod dependencies by calling `pod install` for your Podfile.
3. Specify the minimum target version for the Pod library.

If you don't specify the minimum target version and a dependency Pod requires a higher deployment target, you will get an error.

```
kotlin {  
    ios()  
  
    cocoapods {  
        summary = "CocoaPods test library"  
        homepage = "https://github.com/JetBrains/kotlin"  
        ios.deploymentTarget = "13.5"  
        pod("AFNetworking") {  
            version = "~> 4.0.0"  
        }  
        podfile = project.file("../ios-app/Podfile")  
    }  
}
```

4. Add the name and path of the Kotlin Pod you want to include in the Xcode project to Podfile.

```
use_frameworks!

platform :ios, '13.5'

target 'ios-app' do
    pod 'kotlin_library', :path => '../kotlin-library'
end
```

5. Re-import the project.

Add a dependency between a Kotlin Pod with an Xcode project with several targets

1. Create an Xcode project with a Podfile if you haven't done so yet.
2. Add the path to your Xcode project Podfile with `podfile = project.file(..)` to `build.gradle.kts` (`build.gradle`) of your Kotlin project. This step helps synchronize your Xcode project with Kotlin Pod dependencies by calling `pod install` for your Podfile.
3. Add dependencies to the Pod libraries that you want to use in your project with `pod()`.
4. For each target, specify the minimum target version for the Pod library.

If you don't specify the minimum deployment target version and a dependency Pod requires a higher deployment target, you will get an error.

```
kotlin {
    ios()
    tvos()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"
        ios.deploymentTarget = "13.5"
        tvos.deploymentTarget = "13.4"

        pod("AFNetworking") {
            version = "~> 4.0.0"
        }
        podfile = project.file("../severalTargetsXcodeProject/Podfile") // specify the path
    }
}
```

5. Add the name and path of the Kotlin Pod you want to include in the Xcode project to the Podfile.

```
target 'iosApp' do
```

```

use_frameworks!
platform :ios, '13.5'
# Pods for iosApp
pod 'kotlin_library', :path => '../kotlin-library'
end

target 'TVosApp' do
  use_frameworks!
  platform :tvos, '13.4'

  # Pods for TVosApp
  pod 'kotlin_library', :path => '../kotlin-library'
end

```

8. Re-import the project.

You can find a sample project [here](#).

Kotlin/Native as an Apple framework – tutorial

Kotlin/Native provides bi-directional interoperability with Objective-C/Swift. Objective-C frameworks and libraries can be used in Kotlin code. Kotlin modules can be used in Swift/Objective-C code too. Besides that, Kotlin/Native has [C Interop](#). There is also the [Kotlin/Native as a Dynamic Library](#) tutorial for more information.

In this tutorial, you will see how to use Kotlin/Native code from Objective-C and Swift applications on macOS and iOS.

In this tutorial you'll:

- [create a Kotlin Library](#) and compile it to a framework
- examine the generated [Objective-C and Swift API](#) code
- use the framework from [Objective-C and Swift](#)
- [Configure Xcode](#) to use the framework for [macOS](#) and [iOS](#)

Create a Kotlin library

The Kotlin/Native compiler can produce a framework for macOS and iOS out of the Kotlin code. The created framework contains all declarations and binaries needed to use it with Objective-C and Swift. The best way to understand the techniques is to try it for ourselves. Let's create a tiny Kotlin library first and use it from an Objective-C program.

Create the hello.kt file with the library contents:

```

package example

object Object {

```

```

    val field = "A"
}

interface Interface {
    fun iMember() {}
}

class Clazz : Interface {
    fun member(p: Int): ULong? = 42UL
}

fun forIntegers(b: Byte, s: UShort, i: Int, l: ULong?) { }
fun forFloats(f: Float, d: Double?) { }

fun strings(str: String?) : String {
    return "That is '$str' from C"
}

fun acceptFun(f: (String) -> String?) = f("Kotlin/Native rocks!")
fun supplyFun() : (String) -> String? = { "$it is cool!" }

```

While it is possible to use the command line, either directly or by combining it with a script file (such as .sh or .bat file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is therefore better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the [A Basic Kotlin/Native Application](#) tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and [multiplatform](#) builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following build.gradle(.kts) Gradle build file:

Groovy

```

plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.5.20'
}

repositories {
    mavenCentral()
}

kotlin {
    macosX64("native") {
        binaries {
            framework {
                baseUrl = "Demo"
            }
        }
    }
}

```

```

        }
    }
}

wrapper {
    gradleVersion = "6.7.1"
    distributionType = "ALL"
}

```

Kotlin

```

plugins {
    kotlin("multiplatform") version "1.5.20"
}

repositories {
    mavenCentral()
}

kotlin {
    macosX64("native") {
        binaries {
            framework {
                basePath = "Demo"
            }
        }
    }
}

tasks.wrapper {
    gradleVersion = "6.7.1"
    distributionType = Wrapper.DistributionType.ALL
}

```

The prepared project sources can be directly downloaded from GitHub:

- [Groovy](#)
- [Kotlin](#)

Move the sources file into the `src/nativeMain/kotlin` folder under the project. That is the default path, where sources are located, when the `kotlin-multiplatform` plugin is used. Use the following block to configure the project to generate a dynamic or shared library:

```

binaries {
    framework {
        basePath = "Demo"
    }
}

```

Along with macOS X64, Kotlin/Native supports iOS arm32, arm64 and X64 targets. You may replace the `macosX64` with respective functions as shown in the table:

Target platform/device	Gradle function
macOS x86_64	macosX64()
iOS ARM 32	iosArm32()
iOS ARM 64	iosArm64()
iOS Simulator (x86_64)	iosX64()

Run the linkNative Gradle task to build the library [in the IDE](#) or by calling the following console command:

```
./gradlew linkNative
```

Depending on the variant, the build generates the framework into the build/bin/native/debugFramework and build/bin/native/releaseFramework folders. Let's see what is inside.

Generated framework headers

Each of the created frameworks contains the header file in <Framework>/Headers/Demo.h. The headers do not depend on the target platform (at least with Kotlin/Native v.0.9.2). It contains the definitions for our Kotlin code and a few Kotlin-wide declarations.

The way Kotlin/Native exports symbols is subject to change without notice.

Kotlin/Native runtime declarations

Take a look at Kotlin runtime declarations:

```
NS_ASSUME_NONNULL_BEGIN

@interface KotlinBase : NSObject
- (instancetype)init __attribute__((unavailable));
+ (instancetype)new __attribute__((unavailable));
+ (void)initialize __attribute__((objc_requires_super));
@end;

@interface KotlinBase (KotlinBaseCopying) <NSCopying>
@end;

__attribute__((objc_runtime_name("KotlinMutableSet")))
__attribute__((swift_name("KotlinMutableSet")))
@interface DemoMutableSet<ObjectType> : NSMutableSet<ObjectType>
@end;
```

```

__attribute__((objc_runtime_name("KotlinMutableDictionary")))
__attribute__((swift_name("KotlinMutableDictionary")))
@interface DemoMutableDictionary<KeyType, ObjectType> : NSMutableDictionary<KeyType,
ObjectType>
@end;

@interface NSError (NSErrorKotlinException)
@property (readonly) id _Nullable kotlinException;
@end;

```

Kotlin classes have a KotlinBase base class in Objective-C, the class extends the NSObject class there. There are also have wrappers for collections and exceptions. Most of the collection types are mapped to similar collection types from the other side:

Kotlin	Swift	Objective-C
List	Array	NSArray
MutableList	NSMutableArray	NSMutableArray
Set	Set	NSSet
Map	Dictionary	NSDictionary
MutableMap	NSMutableDictionary	NSMutableDictionary

Kotlin numbers and NSNumber

The next part of the <Framework>/Headers/Demo.h contains number type mappings between Kotlin/Native and NSNumber. There is the base class called DemoNumber in Objective-C and KotlinNumber in Swift. It extends NSNumber. There are also child classes per Kotlin number type:

Kotlin	Swift	Objective-C	Simple type
-	KotlinNumber	<Package>Number	-
Byte	KotlinByte	<Package>Byte	char
UByte	KotlinUByte	<Package>UByte	unsigned char
Short	KotlinShort	<Package>Short	short
UShort	KotlinUShort	<Package>UShort	unsigned short

Kotlin	Swift	Objective-C	Simple type
Int	KotlinInt	<Package>Int	int
UInt	KotlinUInt	<Package>UInt	unsigned int
Long	KotlinLong	<Package>Long	long long
ULong	KotlinULong	<Package>ULong	unsigned long long
Float	KotlinFloat	<Package>Float	float
Double	KotlinDouble	<Package>Double	double
Boolean	KotlinBoolean	<Package>Boolean	BOOL/Bool

Every number type has a class method to create a new instance from the related simple type. Also, there is an instance method to extract a simple value back. Schematically, declarations look like that:

```
__attribute__((objc_runtime_name("Kotlin__TYPE_")))
__attribute__((swift_name("Kotlin__TYPE_")))
@interface Demo__TYPE_ : DemoNumber
- (instancetype)initWith__TYPE_:(__CTYPE__)
+ (instancetype)numberWith__TYPE_:(__CTYPE__)
@end;
```

Where `__TYPE_` is one of the simple type names and `__CTYPE` is the related Objective-C type, for example, `initWithChar(char)`.

These types are used to map boxed Kotlin number types into Objective-C and Swift. In Swift, you may simply call the constructor to create an instance, for example, `KotlinLong(value: 42)`.

Classes and objects from Kotlin

Let's see how class and object are mapped to Objective-C and Swift. The generated `<Framework>/Headers/Demo.h` file contains the exact definitions for Class, Interface, and Object:

```
NS_ASSUME_NONNULL_BEGIN

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Object")))
@interface DemoObject : KotlinBase
+ (instancetype)alloc __attribute__((unavailable));
+ (instancetype)allocWithZone:(struct _NSZone *)zone __attribute__((unavailable));
+ (instancetype)object __attribute__((swift_name("init())));
@property (readonly) NSString *field;
@end;
```

```

__attribute__((swift_name("Interface")))
@protocol DemoInterface
@required
- (void)iMember __attribute__((swift_name("iMember())));
@end;

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Clazz")))
@interface DemoClazz : KotlinBase <DemoInterface>
- (instancetype)init __attribute__((swift_name("init())))
__attribute__((objc_designated_initializer));
+ (instancetype)new __attribute__((availability(swift, unavailable, message="use object
initializers instead")));
- (DemoLong * _Nullable)memberP:(int32_t)p __attribute__((swift_name("member(p:)")));
@end;

```

The code is full of Objective-C attributes, which are intended to help the use of the framework from both Objective-C and Swift languages. DemoClazz, DemoInterface, and DemoObject are created for Clazz, Interface, and Object respectively. The Interface is turned into @protocol, both a class and an object are represented as @interface. The Demo prefix comes from the -output parameter of the kotlinc-native compiler and the framework name. You can see here that the nullable return type ULong? is turned into DemoLong* in Objective-C.

Global declarations from Kotlin

All global functions from Kotlin are turned into DemoLibKt in Objective-C and into LibKt in Swift, where Demo is the framework name and set by the -output parameter of kotlinc-native.

```

NS_ASSUME_NONNULL_BEGIN

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("LibKt")))
@interface DemoLibKt : KotlinBase
+ (void)forIntegersB:(int8_t)b s:(int16_t)s i:(int32_t)i l:(DemoLong * _Nullable)l
__attribute__((swift_name("forIntegers(b:s:i:l:)")));
+ (void)forFloatsF:(float)f d:(DemoDouble * _Nullable)d
__attribute__((swift_name("forFloats(f:d:)")));
+ (NSString *)stringsStr:(NSString * _Nullable)str
__attribute__((swift_name("strings(str:)")));
+ (NSString * _Nullable)acceptFunF:(NSString * _Nullable (^)(NSString *))f
__attribute__((swift_name("acceptFun(f:)")));
+ (NSString * _Nullable (^)(NSString *))supplyFun __attribute__((swift_name("supplyFun())));
@end;

```

You see that Kotlin String and Objective-C NSString* are mapped transparently. Similarly, Unit type from Kotlin is mapped to void. We see primitive types are mapped directly. Non-nullable primitive types are mapped transparently. Nullable primitive types are mapped into Kotlin<TYPE>* types, as shown in the table [above](#). Both higher order functions acceptFunF and supplyFun are included, and accept Objective-C blocks.

More information about all other types mapping details can be found in the [Objective-C Interop](#) documentation article

Garbage collection and reference counting

Objective-C and Swift use reference counting. Kotlin/Native has its own garbage collection too. Kotlin/Native garbage collection is integrated with Objective-C/Swift reference counting. You do not need to use anything special to control the lifetime of Kotlin/Native instances from Swift or Objective-C.

Use the code from Objective-C

Let's call the framework from Objective-C. For that, create the main.m file with the following content:

```
#import <Foundation/Foundation.h>
#import <Demo/Demo.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        [[DemoObject object] field];

        DemoClazz* clazz = [[ DemoClazz alloc] init];
        [clazz memberP:42];

        [DemoLibKt forIntegersB:1 s:1 i:3 l:[DemoULong numberWithUnsignedLongLong:4]];
        [DemoLibKt forIntegersB:1 s:1 i:3 l:nil];

        [DemoLibKt forFloatsF:2.71 d:[DemoDouble numberWithDouble:2.71]];
        [DemoLibKt forFloatsF:2.71 d:nil];

        NSString* ret = [DemoLibKt acceptFunF:^(NSString * _Nullable(it) {
            return [it stringByAppendingString:@" Kotlin is fun"];
        }];
        NSLog(@"%@", ret);
        return 0;
    }
}
```

Here you call Kotlin classes directly from Objective-C code. A Kotlin object has the class method function object, which allows us to get the only instance of the object and to call Object methods on it. The widespread pattern is used to create an instance of the Clazz class. You call the [[DemoClazz alloc] init] on Objective-C. You may also use [DemoClazz new] for constructors without parameters. Global declarations from the Kotlin sources are scoped under the DemoLibKt class in Objective-C. All methods are turned into class methods of that class. The strings function is turned into DemoLibKt.stringsStr function in Objective-C, you can pass NSString directly to it. The return is visible as NSString too.

Use the code from Swift

The framework that you compiled with Kotlin/Native has helper attributes to make it easier to use with Swift. Convert the previous Objective-C example into Swift. As a result, you'll have the following code in main.swift:

```

import Foundation
import Demo

let kotlinObject = Object()
assert(kotlinObject == Object(), "Kotlin object has only one instance")

let field = Object().field

let clazz = Clazz()
clazz.member(p: 42)

LibKt.forIntegers(b: 1, s: 2, i: 3, l: 4)
LibKt.forFloats(f: 2.71, d: nil)

let ret = LibKt.acceptFun { "=\"$0\" Kotlin is fun" }
if (ret != nil) {
    print(ret!)
}

```

The Kotlin code is turned into very similar looking code in Swift. There are some small differences, though. In Kotlin any object has only one instance. Kotlin object Object now has a constructor in Swift, and we use the Object() syntax to access the only instance of it. The instance is always the same in Swift, so that Object() === Object() is true. Methods and property names are translated as-is. Kotlin String is turned into Swift String too. Swift hides NSNumber* boxing from us too. We can pass a Swift closure to Kotlin and call a Kotlin lambda function from Swift too.

More documentation on the types mapping can be found in the [Objective-C Interop](#) article.

Xcode and framework dependencies

You need to configure an Xcode project to use our framework. The configuration depends on the target platform.

Xcode for macOS target

First, in the General tab of the target configuration, under the Linked Frameworks and Libraries section, you need to include our framework. This will make Xcode look at our framework and resolve imports both from Objective-C and Swift.

The second step is to configure the framework search path of the produced binary. It is also known as rpath or [run-time search path](#). The binary uses the path to look for the required frameworks. We do not recommend installing additional frameworks to the OS if it is not needed. You should understand the layout of your future application, for example, you may have the Frameworks folder under the application bundle with all the frameworks you use. The @rpath parameter can be configured in Xcode. You need to open the project configuration and find the Runpath Search Paths section. Here you specify the relative path to the compiled framework.

Xcode for iOS targets

First, you need to include the compiled framework in the Xcode project. To do this, add the framework to the Frameworks, Libraries, and Embedded Content section of the General tab of the target configuration page.

The second step is to then include the framework path into the Framework Search Paths section of the Build Settings tab of the target configuration page. It is possible to use the \$(PROJECT_DIR) macro to simplify the setup.

The iOS simulator requires a framework compiled for the ios_x64 target, the iOS_sim folder in our case.

This [Stackoverflow thread](#) contains a few more recommendations. Also, the [CocoaPods](#) package manager may be helpful to automate the process too.

Next steps

Kotlin/Native has bidirectional interop with Objective-C and Swift languages. Kotlin objects integrate with Objective-C/Swift reference counting. Unused Kotlin objects are automatically removed. The [Objective-C Interop](#) article contains more information on the interop implementation details. Of course, it is possible to import an existing framework and use it from Kotlin. Kotlin/Native comes with a good set of pre-imported system frameworks.

Kotlin/Native supports C interop too. Check out the [Kotlin/Native as a Dynamic Library](#) tutorial for that.

Debugging Kotlin/Native

Currently, the Kotlin/Native compiler produces debug info compatible with the DWARF 2 specification, so modern debugger tools can perform the following operations:

- breakpoints
- stepping
- inspection of type information
- variable inspection

Supporting the DWARF 2 specification means that the debugger tool recognizes Kotlin as C89, because before the DWARF 5 specification, there is no identifier for the Kotlin language type in specification.

Produce binaries with debug info with Kotlin/Native compiler

To produce binaries with the Kotlin/Native compiler, use the -g option on the command line.

```
0:b-debugger-fixes:minamoto@unit-703(0)# cat - > hello.kt
fun main(args: Array<String>) {
    println("Hello world")
    println("I need your clothes, your boots and your motorcycle")
}
0:b-debugger-fixes:minamoto@unit-703(0)# dist/bin/konanc -g hello.kt -o terminator
```

```

KtFile: hello.kt
0:b-debugger-fixes:minamoto@unit-703(0)# lldb terminator.kexe
(lldb) target create "terminator.kexe"
Current executable set to 'terminator.kexe' (x86_64).
(lldb) b kfun:main(kotlin.Array<kotlin.String>)
Breakpoint 1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at hello.kt:2,
address = 0x00000001000012e4
(lldb) r
Process 28473 launched: '/Users/minamoto/ws/.git-trees/debugger-fixes/terminator.kexe' (x86_64)
Process 28473 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00000001000012e4 terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) at
hello.kt:2
1     fun main(args: Array<String>) {
-> 2         println("Hello world")
3         println("I need your clothes, your boots and your motocycle")
4     }
(lldb) n
Hello world
Process 28473 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x00000001000012f0 terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) at
hello.kt:3
1     fun main(args: Array<String>) {
2         println("Hello world")
-> 3         println("I need your clothes, your boots and your motocycle")
4     }
(lldb)

```

Breakpoints

Modern debuggers provide several ways to set a breakpoint, see below for a tool-by-tool breakdown:

lldb

- by name

```

(lldb) b -n kfun:main(kotlin.Array<kotlin.String>)
Breakpoint 4: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at
hello.kt:2, address = 0x00000001000012e4

```

-n is optional, this flag is applied by default

- by location (filename, line number)

```

(lldb) b -f hello.kt -l 1
Breakpoint 1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at
hello.kt:2, address = 0x00000001000012e4

```

- by address

```
(lldb) b -a 0x00000001000012e4
Breakpoint 2: address = 0x00000001000012e4
```

- by regex, you might find it useful for debugging generated artifacts, like lambda etc. (where used # symbol in name).

```
3: regex = 'main\\(', locations = 1
  3.1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at hello.kt:2,
address = terminator.kexe[0x00000001000012e4], unresolved, hit count = 0
```

gdb

- by regex

```
(gdb) rbreak main(
Breakpoint 1 at 0x1000109b4
struct ktype:kotlin.Unit &kfun:main(kotlin.Array<kotlin.String>);
```

- by name unusable, because : is a separator for the breakpoint by location

```
(gdb) b kfun:main(kotlin.Array<kotlin.String>)
No source file named kfun.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (kfun:main(kotlin.Array<kotlin.String>)) pending
```

- by location

```
(gdb) b hello.kt:1
Breakpoint 2 at 0x100001704: file /Users/minamoto/ws/.git-trees/hello.kt, line 1.
```

- by address

```
(gdb) b *0x100001704
Note: breakpoint 2 also set at pc 0x100001704.
Breakpoint 3 at 0x100001704: file /Users/minamoto/ws/.git-trees/hello.kt, line 2.
```

Stepping

Stepping functions works mostly the same way as for C/C++ programs.

Variable inspection

Variable inspections for var variables works out of the box for primitive types. For non-primitive types there are custom

pretty printers for lldb in konan_lldb.py:

```
λ cat main.kt | nl
 1 fun main(args: Array<String>) {
 2     var x = 1
 3     var y = 2
 4     var p = Point(x, y)
 5     println("p = $p")
 6 }

 7 data class Point(val x: Int, val y: Int)

λ lldb ./program.kexe -o 'b main.kt:5' -o
(lldb) target create "./program.kexe"
Current executable set to './program.kexe' (x86_64).
(lldb) b main.kt:5
Breakpoint 1: where = program.kexe`kfun:main(kotlin.Array<kotlin.String>) + 289 at main.kt:5,
address = 0x000000000040af11
(lldb) r
Process 4985 stopped
* thread #1, name = 'program.kexe', stop reason = breakpoint 1.1
  frame #0: program.kexe`kfun:main(kotlin.Array<kotlin.String>) at main.kt:5
 2     var x = 1
 3     var y = 2
 4     var p = Point(x, y)
-> 5     println("p = $p")
 6
 7
 8 data class Point(val x: Int, val y: Int)

Process 4985 launched: './program.kexe' (x86_64)
(lldb) fr var
(int) x = 1
(int) y = 2
(ObjHeader *) p = 0x00000000007643d8
(lldb) command script import dist/tools/konan_lldb.py
(lldb) fr var
(int) x = 1
(int) y = 2
(ObjHeader *) p = [x: ..., y: ...]
(lldb) p p
(ObjHeader *) $2 = [x: ..., y: ...]
(lldb) script lldb.frame.FindVariable("p").GetChildMemberWithName("x").Dereference().GetValue()
'1'
(lldb)
```

Getting representation of the object variable (var) could also be done using the built-in runtime function Konan_DebugPrint (this approach also works for gdb, using a module of command syntax):

```
0:b-debugger-fixes:minamoto@unit-703(0)# cat ./debugger-plugin/1.kt | nl -p
 1 fun foo(a:String, b:Int) = a + b
 2 fun one() = 1
 3 fun main(arg:Array<String>) {
 4     var a_variable = foo("(a_variable) one is ", 1)
 5     var b_variable = foo("(b_variable) two is ", 2)
 6     var c_variable = foo("(c_variable) two is ", 3)
 7     var d_variable = foo("(d_variable) two is ", 4)
```

```

8     println(a_variable)
9     println(b_variable)
10    println(c_variable)
11    println(d_variable)
12 }

0:b-debugger-fixes:minamoto@unit-703(0) # lldb ./program.kexe -o 'b -f 1.kt -l 9' -o r
(lldb) target create "./program.kexe"
Current executable set to './program.kexe' (x86_64).
(lldb) b -f 1.kt -l 9
Breakpoint 1: where = program.kexe`kfun:main(kotlin.Array<kotlin.String>) + 463 at 1.kt:9,
address = 0x0000000100000dbf
(lldb) r
(a_variable) one is 1
Process 80496 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
  frame #0: 0x0000000100000dbf program.kexe`kfun:main(kotlin.Array<kotlin.String>) at 1.kt:9
6       var c_variable = foo("(c_variable) two is ", 3)
7       var d_variable = foo("(d_variable) two is ", 4)
8       println(a_variable)
-> 9       println(b_variable)
10      println(c_variable)
11      println(d_variable)
12 }

Process 80496 launched: './program.kexe' (x86_64)
(lldb) expression -- (int32_t)Konan_DebugPrint(a_variable)
(a_variable) one is 1(int32_t) $0 = 0
(lldb)

```

Known issues

- performance of Python bindings.

Symbolicating iOS crash reports

Debugging an iOS application crash sometimes involves analyzing crash reports. More info about crash reports can be found in the [Apple documentation](#).

Crash reports generally require symbolication to become properly readable: symbolication turns machine code addresses into human-readable source locations. The document below describes some specific details of symbolicating crash reports from iOS applications using Kotlin.

Producing .dSYM for release Kotlin binaries

To symbolicate addresses in Kotlin code (e.g. for stack trace elements corresponding to Kotlin code) .dSYM bundle for Kotlin code is required.

By default, Kotlin/Native compiler produces .dSYM for release (i.e. optimized) binaries on Darwin platforms. This can be disabled with -Xadd-light-debug=disable compiler flag. At the same time, this option is disabled by default for other platforms. To enable it, use the -Xadd-light-debug=enable compiler option.

Groovy

```
kotlin {  
    targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget) {  
        binaries.all {  
            freeCompilerArgs += "-Xadd-light-debug={enable|disable}"  
        }  
    }  
}
```

Kotlin

```
kotlin {  
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {  
        binaries.all {  
            freeCompilerArgs += "-Xadd-light-debug={enable|disable}"  
        }  
    }  
}
```

In projects created from IntelliJ IDEA or AppCode templates these .dSYM bundles are then discovered by Xcode automatically.

Make frameworks static when using rebuild from bitcode

Rebuilding Kotlin-produced framework from bitcode invalidates the original .dSYM. If it is performed locally, make sure the updated .dSYM is used when symbolicating crash reports.

If rebuilding is performed on App Store side, then .dSYM of rebuilt dynamic framework seems discarded and not downloadable from App Store Connect. In this case, it may be required to make the framework static.

Groovy

```
kotlin {  
    targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget) {  
        binaries.withType(org.jetbrains.kotlin.gradle.plugin.mpp.Framework) {  
            isStatic = true  
        }  
    }  
}
```

```

kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.withType<org.jetbrains.kotlin.gradle.plugin.mpp.Framework> {
            isStatic = true
        }
    }
}

```

Decode inlined stack frames

Xcode doesn't seem to properly decode stack trace elements of inlined function calls (these aren't only Kotlin inline functions but also functions that are inlined when optimizing machine code). So some stack trace elements may be missing. If this is the case, consider using lldb to process crash report that is already symbolicated by Xcode, for example:

```
$ lldb -b -o "script import lldb.macosx" -o "crashlog file.crash"
```

This command should output crash report that is additionally processed and includes inlined stack trace elements.

More details can be found in [LLDB documentation](#).

Tips for improving Kotlin/Native compilation times

The Kotlin/Native compiler is constantly receiving updates that improve its performance. With the latest Kotlin/Native compiler and a properly configured build environment, you can significantly improve the compilation times of your projects with Kotlin/Native targets.

Read on for our tips on how to speed up the Kotlin/Native compilation process.

General recommendations

- Use the most recent version of Kotlin. This way you will always have the latest performance improvements.
- Avoid creating huge classes. They take a long time to compile and load during execution.
- Preserve downloaded and cached components between builds. When compiling projects, Kotlin/Native downloads the required components and caches some results of its work to the `$USER_HOME/.konan` directory. The compiler uses this directory for subsequent compilations, making them take less time to complete.

When building in containers (such as Docker) or with continuous integration systems, the compiler may have to create the `~/.konan` directory from scratch for each build. To avoid this step, configure your environment to preserve `~/.konan` between builds. For example, redefine its location using the `KONAN_DATA_DIR` environment variable.

Gradle configuration

The first compilation with Gradle usually takes more time than subsequent ones due to the need to download the dependencies, build caches, and perform additional steps. You should build your project at least twice to get an accurate reading of the actual compilation times.

Here are some recommendations for configuring Gradle for better compilation performance:

- Increase the [Gradle heap size](#). Add `org.gradle.jvmargs=-Xmx3g` to `gradle.properties`. If you use [parallel builds](#), you might need to make the heap even larger or choose the right number of threads with `org.gradle.parallel.threads`.
- Build only the binaries you need. Don't run Gradle tasks that build the whole project, such as `build` or `assemble`, unless you really need to. These tasks build the same code more than once, increasing the compilation times. In typical cases such as running tests from IntelliJ IDEA or starting the app from Xcode, the Kotlin tooling avoids executing unnecessary tasks.

If you have a non-typical case or build configuration, you might need to choose the task yourself.

- `linkDebug*`: To run your code during development, you usually need only one binary, so running the corresponding `linkDebug*` task should be enough. Keep in mind that compiling a release binary (`linkRelease*`) takes more time than compiling a debug one.
- `packForXcode`: Since iOS simulators and devices have different processor architectures, it's a common approach to distribute a Kotlin/Native binary as a universal (fat) framework. During local development, it will be faster to build the `.framework` for only the platform you're using.

To build a platform-specific framework, call the `packForXcode` task generated by the [KMM project wizard](#).

Remember that in this case, you will need to clean the build using `./gradlew clean` after switching between the device and the simulator. See [this issue](#) for details.

- Don't disable the [Gradle daemon](#) without having a good reason to. [Kotlin/Native runs from the Gradle daemon](#) by default. When it's enabled, the same JVM process is used and there is no need to warm it up for each compilation.
- Use the Gradle [build caches](#):
 - Local build cache: Add `org.gradle.caching=true` to your `gradle.properties` or run with `--build-cache` on the command line.
 - Remote build cache in continuous integration environments. Learn how to [configure the remote build cache](#).

- Use the compiler caches. Starting from 1.5.0-M1, linuxX64 and iosArm64 targets have experimental opt-in support for compiler caches. They improve compilation times for debug builds (for linuxX64, this feature is only available on Linux hosts). To enable the compiler caches, add kotlin.native.cacheKind.linuxX64=static or kotlin.native.cacheKind.iosArm64=static to gradle.properties.

iosX64 and macosX64 targets already have the compiler caches enabled by default.

- Enable previously disabled features of Kotlin/Native. There are properties that disable the Gradle daemon and compiler caches – kotlin.native.disableCompilerDaemon=true and kotlin.native.cacheKind=none. If you had issues with these features before and added these lines to your gradle.properties or Gradle arguments, remove them and check whether the build completes successfully. It is possible that these properties were added previously to work around issues that have already been fixed.

Kotlin/Native FAQ

How do I run my program?

Define a top-level function fun main(args: Array<String>) or just fun main() if you are not interested in passed arguments, please ensure it's not in a package. Also compiler switch -entry could be used to make any function taking Array<String> or no arguments and return Unit as an entry point.

What is Kotlin/Native memory management model?

Kotlin/Native provides an automated memory management scheme, similar to what Java or Swift provides. The current implementation includes an automated reference counter with a cycle collector to collect cyclical garbage.

How do I create a shared library?

Use the -produce dynamic compiler switch, or binaries.sharedLib() in Gradle.

```
kotlin {
    iosArm64("myLib") {
        binaries.sharedLib()
    }
}
```

It will produce a platform-specific shared object (.so on Linux, .dylib on macOS, and .dll on Windows targets) and a C language header, allowing the use of all public APIs available in your Kotlin/Native program from C/C++ code. See [this example](#) of using such a shared object to provide a bridge between Python and Kotlin/Native.

How do I create a static library or an object file?

Use the -produce static compiler switch, or binaries.staticLib() in Gradle.

```
kotlin {  
    iosArm64("mylib") {  
        binaries.staticLib()  
    }  
}
```

It will produce a platform-specific static object (.a library format) and a C language header, allowing you to use all the public APIs available in your Kotlin/Native program from C/C++ code.

How do I run Kotlin/Native behind a corporate proxy?

As Kotlin/Native needs to download a platform specific toolchain, you need to specify -Dhttp.proxyHost=xxx - Dhttp.proxyPort=xxx as the compiler's or gradlew arguments, or set it via the JAVA_OPTS environment variable.

How do I specify a custom Objective-C prefix/name for my Kotlin framework?

Use the -module-name compiler option or matching Gradle DSL statement.

Groovy

```
kotlin {  
    iosArm64("myapp") {  
        binaries.framework {  
            freeCompilerArgs += ["-module-name", "TheName"]  
        }  
    }  
}
```

Kotlin

```
kotlin {  
    iosArm64("myapp") {  
        binaries.framework {  
            freeCompilerArgs += listOf("-module-name", "TheName")  
        }  
    }  
}
```

How do I rename the iOS framework?

The default name for an iOS framework is <project name>.framework. To set a custom name, use the `baseName` option. This will also set the module name.

```
kotlin {  
    iosArm64("myapp") {  
        binaries {  
            framework {  
                baseName = "TheName"  
            }  
        }  
    }  
}
```

How do I enable bitcode for my Kotlin framework?

By default gradle plugin adds it on iOS target.

- For debug build it embeds placeholder LLVM IR data as a marker.
- For release build it embeds bitcode as data.

Or commandline arguments: `-Xembed-bitcode` (for release) and `-Xembed-bitcode-marker` (debug)

Setting this in a Gradle DSL:

```
kotlin {  
    iosArm64("myapp") {  
        binaries {  
            framework {  
                // Use "marker" to embed the bitcode marker (for debug builds).  
                // Use "disable" to disable embedding.  
                embedBitcode("bitcode") // for release binaries.  
            }  
        }  
    }  
}
```

These options have nearly the same effect as clang's `-fembed-bitcode`/ `-fembed-bitcode-marker` and swiftc's `-embed-bitcode`/ `-embed-bitcode-marker`.

Why do I see InvalidMutabilityException?

It likely happens, because you are trying to mutate a frozen object. An object can transfer to the frozen state either explicitly, as objects reachable from objects on which the `kotlin.native.concurrent.freeze` is called, or implicitly (i.e. reachable from enum or global singleton object - see the next question).

How do I make a singleton object mutable?

Currently, singleton objects are immutable (i.e. frozen after creation), and it's generally considered good practise to have the global state immutable. If for some reason you need a mutable state inside such an object, use the `@konan.ThreadLocal` annotation on the object. Also the `kotlin.native.concurrent.AtomicReference` class could be used to store different pointers to frozen objects in a frozen object and automatically update them.

How can I compile my project with unreleased versions of Kotlin/Native?

First, please consider trying [preview versions](#).

In case you need an even more recent development version, you can build Kotlin/Native from source code: clone [Kotlin repository](#) and follow [these steps](#).

Kotlin releases

We ship different types of releases:

- Feature releases (1. x) that bring major changes in the language.
- Incremental releases (1. x. y) that are shipped between feature releases and include updates in the tooling, performance improvements, and bug fixes.
- Bug fix releases (1. x. yz) that include bug fixes for incremental releases.

For example, for the feature release 1.3 we had several incremental releases including 1.3.10, 1.3.20, and 1.3.70. For 1.3.70, we had 2 bug fix releases – 1.3.71 and 1.3.72.

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

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

Update to a new release

IntelliJ IDEA and Android Studio suggest updating to a new release once it is out. When you accept the suggestion, it automatically updates the Kotlin plugin to the new version. You can check the Kotlin version in Tools | Kotlin | Configure Plugin Updates.

If you have projects created with earlier Kotlin versions, change the Kotlin version in your projects and update kotlinx libraries if necessary – check the [recommended versions](#).

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

IDE support

The IDE support for the latest version of the language is available for the following versions of IntelliJ IDEA and Android Studio:

- IntelliJ IDEA:
 - Latest stable ([IntelliJ IDEA 2021.1](#) version)
 - Previous stable ([IntelliJ IDEA 2020.3](#) version)
 - [Early access](#) versions
- Android Studio:
 - [Latest released](#) version
 - [Early access](#) versions

Release details

The following table lists details of latest Kotlin releases.

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

Build info	Build highlights	Recommended kotlinx library versions
------------	------------------	--------------------------------------

Build info	Build highlights	Recommended kotlinx library versions
1.5.20 Released: June 24, 2021 Release on GitHub	<p>An incremental release with various improvements such as:</p> <ul style="list-style-type: none"> String concatenation via invokedynamic on JVM by default Improved support for Lombok and support for JSpecify Kotlin/Native: KDoc export to Objective-C headers and faster Array.copyInto() inside one array Gradle: caching of annotation processors' classloaders and support for the --parallel Gradle property Aligned behavior of stdlib functions across platforms <p>Learn more in:</p> <ul style="list-style-type: none"> Release blog post What's new in Kotlin 1.5.20 	<ul style="list-style-type: none"> kotlinx.serialization version: 1.2.1 kotlinx.coroutines version: 1.5.0 kotlinx.atomicfu version: 0.16.1 ktor version: 1.6.0 kotlinx.html version: 0.7.2 kotlinx-nodejs version: 0.0.7 <p>The versions of libraries from kotlin-wrappers (such as kotlin-react) can be found in the corresponding repository.</p>
1.5.10 Released: May 24, 2021 Release on GitHub	<p>A bug fix release for Kotlin 1.5.0.</p> <p>Learn more about Kotlin 1.5.0.</p>	<ul style="list-style-type: none"> kotlinx.serialization version: 1.2.1 kotlinx.coroutines version: 1.5.0 kotlinx.atomicfu version: 0.16.1 ktor version: 1.5.4 kotlinx.html version: 0.7.2 kotlinx-nodejs version: 0.0.7 <p>The versions of libraries from kotlin-wrappers (such as kotlin-react) can be found in the corresponding repository.</p>

Build info	Build highlights	Recommended kotlinx library versions
1.5.0 Released: May 5, 2021	A feature release with new language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs. Learn more in: Release blog post What's new in Kotlin 1.5.0 Compatibility Guide	<ul style="list-style-type: none"> kotlinx.serialization version: 1.2.1 kotlinx.coroutines version: 1.5.0-RC kotlinx.atomicfu version: 0.16.1 ktor version: 1.5.3 kotlinx.html version: 0.7.2 kotlinx-nodejs version: 0.0.7 <p>The versions of libraries from kotlin-wrappers (such as kotlin-react) can be found in the corresponding repository.</p>
1.4.32 Released: March 22, 2021	A bug fix release for Kotlin 1.4.30. Learn more about Kotlin 1.4.30 .	<ul style="list-style-type: none"> kotlinx.serialization version: 1.1.0 kotlinx.coroutines version: 1.4.3 kotlinx.atomicfu version: 0.15.2 ktor version: 1.5.2 kotlinx.html version: 0.7.2 kotlinx-nodejs version: 0.0.7 <p>The versions of libraries from kotlin-wrappers (such as kotlin-react) can be found in the corresponding repository.</p>

Build info	Build highlights	Recommended kotlinx library versions
1.4.31 Released: February 25, 2021	A bug fix release for Kotlin 1.4.30 Learn more about Kotlin 1.4.30 .	<ul style="list-style-type: none"> • kotlinx.serialization version: 1.1.0 • kotlinx.coroutines version: 1.4.2 • kotlinx.atomicfu version: 0.15.1 • ktor version: 1.5.1 • kotlinx.html version: 0.7.2 • kotlinx-nodejs version: 0.0.7 <p>The versions of libraries from kotlin-wrappers (such as kotlin-react) can be found in the corresponding repository.</p>
1.4.30 Released: February 3, 2021	An incremental release with various improvements such as: <ul style="list-style-type: none"> • New JVM backend, now in Beta • Preview of new language features • Improved Kotlin/Native performance • Standard library API improvements <p>Learn more in:</p> <ul style="list-style-type: none"> • Release blog post • What's new in Kotlin 1.4.30 	<ul style="list-style-type: none"> • kotlinx.serialization version: 1.1.0-RC • kotlinx.coroutines version: 1.4.2 • kotlinx.atomicfu version: 0.15.1 • ktor version: 1.5.1 • kotlinx.html version: 0.7.2 • kotlinx-nodejs version: 0.0.7 <p>The versions of libraries from kotlin-wrappers (such as kotlin-react) can be found in the corresponding repository.</p>

Build info	Build highlights	Recommended kotlinx library versions
1.4.21 Released: December 7, 2020	A bug fix release for Kotlin 1.4.20. <u>Release on GitHub</u>	<ul style="list-style-type: none"> • kotlinx.serialization version: 1.0.1 • kotlinx.coroutines version: 1.4.1 • kotlinx.atomicfu version: 0.14.4 • ktor version: 1.4.1 • kotlinx.html version: 0.7.2 • kotlinx-nodejs version: 0.0.6 <p>The versions of libraries from kotlin-wrappers (such as kotlin-react) can be found in the corresponding repository.</p>
1.4.20 Released: November 23, 2020	An incremental release with various improvements such as: <ul style="list-style-type: none"> • Supporting new JVM features, like string concatenation via invokedynamic • Improved performance and exception handling for KMM projects • Extensions for JDK Path: Path("dir") / "file.txt" <u>Release on GitHub</u> Learn more in: <ul style="list-style-type: none"> • Release blog post • What's new in Kotlin 1.4.20 	<ul style="list-style-type: none"> • kotlinx.serialization version: 1.0.1 • kotlinx.coroutines version: 1.4.1 • kotlinx.atomicfu version: 0.14.4 • ktor version: 1.4.1 • kotlinx.html version: 0.7.2 • kotlinx-nodejs version: 0.0.6 <p>The versions of libraries from kotlin-wrappers (such as kotlin-react) can be found in the corresponding repository.</p>

Build info	Build highlights	Recommended kotlinx library versions
1.4.10 Released: September 7, 2020	A bug fix release for Kotlin 1.4.0. Learn more about Kotlin 1.4.0 .	<ul style="list-style-type: none"> • kotlinx.serialization version: 1.0.0-RC • kotlinx.coroutines version: 1.3.9 • kotlinx.atomicfu version: 0.14.4 • ktor version: 1.4.0 • kotlinx.html version: 0.7.2 • kotlinx-nodejs version: 0.0.6 <p>The versions of libraries from kotlin-wrappers (such as kotlin-react) can be found in the corresponding repository.</p>
1.4.0 Released: August 17, 2020	A feature release with many features and improvements that mostly focus on quality and performance. Learn more in: <u>Release on GitHub</u> <ul style="list-style-type: none"> • Release blog post • What's new in Kotlin 1.4.0 • Compatibility Guide • Migrating to Kotlin 1.4.0 	<ul style="list-style-type: none"> • kotlinx.serialization version: 1.0.0-RC • kotlinx.coroutines version: 1.3.9 • kotlinx.atomicfu version: 0.14.4 • ktor version: 1.4.0 • kotlinx.html version: 0.7.2 • kotlinx-nodejs version: 0.0.6 <p>The versions of libraries from kotlin-wrappers (such as kotlin-react) can be found in the corresponding repository.</p>

Build info	Build highlights	Recommended kotlinx library versions
1.3.72	A bug fix release for Kotlin 1.3.70.	<ul style="list-style-type: none">• kotlinx.serialization version: 0.20.0
Released: April 15, 2020	Learn more about Kotlin 1.3.70 .	<ul style="list-style-type: none">• kotlinx.coroutines version: 1.3.8• kotlinx.atomicfu version: 0.14.2• ktor version: 1.3.2• kotlinx.html version: 0.7.1• kotlinx-nodejs version: 0.0.3
Release on GitHub		The versions of libraries from kotlin-wrappers (such as kotlin-react) can be found in the corresponding repository .

On the JVM, you usually can use library versions other than the recommended ones.

Kotlin plugin releases

The [IntelliJ Kotlin plugin](#) and [IntelliJ IDEA](#) are on the same release cycle. To speed up the testing and delivery of new features, the plugin and the platform have been moved to the same codebase and ship simultaneously. Kotlin releases happen independently according to the [new release cadence](#).

Kotlin and the Kotlin plugin have distinct sets of features:

- Kotlin releases contain language, compiler, and standard library features.
- Kotlin plugin releases introduce only IDE related features. For example, code formatting and debugging tools.

This also affects the versioning of the Kotlin plugin. Releases now have the same version as the simultaneous IntelliJ IDEA release. You can learn more about new release cadence in this [blog post](#).

Update to a new release

IntelliJ IDEA and Android Studio suggest updating to a new release once it is out. When you accept the suggestion, it automatically updates the Kotlin plugin to the new version. You can check the Kotlin plugin version in Tools | Kotlin | Configure Kotlin Plugin Updates.

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

Release details

The following table lists the details of the latest Kotlin plugin releases:

Release info	Release highlights
2021.1	<ul style="list-style-type: none">• Performance improvements
Released: April 7, 2021	<ul style="list-style-type: none">• Evaluation of custom getters during debugging• Improved Change Signature refactoring• Code completion for type arguments• UML diagrams for Kotlin classes
	<p>Learn more in:</p> <ul style="list-style-type: none">• What's new in Kotlin plugin 2021.1• What's New in IntelliJ IDEA 2021.1
2020.3	<ul style="list-style-type: none">• New types of inline refactorings
Released: December 1, 2020	<ul style="list-style-type: none">• Structural search and replace• EditorConfig support• Project templates for Jetpack Compose for Desktop
	<p>Learn more in:</p> <ul style="list-style-type: none">• What's new in Kotlin plugin 2020.3• IntelliJ IDEA 2020.3 release blog post

Kotlin roadmap

Last modified on May 2021

Next update November 2021

Welcome to the Kotlin roadmap! Get a sneak peek into the priorities of the Kotlin Team.

Key priorities

The goal of this roadmap is to give you a big picture. Here's a list of our key priorities – the areas we are investing the most effort into:

- Fast turnaround: making the change-test-debug cycle really fast.
- New compiler: a rewrite of the Kotlin compiler optimized for speed, parallelism, and unification. Later we will also work on pluggability.
- Fast and smooth IDE: improving the stability and performance of the Kotlin IDE.
- Kotlin for JVM server-side development: expanding support for server-side use cases across the Kotlin ecosystem.
- Kotlin Multiplatform Mobile: improving the user experience and feature set for sharing code on mobile platforms.

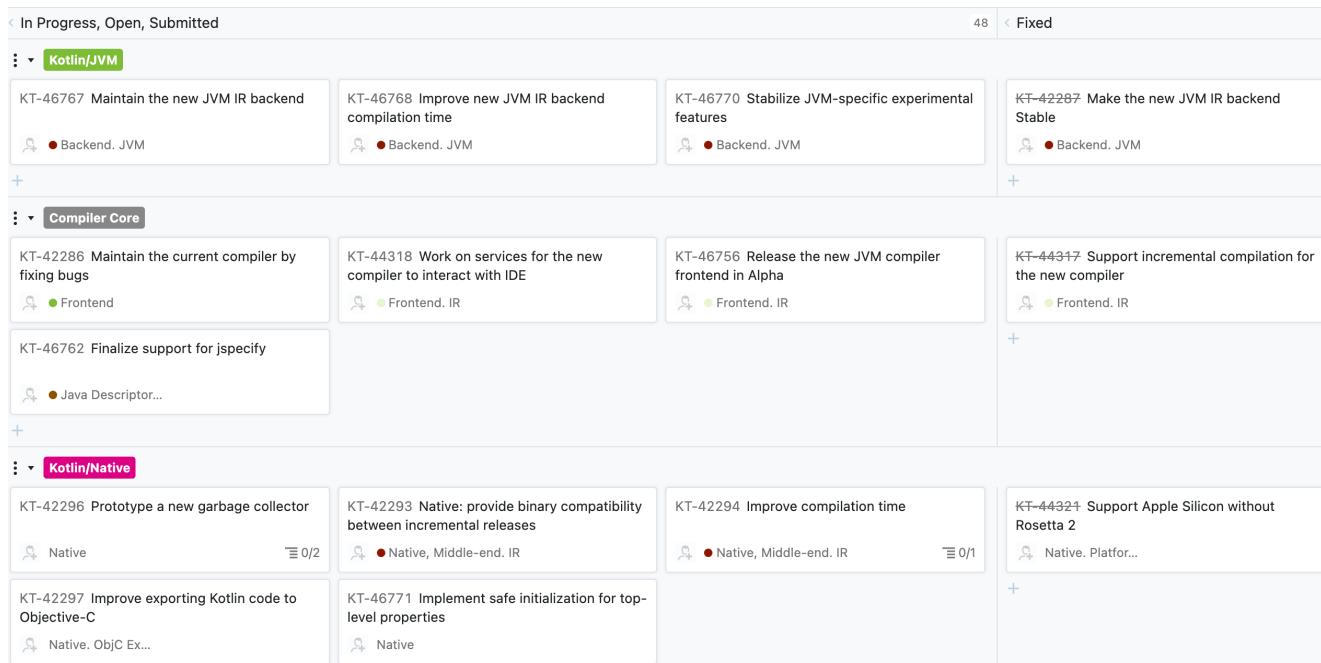
Kotlin roadmap by subsystem

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!



The screenshot shows a YouTrack board with three main sections: **Kotlin/JVM**, **Compiler Core**, and **Kotlin/Native**.

- Kotlin/JVM**: Contains issues like KT-46767, KT-46768, KT-46770, and KT-42287. Labels include Backend. JVM.
- Compiler Core**: Contains issues like KT-42286, KT-44318, KT-46756, and KT-44317. Labels include Frontend, IR.
- Kotlin/Native**: Contains issues like KT-42296, KT-42293, KT-42294, and KT-44321. Labels include Native, Middle-end. IR, and Native. Platfor...

Roadmap board in YouTrack

Roadmap details

Subsystem	In focus now	Postponed for later
Language	<ul style="list-style-type: none"> ■■■ Support sealed (exhaustive) whens ■■■ Release OptIn annotations ■■■ Support programmatic creation of annotation class instances ■■■ Stabilize typeOf ■■■ Stabilize builder inference ■■■ Allow repeating annotations with runtime retention when compiling under Java 8 ■■■ Support annotations on class type parameters ■■■ Improve type inference in corner cases for popular Java APIs ■ Prototype multiple receivers 	<ul style="list-style-type: none"> Support JVM sealed classes
Compiler core	<ul style="list-style-type: none"> ■■■ Release the new compiler frontend in Alpha for JVM target ■■■ Finalize support for jspecify Maintain the current compiler (bug-fixing only) Work on services for the new compiler to interact with IDE 	<ul style="list-style-type: none"> Stable Compiler Plugin API Scripting improvements
Kotlin/JVM	<ul style="list-style-type: none"> ■■■ Maintain the new JVM IR backend ■■■ Improve new JVM IR backend compilation time ■■■ Stabilize JVM-specific experimental features 	

Subsystem	In focus now	Postponed for later
Kotlin/JS	<ul style="list-style-type: none"> • Make the new JS IR backend Stable • JS IR BE: Add an ability to generate separate JS files for each module • Maintain the old JS backend by fixing critical bugs 	<ul style="list-style-type: none"> • ES6 support • Improve Dukat support
Kotlin/Wasm	<ul style="list-style-type: none"> • Implement an experimental version of Kotlin/Wasm compiler backend 	Note: Wasm support in Kotlin/Native (through LLVM) will be deprecated and removed
Kotlin/Native	<ul style="list-style-type: none"> • Implement safe initialization for top-level properties • Prototype a new garbage collector • Improve compilation time • Improve exporting Kotlin code to Objective-C • Provide binary compatibility between incremental releases 	<ul style="list-style-type: none"> • Support Mac Catalyst • Direct interoperability with Swift • Interoperability with C++ • Support Alpine Linux
Kotlin Multiplatform	<ul style="list-style-type: none"> • Support the Apple Silicon target in the Kotlin Multiplatform tooling • Improve UX of using Native libraries in Kotlin • Improve dependency management for iOS 	<ul style="list-style-type: none"> • Improve Kotlin/Native debugging experience • Improve Gradle and Compiler error messages • Sharing code between JVM and Android
IDE	<ul style="list-style-type: none"> • Prototype IDE plugin with the new compiler frontend • Improve IDE performance • Improve debugging experience • Improve the New Project wizard • Move the Kotlin plugin to the IntelliJ platform development infrastructure 	<ul style="list-style-type: none"> • Advanced tooling that users have in Java but is missing in Kotlin • Quality of less frequently used features, except blocking problems

Subsystem	In focus now	Postponed for later
Build tools	<ul style="list-style-type: none"> • Improve the performance of Gradle incremental compilation • Decrease time for opening Gradle projects • [A] Improve the quality of Gradle import • [A] Improve user experience with the Kotlin Gradle plugin 	<ul style="list-style-type: none"> • Improvements in Kotlin Maven support
Libraries	<ul style="list-style-type: none"> • [A] Improve kotlinx-serialization (release v1.3.0) • [A] Improve kotlinx-coroutines (release v1.6.0) • [A] Stabilize Duration API in the standard library • [A] Get rid of !! for readLine() in the standard library • [A] Stabilize and document atomicfu • Improve kotlinx-datetime library • Support java.nio.Path extension in the standard library 	<ul style="list-style-type: none"> • Improve usability of multi-threaded coroutines library for Kotlin/Native • kotlinx-cli • binary-compatibility-validator • kotlinx-io • Any new multiplatform libraries
Website	<ul style="list-style-type: none"> • Make the Kotlin website mobile friendly • [A] Make the UI and navigation consistent • [A] Update community graphic assets to the new Kotlin visual style • Revamp Kotlin documentation 	

- 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.
- It lists some things that are postponed and will NOT get the team's attention in the nearest future.
- We will adjust our priorities as we go and update the roadmap approximately every six months.

What's changed since January 2021

Completed items

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

- Language: [Support JVM records](#)
- Language: [Release inline classes as Stable, secure Valhalla compatibility](#)
- Language: [Sealed interfaces and more sealed classes freedom](#)
- Compiler core: [Support incremental compilation for the new compiler](#)
- Kotlin/JVM: [Make the new JVM IR backend Stable](#)
- Kotlin/JS: [JS IR BE: Prototype lazy initialization for top-level properties like in JVM](#)
- Kotlin/Wasm: [Prototype a compiler for Wasm GC proposal](#)
- Kotlin/Native: [Support Apple Silicon without Rosetta 2](#)
- Multiplatform: [Introduce a complex KMM application sample](#)
- Multiplatform: [Improve frontend and IDE import stability for Multiplatform projects](#)
- IDE: [Improve cross-language support in the Inline Method and Change Signature refactorings](#)
- Build tools: [Support the Gradle configuration cache](#)
- Libraries: [Make multiplatform kotlin.text API locale-agnostic by default](#)
- Libraries: [Implement unambiguous API for Char conversion](#)
- Libraries: [Implement multiplatform API for characters](#)
- Libraries: [Improve kotlinx-serialization \(release v1.1\)](#)
- Libraries: [Improve kotlinx-serialization \(release v1.2\)](#)
- Libraries: [Improve kotlinx-coroutines \(release v1.5\)](#)
- Website: [Design a new Kotlin visual style](#)
- Website: [Close try.kotlinlang.org](#)

Postponed items

We've decided to postpone the following items from the previous roadmap:

- [Support JVM sealed classes](#)

- [Improve Kotlin/Native debugging experience](#)

Other postponed items remain in this state from earlier roadmap versions.

New items

We've added the following items to the roadmap:

- **A A** Language: [Support sealed \(exhaustive\) whens](#)
- **A A** Language: [Release OptIn annotations](#)
- **A A** Language: [Support programmatic creation of annotation class instances](#)
- **A A** Language: [Stabilize typeOf](#)
- **A A** Language: [Stabilize builder inference](#)
- **A A** Language: [Allow repeating annotations with runtime retention when compiling under Java 8](#)
- **A A** Language: [Support annotations on class type parameters](#)
- **A A** Language: [Improve type inference in corner cases for popular Java APIs](#)
- **A A** Compiler core: [Release the new compiler frontend in Alpha for JVM target](#)
- **A A** Compiler core: [Finalize support for jspecify](#)
- **A A** Kotlin/JVM: [Maintain the new JVM IR backend](#)
- **A A** Kotlin/JVM: [Improve new JVM IR backend compilation time](#)
- **A A** Kotlin/JVM: [Stabilize JVM-specific experimental features](#)
- **A A** Kotlin/Wasm: [Implement an experimental version of Kotlin/Wasm compiler backend](#)
- **A A** Kotlin/Native: [Implement safe initialization for top-level properties](#)
- **A A** Multiplatform: [Support the Apple Silicon target in the Kotlin Multiplatform tooling](#)
- **A A** IDE: [Improve the New Project wizard](#)
- **A A** Build tools: [Improve the quality of Gradle import](#)
- **A A** Build tools: [Improve user experience with the Kotlin Gradle plugin](#)
- **A A** Libraries: [Improve kotlinx-serialization \(release v1.3.0\)](#)
- **A A** Libraries: [Improve kotlinx-coroutines \(release v1.6.0\)](#)
- **A A** Libraries: [Stabilize Duration API in the standard library](#)
- **A A** Libraries: [Get rid of !! for readLine\(\) in the standard library](#)

- Libraries: [Stabilize and document atomicfu](#)
- Website: [Make the UI and navigation consistent](#)
- Website: [Update community graphic assets to the new Kotlin visual style](#)

Items in progress

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

Collections overview

The Kotlin Standard Library provides a comprehensive set of tools for managing collections – groups of a variable number of items (possibly zero) that share significance to the problem being solved and are operated upon commonly.

Collections are a common concept for most programming languages, so if you're familiar with, for example, Java or Python collections, you can skip this introduction and proceed to the detailed sections.

A collection usually contains a number of objects (this number may also be zero) of the same type. Objects in a collection are called elements or items. For example, all the students in a department form a collection that can be used to calculate their average age.

The following collection types are relevant for Kotlin:

- List is an ordered collection with access to elements by indices – integer numbers that reflect their position.
Elements can occur more than once in a list. An example of a list is a sentence: it's a group of words, their order is important, and they can repeat.
- Set is a collection of unique elements. It reflects the mathematical abstraction of set: a group of objects without repetitions. Generally, the order of set elements has no significance. For example, an alphabet is a set of letters.
- Map (or dictionary) is a set of key-value pairs. Keys are unique, and each of them maps to exactly one value. The values can be duplicates. Maps are useful for storing logical connections between objects, for example, an employee's ID and their position.

Kotlin lets you manipulate collections independently of the exact type of objects stored in them. In other words, you add a `String` to a list of `String`s the same way as you would do with `Int`s or a user-defined class. So, the Kotlin Standard Library offers generic interfaces, classes, and functions for creating, populating, and managing collections of any type.

The collection interfaces and related functions are located in the `kotlin.collections` package. Let's get an overview of its contents.

Collection types

The Kotlin Standard Library provides implementations for basic collection types: sets, lists, and maps. A pair of interfaces represent each collection type:

- A read-only interface that provides operations for accessing collection elements.
- A mutable interface that extends the corresponding read-only interface with write operations: adding, removing, and updating its elements.

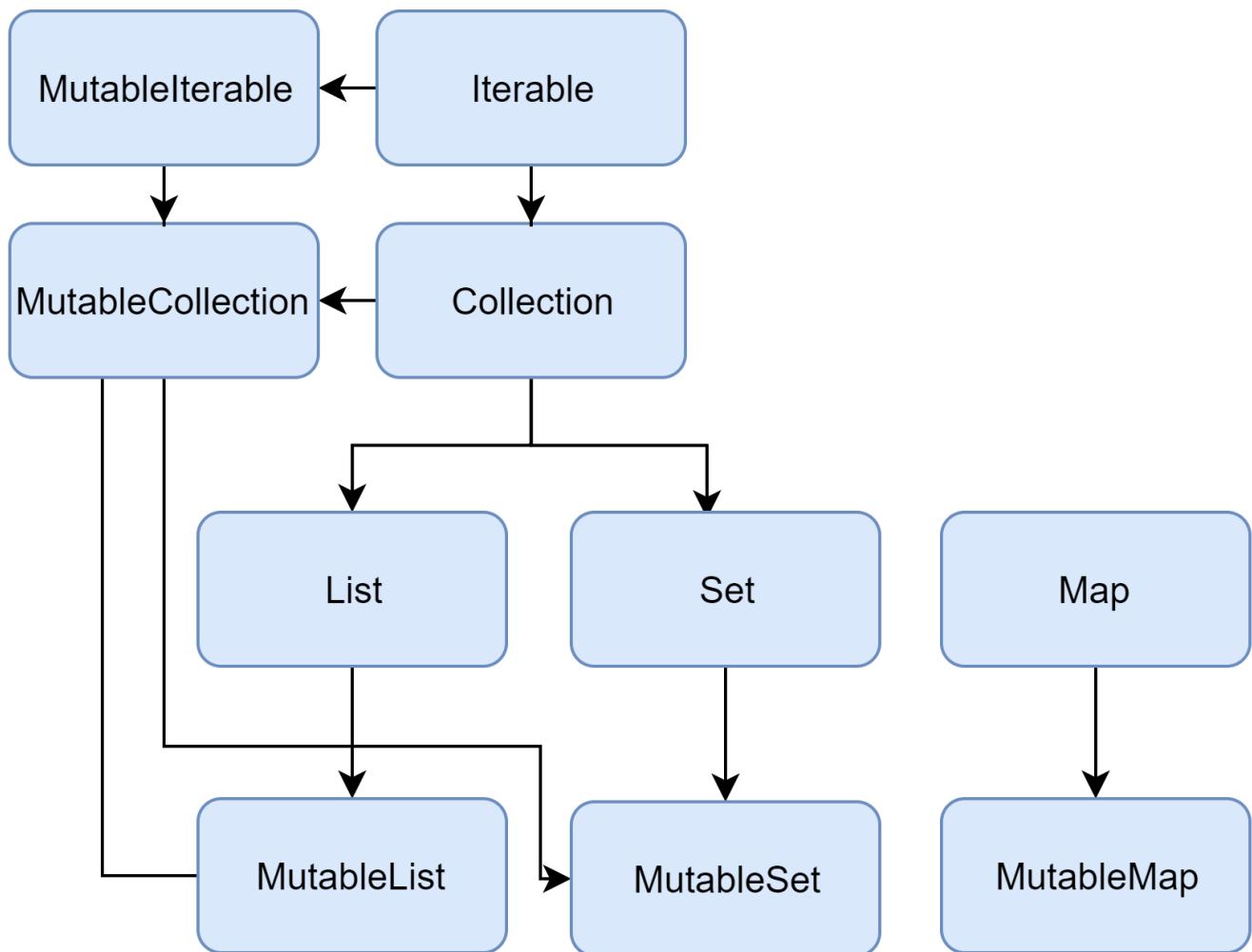
Note that altering a mutable collection doesn't require it to be a `var`: write operations modify the same mutable collection object, so the reference doesn't change. Although, if you try to reassign a `val` collection, you'll get a compilation error.

```
fun main() {  
    //sampleStart  
    val numbers = mutableListOf("one", "two", "three", "four")  
    numbers.add("five")    // this is OK  
    //numbers = mutableListOf("six", "seven")      // compilation error  
    //sampleEnd  
}
```

The read-only collection types are covariant. This means that, if a `Rectangle` class inherits from `Shape`, you can use a `List<Rectangle>` anywhere the `List<Shape>` is required. In other words, the collection types have the same subtyping relationship as the element types. Maps are covariant on the value type, but not on the key type.

In turn, mutable collections aren't covariant; otherwise, this would lead to runtime failures. If `MutableList<Rectangle>` was a subtype of `MutableList<Shape>`, you could insert other `Shape` inheritors (for example, `Circle`) into it, thus violating its `Rectangle` type argument.

Below is a diagram of the Kotlin collection interfaces:



Collection interfaces hierarchy

Let's walk through the interfaces and their implementations.

Collection

`Collection<T>` is the root of the collection hierarchy. This interface represents the common behavior of a read-only collection: retrieving size, checking item membership, and so on. `Collection` inherits from the `Iterable<T>` interface that defines the operations for iterating elements. You can use `Collection` as a parameter of a function that applies to different collection types. For more specific cases, use the `Collection`'s inheritors: [List](#) and [Set](#).

```

fun printAll(strings: Collection<String>) {
    for(s in strings) print("$s ")
    println()
}

fun main() {
    val stringList = listOf("one", "two", "one")
    printAll(stringList)

    val stringSet = setOf("one", "two", "three")
  
```

```
    printAll(stringSet)
}
```

MutableCollection<T> is a Collection with write operations, such as add and remove.

```
fun List<String>.getShortWordsTo(shortWords: MutableList<String>, maxLength: Int) {
    this.filterTo(shortWords) { it.length <= maxLength }
    // throwing away the articles
    val articles = setOf("a", "A", "an", "An", "the", "The")
    shortWords -= articles
}

fun main() {
    val words = "A long time ago in a galaxy far far away".split(" ")
    val shortWords = mutableListOf<String>()
    words.getShortWordsTo(shortWords, 3)
    println(shortWords)
}
```

List

List<T> stores elements in a specified order and provides indexed access to them. Indices start from zero – the index of the first element – and go to lastIndex which is the (list.size - 1).

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println("Number of elements: ${numbers.size}")
    println("Third element: ${numbers.get(2)}")
    println("Fourth element: ${numbers[3]}")
    println("Index of element \"two\" ${numbers.indexOf("two")}")
//sampleEnd
}
```

List elements (including nulls) can duplicate: a list can contain any number of equal objects or occurrences of a single object. Two lists are considered equal if they have the same sizes and structurally equal elements at the same positions.

```
data class Person(var name: String, var age: Int)

fun main() {
//sampleStart
    val bob = Person("Bob", 31)
    val people = listOf(Person("Adam", 20), bob, bob)
    val people2 = listOf(Person("Adam", 20), Person("Bob", 31), bob)
    println(people == people2)
    bob.age = 32
    println(people == people2)
//sampleEnd
}
```

MutableList<T> is a List with list-specific write operations, for example, to add or remove an element at a specific position.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.add(5)
    numbers.removeAt(1)
    numbers[0] = 0
    numbers.shuffle()
    println(numbers)
//sampleEnd
}
```

As you see, in some aspects lists are very similar to arrays. However, there is one important difference: an array's size is defined upon initialization and is never changed; in turn, a list doesn't have a predefined size; a list's size can be changed as a result of write operations: adding, updating, or removing elements.

In Kotlin, the default implementation of List is ArrayList which you can think of as a resizable array.

Set

Set<T> stores unique elements; their order is generally undefined. null elements are unique as well: a Set can contain only one null. Two sets are equal if they have the same size, and for each element of a set there is an equal element in the other set.

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3, 4)
    println("Number of elements: ${numbers.size}")
    if (numbers.contains(1)) println("1 is in the set")

    val numbersBackwards = setOf(4, 3, 2, 1)
    println("The sets are equal: ${numbers == numbersBackwards}")
//sampleEnd
}
```

MutableSet is a Set with write operations from MutableCollection.

The default implementation of Set – LinkedHashSet – preserves the order of elements insertion. Hence, the functions that rely on the order, such as first() or last(), return predictable results on such sets.

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3, 4) // LinkedHashSet is the default implementation
    val numbersBackwards = setOf(4, 3, 2, 1)

    println(numbers.first() == numbersBackwards.first())
    println(numbers.first() == numbersBackwards.last())
//sampleEnd
}
```

An alternative implementation – HashSet – says nothing about the elements order, so calling such functions on it returns unpredictable results. However, HashSet requires less memory to store the same number of elements.

Map

Map<K, V> is not an inheritor of the Collection interface; however, it's a Kotlin collection type as well. A Map stores key-value pairs (or entries); keys are unique, but different keys can be paired with equal values. The Map interface provides specific functions, such as access to value by key, searching keys and values, and so on.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)

    println("All keys: ${numbersMap.keys}")
    println("All values: ${numbersMap.values}")
    if ("key2" in numbersMap) println("Value by key \"key2\": ${numbersMap["key2"]}")
    if (1 in numbersMap.values) println("The value 1 is in the map")
    if (numbersMap.containsValue(1)) println("The value 1 is in the map") // same as previous
//sampleEnd
}
```

Two maps containing the equal pairs are equal regardless of the pair order.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
    val anotherMap = mapOf("key2" to 2, "key1" to 1, "key4" to 1, "key3" to 3)

    println("The maps are equal: ${numbersMap == anotherMap}")
//sampleEnd
}
```

MutableMap is a Map with map write operations, for example, you can add a new key-value pair or update the value associated with the given key.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap.put("three", 3)
    numbersMap["one"] = 11

    println(numbersMap)
//sampleEnd
}
```

The default implementation of Map – LinkedHashMap – preserves the order of elements insertion when iterating the map. In turn, an alternative implementation – HashMap – says nothing about the elements order.

Constructing collections

Construct from elements

The most common way to create a collection is with the standard library functions `listOf<T>()`, `setOf<T>()`, `mutableListOf<T>()`, `mutableSetOf<T>()`. If you provide a comma-separated list of collection elements as arguments, the compiler detects the element type automatically. When creating empty collections, specify the type explicitly.

```
val numbersSet = setOf("one", "two", "three", "four")
val emptySet = mutableSetOf<String>()
```

The same is available for maps with the functions `mapOf()` and `mutableMapOf()`. The map's keys and values are passed as Pair objects (usually created with `to` infix function).

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
```

Note that the `to` notation creates a short-living Pair object, so it's recommended that you use it only if performance isn't critical. To avoid excessive memory usage, use alternative ways. For example, you can create a mutable map and populate it using the write operations. The `apply()` function can help to keep the initialization fluent here.

```
val numbersMap = mutableMapOf<String, String>().apply { this["one"] = "1"; this["two"] = "2" }
```

Empty collections

There are also functions for creating collections without any elements: `emptyList()`, `emptySet()`, and `emptyMap()`. When creating empty collections, you should specify the type of elements that the collection will hold.

```
val empty = emptyList<String>()
```

Initializer functions for lists

For lists, there is a constructor that takes the list size and the initializer function that defines the element value based on its index.

```
fun main() {
//sampleStart
    val doubled = List(3, { it * 2 }) // or MutableList if you want to change its content
later
    println(doubled)
```

```
//sampleEnd  
}
```

Concrete type constructors

To create a concrete type collection, such as an `ArrayList` or `LinkedList`, you can use the available constructors for these types. Similar constructors are available for implementations of `Set` and `Map`.

```
val linkedList = LinkedList<String>(listOf("one", "two", "three"))  
val presizedSet = HashSet<Int>(32)
```

Copy

To create a collection with the same elements as an existing collection, you can use copying operations. Collection copying operations from the standard library create shallow copy collections with references to the same elements. Thus, a change made to a collection element reflects in all its copies.

Collection copying functions, such as `toList()`, `toMutableList()`, `toSet()` and others, create a snapshot of a collection at a specific moment. Their result is a new collection of the same elements. If you add or remove elements from the original collection, this won't affect the copies. Copies may be changed independently of the source as well.

```
fun main() {  
//sampleStart  
    val sourceList = mutableListOf(1, 2, 3)  
    val copyList = sourceList.toMutableList()  
    val readOnlyCopyList = sourceList.toList()  
    sourceList.add(4)  
    println("Copy size: ${copyList.size}")  
  
    //readOnlyCopyList.add(4)           // compilation error  
    println("Read-only copy size: ${readOnlyCopyList.size}")  
//sampleEnd  
}
```

These functions can also be used for converting collections to other types, for example, build a set from a list or vice versa.

```
fun main() {  
//sampleStart  
    val sourceList = mutableListOf(1, 2, 3)  
    val copySet = sourceList.toMutableSet()  
    copySet.add(3)  
    copySet.add(4)  
    println(copySet)  
//sampleEnd
```

```
}
```

Alternatively, you can create new references to the same collection instance. New references are created when you initialize a collection variable with an existing collection. So, when the collection instance is altered through a reference, the changes are reflected in all its references.

```
fun main() {
//sampleStart
    val sourceList = mutableListOf(1, 2, 3)
    val referenceList = sourceList
    referenceList.add(4)
    println("Source size: ${sourceList.size}")
//sampleEnd
}
```

Collection initialization can be used for restricting mutability. For example, if you create a List reference to a MutableList, the compiler will produce errors if you try to modify the collection through this reference.

```
fun main() {
//sampleStart
    val sourceList = mutableListOf(1, 2, 3)
    val referenceList: List<Int> = sourceList
    //referenceList.add(4)           //compilation error
    sourceList.add(4)
    println(referenceList) // shows the current state of sourceList
//sampleEnd
}
```

Invoke functions on other collections

Collections can be created in result of various operations on other collections. For example, [filtering](#) a list creates a new list of elements that match the filter:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val longerThan3 = numbers.filter { it.length > 3 }
    println(longerThan3)
//sampleEnd
}
```

[Mapping](#) produces a list of a transformation results:

```

fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3)
    println(numbers.map { it * 3 })
    println(numbers.mapIndexed { idx, value -> value * idx })
//sampleEnd
}

```

Association produces maps:

```

fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
//sampleEnd
}

```

For more information about operations on collections in Kotlin, see [Collection operations overview](#).

Iterators

For traversing collection elements, the Kotlin standard library supports the commonly used mechanism of iterators – objects that provide access to the elements sequentially without exposing the underlying structure of the collection. Iterators are useful when you need to process all the elements of a collection one-by-one, for example, print values or make similar updates to them.

Iterators can be obtained for inheritors of the `Iterable<T>` interface, including Set and List, by calling the `iterator()` function.

Once you obtain an iterator, it points to the first element of a collection; calling the `next()` function returns this element and moves the iterator position to the following element if it exists.

Once the iterator passes through the last element, it can no longer be used for retrieving elements; neither can it be reset to any previous position. To iterate through the collection again, create a new iterator.

```

fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val numbersIterator = numbers.iterator()
    while (numbersIterator.hasNext()) {
        println(numbersIterator.next())
    }
//sampleEnd
}

```

Another way to go through an Iterable collection is the well-known for loop. When using for on a collection, you obtain

the iterator implicitly. So, the following code is equivalent to the example above:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    for (item in numbers) {
        println(item)
    }
//sampleEnd
}
```

Finally, there is a useful `forEach()` function that lets you automatically iterate a collection and execute the given code for each element. So, the same example would look like this:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    numbers.forEach {
        println(it)
    }
//sampleEnd
}
```

List iterators

For lists, there is a special iterator implementation: [ListIterator](#). It supports iterating lists in both directions: forwards and backwards.

Backward iteration is implemented by the functions `hasPrevious()` and `previous()`. Additionally, the `ListIterator` provides information about the element indices with the functions `nextIndex()` and `previousIndex()`.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val listIterator = numbers.listIterator()
    while (listIterator.hasNext()) listIterator.next()
    println("Iterating backwards:")
    while (listIterator.hasPrevious()) {
        print("Index: ${listIterator.previousIndex()}")
        println(", value: ${listIterator.previous()}")
    }
//sampleEnd
}
```

Having the ability to iterate in both directions, means the `ListIterator` can still be used after it reaches the last element.

Mutable iterators

For iterating mutable collections, there is [MutableIterator](#) that extends [Iterator](#) with the element removal function [remove\(\)](#). So, you can remove elements from a collection while iterating it.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    val mutableIterator = numbers.iterator()

    mutableIterator.next()
    mutableIterator.remove()
    println("After removal: $numbers")
//sampleEnd
}
```

In addition to removing elements, the [MutableListIterator](#) can also insert and replace elements while iterating the list.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "four", "four")
    val mutableListIterator = numbers.listIterator()

    mutableListIterator.next()
    mutableListIterator.add("two")
    mutableListIterator.next()
    mutableListIterator.set("three")
    println(numbers)
//sampleEnd
}
```

Ranges and progressions

Kotlin lets you easily create ranges of values using the [rangeTo\(\)](#) function from the `kotlin.ranges` package and its operator form ... Usually, `rangeTo()` is complemented by `in` or `!in` functions.

```
if (i in 1..4) { // equivalent of 1 <= i && i <= 4
    print(i)
}
```

Integral type ranges ([IntRange](#), [LongRange](#), [CharRange](#)) have an extra feature: they can be iterated over. These ranges are also [progressions](#) of the corresponding integral types.

Such ranges are generally used for iteration in for loops.

```
fun main() {  
    //sampleStart  
    for (i in 1..4) print(i)  
    //sampleEnd  
}
```

To iterate numbers in reverse order, use the [downTo](#) function instead of ...

```
fun main() {  
    //sampleStart  
    for (i in 4 downTo 1) print(i)  
    //sampleEnd  
}
```

It is also possible to iterate over numbers with an arbitrary step (not necessarily 1). This is done via the [step](#) function.

```
fun main() {  
    //sampleStart  
    for (i in 1..8 step 2) print(i)  
    println()  
    for (i in 8 downTo 1 step 2) print(i)  
    //sampleEnd  
}
```

To iterate a number range which does not include its end element, use the [until](#) function:

```
fun main() {  
    //sampleStart  
    for (i in 1 until 10) {           // i in [1, 10), 10 is excluded  
        print(i)  
    }  
    //sampleEnd  
}
```

Range

A range defines a closed interval in the mathematical sense: it is defined by its two endpoint values which are both included in the range. Ranges are defined for comparable types: having an order, you can define whether an arbitrary instance is in the range between two given instances.

The main operation on ranges is `contains`, which is usually used in the form of `in` and `!in` operators.

To create a range for your class, call the `rangeTo()` function on the range start value and provide the end value as an argument. `rangeTo()` is often called in its operator form ...

```

class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int {
        if (this.major != other.major) {
            return this.major - other.major
        }
        return this.minor - other.minor
    }
}

fun main() {
//sampleStart
    val versionRange = Version(1, 11)..Version(1, 30)
    println(Version(0, 9) in versionRange)
    println(Version(1, 20) in versionRange)
//sampleEnd
}

```

Progression

As shown in the examples above, the ranges of integral types, such as Int, Long, and Char, can be treated as [arithmetic progressions](#) of them. In Kotlin, these progressions are defined by special types: [IntProgression](#), [LongProgression](#), and [CharProgression](#).

Progressions have three essential properties: the first element, the last element, and a non-zero step. The first element is first, subsequent elements are the previous element plus a step. Iteration over a progression with a positive step is equivalent to an indexed for loop in Java/JavaScript.

```

for (int i = first; i <= last; i += step) {
    // ...
}

```

When you create a progression implicitly by iterating a range, this progression's first and last elements are the range's endpoints, and the step is 1.

```

fun main() {
//sampleStart
    for (i in 1..10) print(i)
//sampleEnd
}

```

To define a custom progression step, use the `step` function on a range.

```

fun main() {
//sampleStart
    for (i in 1..8 step 2) print(i)
}

```

```
//sampleEnd  
}
```

The last element of the progression is calculated this way:

- For a positive step: the maximum value not greater than the end value such that $(\text{last} - \text{first}) \% \text{step} == 0$.
- For a negative step: the minimum value not less than the end value such that $(\text{last} - \text{first}) \% \text{step} == 0$.

Thus, the last element is not always the same as the specified end value.

```
fun main() {  
//sampleStart  
    for (i in 1..9 step 3) print(i) // the last element is 7  
//sampleEnd  
}
```

To create a progression for iterating in reverse order, use `downTo` instead of `..` when defining the range for it.

```
fun main() {  
//sampleStart  
    for (i in 4 downTo 1) print(i)  
//sampleEnd  
}
```

Progressions implement `Iterable<N>`, where `N` is `Int`, `Long`, or `Char` respectively, so you can use them in various collection functions like `map`, `filter`, and other.

```
fun main() {  
//sampleStart  
    println((1..10).filter { it % 2 == 0 })  
//sampleEnd  
}
```

Sequences

Along with collections, the Kotlin standard library contains another container type – sequences (`Sequence<T>`). Sequences offer the same functions as `Iterable` but implement another approach to multi-step collection processing.

When the processing of an `Iterable` includes multiple steps, they are executed eagerly: each processing step completes and returns its result – an intermediate collection. The following step executes on this collection. In turn, multi-step processing of sequences is executed lazily when possible: actual computing happens only when the result of the whole processing chain is requested.

The order of operations execution is different as well: Sequence performs all the processing steps one-by-one for every single element. In turn, Iterable completes each step for the whole collection and then proceeds to the next step.

So, the sequences let you avoid building results of intermediate steps, therefore improving the performance of the whole collection processing chain. However, the lazy nature of sequences adds some overhead which may be significant when processing smaller collections or doing simpler computations. Hence, you should consider both Sequence and Iterable and decide which one is better for your case.

Construct

From elements

To create a sequence, call the `sequenceOf()` function listing the elements as its arguments.

```
val numbersSequence = sequenceOf("four", "three", "two", "one")
```

From an Iterable

If you already have an Iterable object (such as a List or a Set), you can create a sequence from it by calling `asSequence()`.

```
val numbers = listOf("one", "two", "three", "four")
val numbersSequence = numbers.asSequence()
```

From a function

One more way to create a sequence is by building it with a function that calculates its elements. To build a sequence based on a function, call `generateSequence()` with this function as an argument. Optionally, you can specify the first element as an explicit value or a result of a function call. The sequence generation stops when the provided function returns null. So, the sequence in the example below is infinite.

```
fun main() {
//sampleStart
    val oddNumbers = generateSequence(1) { it + 2 } // `it` is the previous element
    println(oddNumbers.take(5).toList())
    //println(oddNumbers.count())      // error: the sequence is infinite
//sampleEnd
}
```

To create a finite sequence with `generateSequence()`, provide a function that returns null after the last element you need.

```

fun main() {
//sampleStart
    val oddNumbersLessThan10 = generateSequence(1) { if (it < 8) it + 2 else null }
    println(oddNumbersLessThan10.count())
//sampleEnd
}

```

From chunks

Finally, there is a function that lets you produce sequence elements one by one or by chunks of arbitrary sizes – the `sequence()` function. This function takes a lambda expression containing calls of `yield()` and `yieldAll()` functions. They return an element to the sequence consumer and suspend the execution of `sequence()` until the next element is requested by the consumer. `yield()` takes a single element as an argument; `yieldAll()` can take an Iterable object, an Iterator, or another Sequence. A Sequence argument of `yieldAll()` can be infinite. However, such a call must be the last: all subsequent calls will never be executed.

```

fun main() {
//sampleStart
    val oddNumbers = sequence {
        yield(1)
        yieldAll(listOf(3, 5))
        yieldAll(generateSequence(7) { it + 2 })
    }
    println(oddNumbers.take(5).toList())
//sampleEnd
}

```

Sequence operations

The sequence operations can be classified into the following groups regarding their state requirements:

- Stateless operations require no state and process each element independently, for example, `map()` or `filter()`. Stateless operations can also require a small constant amount of state to process an element, for example, `take()` or `drop()`.
- Stateful operations require a significant amount of state, usually proportional to the number of elements in a sequence.

If a sequence operation returns another sequence, which is produced lazily, it's called intermediate. Otherwise, the operation is terminal. Examples of terminal operations are `toList()` or `sum()`. Sequence elements can be retrieved only with terminal operations.

Sequences can be iterated multiple times; however some sequence implementations might constrain themselves to be iterated only once. That is mentioned specifically in their documentation.

Sequence processing example

Let's take a look at the difference between Iterable and Sequence with an example.

Iterable

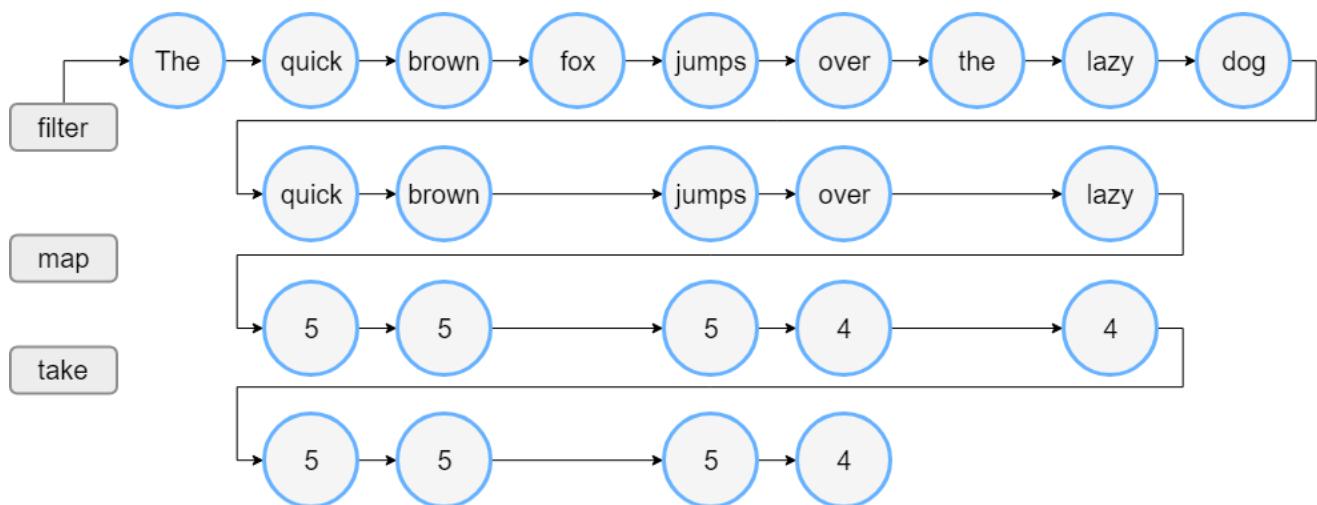
Assume that you have a list of words. The code below filters the words longer than three characters and prints the lengths of first four such words.

```
fun main() {
//sampleStart
    val words = "The quick brown fox jumps over the lazy dog".split(" ")
    val lengthsList = words.filter { println("filter: $it"); it.length > 3 }
        .map { println("Length: ${it.length}"); it.length }
        .take(4)

    println("Lengths of first 4 words longer than 3 chars:")
    println(lengthsList)
//sampleEnd
}
```

When you run this code, you'll see that the filter() and map() functions are executed in the same order as they appear in the code. First, you see filter: for all elements, then length: for the elements left after filtering, and then the output of the two last lines.

This is how the list processing goes:



List processing

Sequence

Now let's write the same with sequences:

```

fun main() {
//sampleStart
    val words = "The quick brown fox jumps over the lazy dog".split(" ")
    //convert the List to a Sequence
    val wordsSequence = words.asSequence()

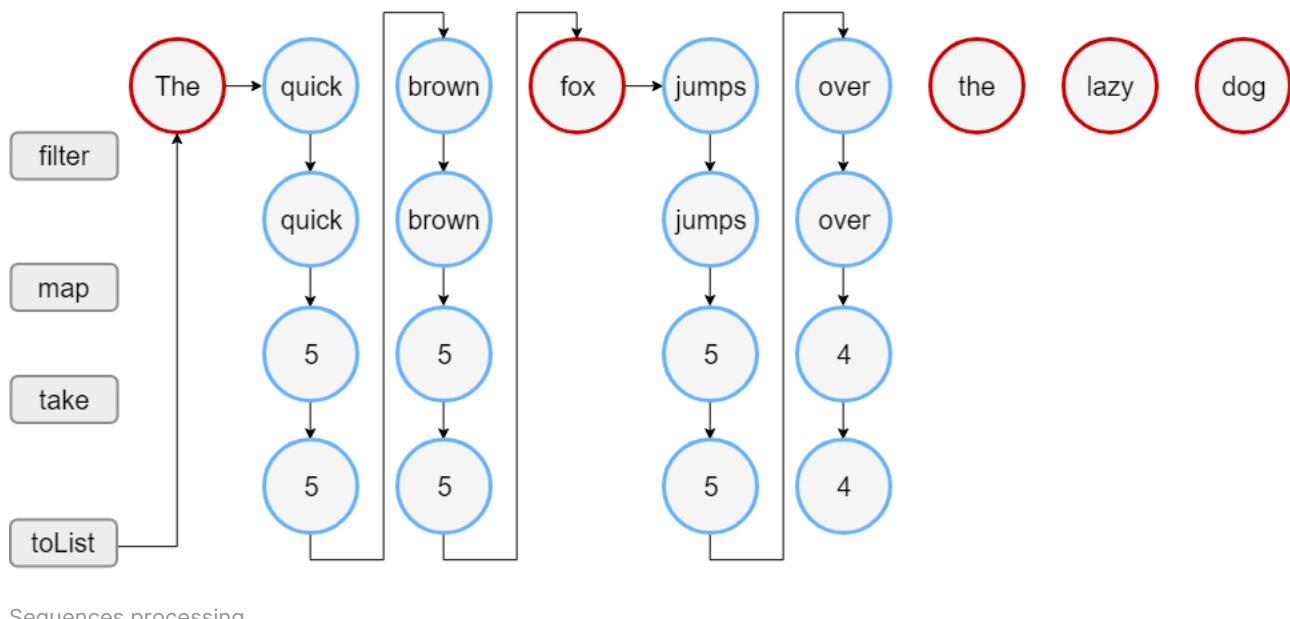
    val lengthsSequence = wordsSequence.filter { println("filter: $it"); it.length > 3 }
        .map { println("length: ${it.length}"); it.length }
        .take(4)

    println("Lengths of first 4 words longer than 3 chars")
    // terminal operation: obtaining the result as a List
    println(lengthsSequence.toList())
//sampleEnd
}

```

The output of this code shows that the filter() and map() functions are called only when building the result list. So, you first see the line of text “Lengths of..” and then the sequence processing starts. Note that for elements left after filtering, the map executes before filtering the next element. When the result size reaches 4, the processing stops because it's the largest possible size that take(4) can return.

The sequence processing goes like this:



In this example, the sequence processing takes 18 steps instead of 23 steps for doing the same with lists.

Collection operations overview

The Kotlin standard library offers a broad variety of functions for performing operations on collections. This includes simple operations, such as getting or adding elements, as well as more complex ones including search, sorting,

filtering, transformations, and so on.

Extension and member functions

Collection operations are declared in the standard library in two ways: [member functions](#) of collection interfaces and [extension functions](#).

Member functions define operations that are essential for a collection type. For example, [Collection](#) contains the function [isEmpty\(\)](#) for checking its emptiness; [List](#) contains [get\(\)](#) for index access to elements, and so on.

When you create your own implementations of collection interfaces, you must implement their member functions. To make the creation of new implementations easier, use the skeletal implementations of collection interfaces from the standard library: [AbstractCollection](#), [AbstractList](#), [AbstractSet](#), [AbstractMap](#), and their mutable counterparts.

Other collection operations are declared as extension functions. These are filtering, transformation, ordering, and other collection processing functions.

Common operations

Common operations are available for both [read-only](#) and [mutable collections](#). Common operations fall into these groups:

- [Transformations](#)
- [Filtering](#)
- [plus and minus operators](#)
- [Grouping](#)
- [Retrieving collection parts](#)
- [Retrieving single elements](#)
- [Ordering](#)
- [Aggregate operations](#)

Operations described on these pages return their results without affecting the original collection. For example, a filtering operation produces a new collection that contains all the elements matching the filtering predicate. Results of such operations should be either stored in variables, or used in some other way, for example, passed in other functions.

```
fun main() {  
    //sampleStart  
    val numbers = listOf("one", "two", "three", "four")  
    numbers.filter { it.length > 3 } // nothing happens with `numbers`, result is lost
```

```

    println("numbers are still $numbers")
    val longerThan3 = numbers.filter { it.length > 3 } // result is stored in `longerThan3`
    println("numbers longer than 3 chars are $longerThan3")
//sampleEnd
}

```

For certain collection operations, there is an option to specify the destination object. Destination is a mutable collection to which the function appends its resulting items instead of returning them in a new object. For performing operations with destinations, there are separate functions with the To postfix in their names, for example, `filterTo()` instead of `filter()` or `associateTo()` instead of `associate()`. These functions take the destination collection as an additional parameter.

```

fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val filterResults = mutableListOf<String>() //destination object
    numbers.filterTo(filterResults) { it.length > 3 }
    numbers.filterIndexedTo(filterResults) { index, _ -> index == 0 }
    println(filterResults) // contains results of both operations
//sampleEnd
}

```

For convenience, these functions return the destination collection back, so you can create it right in the corresponding argument of the function call:

```

fun main() {
    val numbers = listOf("one", "two", "three", "four")
//sampleStart
    // filter numbers right into a new hash set,
    // thus eliminating duplicates in the result
    val result = numbers.mapTo(HashSet()) { it.length }
    println("distinct item lengths are $result")
//sampleEnd
}

```

Functions with destination are available for filtering, association, grouping, flattening, and other operations. For the complete list of destination operations see the [Kotlin collections reference](#).

Write operations

For mutable collections, there are also write operations that change the collection state. Such operations include adding, removing, and updating elements. Write operations are listed in the [Write operations](#) and corresponding sections of [List-specific operations](#) and [Map specific operations](#).

For certain operations, there are pairs of functions for performing the same operation: one applies the operation in-place and the other returns the result as a separate collection. For example, `sort()` sorts a mutable collection in-place, so its state changes; `sorted()` creates a new collection that contains the same elements in the sorted order.

```

fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    val sortedNumbers = numbers.sorted()
    println(numbers == sortedNumbers) // false
    numbers.sort()
    println(numbers == sortedNumbers) // true
//sampleEnd
}

```

Collection transformation operations

The Kotlin standard library provides a set of extension functions for collection transformations. These functions build new collections from existing ones based on the transformation rules provided. In this page, we'll give an overview of the available collection transformation functions.

Map

The mapping transformation creates a collection from the results of a function on the elements of another collection. The basic mapping function is `map()`. It applies the given lambda function to each subsequent element and returns the list of the lambda results. The order of results is the same as the original order of elements. To apply a transformation that additionally uses the element index as an argument, use `mapIndexed()`.

```

fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3)
    println(numbers.map { it * 3 })
    println(numbers.mapIndexed { idx, value -> value * idx })
//sampleEnd
}

```

If the transformation produces null on certain elements, you can filter out the nulls from the result collection by calling the `mapNotNull()` function instead of `map()`, or `mapIndexedNotNull()` instead of `mapIndexed()`.

```

fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3)
    println(numbers.mapNotNull { if (it == 2) null else it * 3 })
    println(numbers.mapIndexedNotNull { idx, value -> if (idx == 0) null else value * idx })
//sampleEnd
}

```

When transforming maps, you have two options: transform keys leaving values unchanged and vice versa. To apply a

given transformation to keys, use `mapKeys()`; in turn, `mapValues()` transforms values. Both functions use the transformations that take a map entry as an argument, so you can operate both its key and value.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    println(numbersMap.mapKeys { it.key.uppercase() })
    println(numbersMap.mapValues { it.value + it.key.length })
//sampleEnd
}
```

Zip

Zipping transformation is building pairs from elements with the same positions in both collections. In the Kotlin standard library, this is done by the `zip()` extension function.

When called on a collection or an array with another collection (array) as an argument, `zip()` returns the List of Pair objects. The elements of the receiver collection are the first elements in these pairs.

If the collections have different sizes, the result of the `zip()` is the smaller size; the last elements of the larger collection are not included in the result.

`zip()` can also be called in the infix form a `zip b`.

```
fun main() {
//sampleStart
    val colors = listOf("red", "brown", "grey")
    val animals = listOf("fox", "bear", "wolf")
    println(colors.zip(animals))

    val twoAnimals = listOf("fox", "bear")
    println(colors.zip(twoAnimals))
//sampleEnd
}
```

You can also call `zip()` with a transformation function that takes two parameters: the receiver element and the argument element. In this case, the result List contains the return values of the transformation function called on pairs of the receiver and the argument elements with the same positions.

```
fun main() {
//sampleStart
    val colors = listOf("red", "brown", "grey")
    val animals = listOf("fox", "bear", "wolf")

    println(colors.zip(animals) { color, animal -> "The ${animal.replaceFirstChar { it.uppercase() }} is $color"})
//sampleEnd
}
```

```
}
```

When you have a List of Pair s, you can do the reverse transformation – unzipping – that builds two lists from these pairs:

- The first list contains the first elements of each Pair in the original list.
- The second list contains the second elements.

To unzip a list of pairs, call `unzip()`.

```
fun main() {
//sampleStart
    val numberPairs = listOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)
    println(numberPairs.unzip())
//sampleEnd
}
```

Associate

Association transformations allow building maps from the collection elements and certain values associated with them. In different association types, the elements can be either keys or values in the association map.

The basic association function `associateWith()` creates a Map in which the elements of the original collection are keys, and values are produced from them by the given transformation function. If two elements are equal, only the last one remains in the map.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
//sampleEnd
}
```

For building maps with collection elements as values, there is the function `associateBy()`. It takes a function that returns a key based on an element's value. If two elements are equal, only the last one remains in the map.

`associateBy()` can also be called with a value transformation function.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println(numbers.associateBy { it.first().uppercaseChar() })
    println(numbers.associateBy(keySelector = { it.first().uppercaseChar() }, valueTransform =
{ it.length }))
//sampleEnd
}
```

```
//sampleEnd  
}
```

Another way to build maps in which both keys and values are somehow produced from collection elements is the function [associate\(\)](#). It takes a lambda function that returns a Pair: the key and the value of the corresponding map entry.

Note that associate() produces short-living Pair objects which may affect the performance. Thus, associate() should be used when the performance isn't critical or it's more preferable than other options.

An example of the latter is when a key and the corresponding value are produced from an element together.

```
fun main() {  
    data class FullName (val firstName: String, val lastName: String)  
  
    fun parseFullName(fullName: String): FullName {  
        val nameParts = fullName.split(" ")  
        if (nameParts.size == 2) {  
            return FullName(nameParts[0], nameParts[1])  
        } else throw Exception("Wrong name format")  
    }  
  
    //sampleStart  
    val names = listOf("Alice Adams", "Brian Brown", "Clara Campbell")  
    println(names.associate { name -> parseFullName(name).let { it.lastName to it.firstName } })  
}  
//sampleEnd
```

Here we call a transform function on an element first, and then build a pair from the properties of that function's result.

Flatten

If you operate nested collections, you may find the standard library functions that provide flat access to nested collection elements useful.

The first function is [flatten\(\)](#). You can call it on a collection of collections, for example, a List of Set s. The function returns a single List of all the elements of the nested collections.

```
fun main() {  
    //sampleStart  
    val numberSets = listOf(setOf(1, 2, 3), setOf(4, 5, 6), setOf(1, 2))  
    println(numberSets.flatten())  
}  
//sampleEnd
```

Another function – [flatMap\(\)](#) provides a flexible way to process nested collections. It takes a function that maps a collection element to another collection. As a result, flatMap() returns a single list of its return values on all the

elements. So, flatMap() behaves as a subsequent call of map() (with a collection as a mapping result) and flatten().

```
data class StringContainer(val values: List<String>)

fun main() {
//sampleStart
    val containers = listOf(
        StringContainer(listOf("one", "two", "three")),
        StringContainer(listOf("four", "five", "six")),
        StringContainer(listOf("seven", "eight"))
    )
    println(containers.flatMap { it.values })
//sampleEnd
}
```

String representation

If you need to retrieve the collection content in a readable format, use functions that transform the collections to strings: `joinToString()` and `joinTo()`.

`joinToString()` builds a single String from the collection elements based on the provided arguments. `joinTo()` does the same but appends the result to the given [Appendable](#) object.

When called with the default arguments, the functions return the result similar to calling `toString()` on the collection: a String of elements' string representations separated by commas with spaces.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println(numbers)
    println(numbers.joinToString())

    val listString = StringBuffer("The list of numbers: ")
    numbers.joinTo(listString)
    println(listString)
//sampleEnd
}
```

To build a custom string representation, you can specify its parameters in function arguments `separator`, `prefix`, and `postfix`. The resulting string will start with the prefix and end with the postfix. The separator will come after each element except the last.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
```

```
    println(numbers.joinToString(separator = " | ", prefix = "start: ", postfix = ": end"))
//sampleEnd
}
```

For bigger collections, you may want to specify the limit – a number of elements that will be included into result. If the collection size exceeds the limit, all the other elements will be replaced with a single value of the truncated argument.

```
fun main() {
//sampleStart
    val numbers = (1..100).toList()
    println(numbers.joinToString(limit = 10, truncated = "<...>"))
//sampleEnd
}
```

Finally, to customize the representation of elements themselves, provide the transform function.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.joinToString { "Element: ${it.toUpperCase()}" })
//sampleEnd
}
```

Filtering collections

Filtering is one of the most popular tasks in collection processing. In Kotlin, filtering conditions are defined by predicates – lambda functions that take a collection element and return a boolean value: true means that the given element matches the predicate, false means the opposite.

The standard library contains a group of extension functions that let you filter collections in a single call. These functions leave the original collection unchanged, so they are available for both mutable and read-only collections. To operate the filtering result, you should assign it to a variable or chain the functions after filtering.

Filter by predicate

The basic filtering function is `filter()`. When called with a predicate, `filter()` returns the collection elements that match it. For both List and Set, the resulting collection is a List, for Map it's a Map as well.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val longerThan3 = numbers.filter { it.length > 3 }
    println(longerThan3)
}
```

```

    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
    println(filteredMap)
//sampleEnd
}

```

The predicates in `filter()` can only check the values of the elements. If you want to use element positions in the filter, use `filterIndexed()`. It takes a predicate with two arguments: the index and the value of an element.

To filter collections by negative conditions, use `filterNot()`. It returns a list of elements for which the predicate yields false.

```

fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    val filteredIdx = numbers.filterIndexed { index, s -> (index != 0) && (s.length < 5) }
    val filteredNot = numbers.filterNot { it.length <= 3 }

    println(filteredIdx)
    println(filteredNot)
//sampleEnd
}

```

There are also functions that narrow the element type by filtering elements of a given type:

- `filterIsInstance()` returns collection elements of a given type. Being called on a `List<Any>`, `filterIsInstance<T>()` returns a `List<T>`, thus allowing you to call functions of the `T` type on its items.

```

fun main() {
//sampleStart
    val numbers = listOf(null, 1, "two", 3.0, "four")
    println("All String elements in upper case:")
    numbers.filterIsInstance<String>().forEach {
        println(it.uppercase())
    }
//sampleEnd
}

```

- `filterNotNull()` returns all non-null elements. Being called on a `List<T?>`, `filterNotNull()` returns a `List<T: Any>`, thus allowing you to treat the elements as non-null objects.

```

fun main() {
//sampleStart
    val numbers = listOf(null, "one", "two", null)
    numbers.filterNotNull().forEach {
        println(it.length) // length is unavailable for nullable Strings
    }
//sampleEnd
}

```

Partition

Another filtering function – `partition()` – filters a collection by a predicate and keeps the elements that don't match it in a separate list. So, you have a Pair of List s as a return value: the first list containing elements that match the predicate and the second one containing everything else from the original collection.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val (match, rest) = numbers.partition { it.length > 3 }

    println(match)
    println(rest)
//sampleEnd
}
```

Test predicates

Finally, there are functions that simply test a predicate against collection elements:

- `any()` returns true if at least one element matches the given predicate.
- `none()` returns true if none of the elements match the given predicate.
- `all()` returns true if all elements match the given predicate. Note that `all()` returns true when called with any valid predicate on an empty collection. Such behavior is known in logic as vacuous truth.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println(numbers.any { it.endsWith("e") })
    println(numbers.none { it.endsWith("a") })
    println(numbers.all { it.endsWith("e") })

    println(emptyList<Int>().all { it > 5 }) // vacuous truth
//sampleEnd
}
```

`any()` and `none()` can also be used without a predicate: in this case they just check the collection emptiness. `any()` returns true if there are elements and false if there aren't; `none()` does the opposite.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val empty = emptyList<String>()

    println(numbers.any())
    println(empty.any())
//sampleEnd
}
```

```
    println(numbers.none())
    println(empty.none())
//sampleEnd
}
```

Plus and minus operators

In Kotlin, [plus](#) (+) and [minus](#) (-) operators are defined for collections. They take a collection as the first operand; the second operand can be either an element or another collection. The return value is a new read-only collection:

- The result of plus contains the elements from the original collection and from the second operand.
- The result of minus contains the elements of the original collection except the elements from the second operand. If it's an element, minus removes its first occurrence; if it's a collection, all occurrences of its elements are removed.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    val plusList = numbers + "five"
    val minusList = numbers - listOf("three", "four")
    println(plusList)
    println(minusList)
//sampleEnd
}
```

For the details on plus and minus operators for maps, see [Map specific operations](#). The [augmented assignment operators plusAssign](#) (+=) and [minusAssign](#) (-=) are also defined for collections. However, for read-only collections, they actually use the plus or minus operators and try to assign the result to the same variable. Thus, they are available only on var read-only collections. For mutable collections, they modify the collection if it's a val. For more details see [Collection write operations](#).

Grouping

The Kotlin standard library provides extension functions for grouping collection elements. The basic function [groupBy\(\)](#) takes a lambda function and returns a Map. In this map, each key is the lambda result and the corresponding value is the List of elements on which this result is returned. This function can be used, for example, to group a list of String s by their first letter.

You can also call [groupBy\(\)](#) with a second lambda argument – a value transformation function. In the result map of [groupBy\(\)](#) with two lambdas, the keys produced by [keySelector](#) function are mapped to the results of the value transformation function instead of the original elements.

```
fun main() {
```

```
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")

    println(numbers.groupBy { it.first().uppercase() })
    println(numbers.groupBy(keySelector = { it.first() }, valueTransform = { it.uppercase()
}))}
//sampleEnd
}
```

If you want to group elements and then apply an operation to all groups at one time, use the function `groupingBy()`. It returns an instance of the `Grouping` type. The `Grouping` instance lets you apply operations to all groups in a lazy manner: the groups are actually built right before the operation execution.

Namely, `Grouping` supports the following operations:

- `eachCount()` counts the elements in each group.
- `fold()` and `reduce()` perform `fold` and `reduce` operations on each group as a separate collection and return the results.
- `aggregate()` applies a given operation subsequently to all the elements in each group and returns the result. This is the generic way to perform any operations on a `Grouping`. Use it to implement custom operations when `fold` or `reduce` are not enough.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.groupingBy { it.first() }.eachCount())
//sampleEnd
}
```

Retrieve collection parts

The Kotlin standard library contains extension functions for retrieving parts of a collection. These functions provide a variety of ways to select elements for the result collection: listing their positions explicitly, specifying the result size, and others.

Slice

`slice()` returns a list of the collection elements with given indices. The indices may be passed either as a `range` or as a collection of integer values.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.slice(1..3))
    println(numbers.slice(0..4 step 2))
//sampleEnd
}
```

```
    println(numbers.slice(setOf(3, 5, 0)))
//sampleEnd
}
```

Take and drop

To get the specified number of elements starting from the first, use the [take\(\)](#) function. For getting the last elements, use [takeLast\(\)](#). When called with a number larger than the collection size, both functions return the whole collection.

To take all the elements except a given number of first or last elements, call the [drop\(\)](#) and [dropLast\(\)](#) functions respectively.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.take(3))
    println(numbers.takeLast(3))
    println(numbers.drop(1))
    println(numbers.dropLast(5))
//sampleEnd
}
```

You can also use predicates to define the number of elements for taking or dropping. There are four functions similar to the ones described above:

- [takeWhile\(\)](#) is [take\(\)](#) with a predicate: it takes the elements up to but excluding the first one not matching the predicate. If the first collection element doesn't match the predicate, the result is empty.
- [takeLastWhile\(\)](#) is similar to [takeLast\(\)](#): it takes the range of elements matching the predicate from the end of the collection. The first element of the range is the element next to the last element not matching the predicate. If the last collection element doesn't match the predicate, the result is empty;
- [dropWhile\(\)](#) is the opposite to [takeWhile\(\)](#) with the same predicate: it returns the elements from the first one not matching the predicate to the end.
- [dropLastWhile\(\)](#) is the opposite to [takeLastWhile\(\)](#) with the same predicate: it returns the elements from the beginning to the last one not matching the predicate.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.takeWhile { !it.startsWith('f') })
    println(numbers.takeLastWhile { it != "three" })
    println(numbers.dropWhile { it.length == 3 })
    println(numbers.dropLastWhile { it.contains('i') })
//sampleEnd
}
```

Chunked

To break a collection onto parts of a given size, use the `chunked()` function. `chunked()` takes a single argument – the size of the chunk – and returns a List of List s of the given size. The first chunk starts from the first element and contains the size elements, the second chunk holds the next size elements, and so on. The last chunk may have a smaller size.

```
fun main() {  
    //sampleStart  
    val numbers = (0..13).toList()  
    println(numbers.chunked(3))  
    //sampleEnd  
}
```

You can also apply a transformation for the returned chunks right away. To do this, provide the transformation as a lambda function when calling `chunked()`. The lambda argument is a chunk of the collection. When `chunked()` is called with a transformation, the chunks are short-living List s that should be consumed right in that lambda.

```
fun main() {  
    //sampleStart  
    val numbers = (0..13).toList()  
    println(numbers.chunked(3) { it.sum() }) // `it` is a chunk of the original collection  
    //sampleEnd  
}
```

Windowed

You can retrieve all possible ranges of the collection elements of a given size. The function for getting them is called `windowed()`: it returns a list of element ranges that you would see if you were looking at the collection through a sliding window of the given size. Unlike `chunked()`, `windowed()` returns element ranges (windows) starting from each collection element. All the windows are returned as elements of a single List.

```
fun main() {  
    //sampleStart  
    val numbers = listOf("one", "two", "three", "four", "five")  
    println(numbers.windowed(3))  
    //sampleEnd  
}
```

`windowed()` provides more flexibility with optional parameters:

- `step` defines a distance between first elements of two adjacent windows. By default the value is 1, so the result contains windows starting from all elements. If you increase the step to 2, you will receive only windows starting from odd elements: first, third, and so on.

- `partialWindows` includes windows of smaller sizes that start from the elements at the end of the collection. For example, if you request windows of three elements, you can't build them for the last two elements. Enabling `partialWindows` in this case includes two more lists of sizes 2 and 1.

Finally, you can apply a transformation to the returned ranges right away. To do this, provide the transformation as a lambda function when calling `windowed()`.

```
fun main() {
//sampleStart
    val numbers = (1..10).toList()
    println(numbers.windowed(3, step = 2, partialWindows = true))
    println(numbers.windowed(3) { it.sum() })
//sampleEnd
}
```

To build two-element windows, there is a separate function - `zipWithNext()`. It creates pairs of adjacent elements of the receiver collection. Note that `zipWithNext()` doesn't break the collection into pairs; it creates a Pair for each element except the last one, so its result on [1, 2, 3, 4] is [[1, 2], [2, 3], [3, 4]], not [[1, 2], [3, 4]]. `zipWithNext()` can be called with a transformation function as well; it should take two elements of the receiver collection as arguments.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.zipWithNext())
    println(numbers.zipWithNext() { s1, s2 -> s1.length > s2.length })
//sampleEnd
}
```

Retrieve single elements

Kotlin collections provide a set of functions for retrieving single elements from collections. Functions described on this page apply to both lists and sets.

As the [definition of list](#) says, a list is an ordered collection. Hence, every element of a list has its position that you can use for referring. In addition to functions described on this page, lists offer a wider set of ways to retrieve and search for elements by indices. For more details, see [List-specific operations](#).

In turn, set is not an ordered collection by [definition](#). However, the Kotlin Set stores elements in certain orders. These can be the order of insertion (in `LinkedHashSet`), natural sorting order (in `SortedSet`), or another order. The order of a set of elements can also be unknown. In such cases, the elements are still ordered somehow, so the functions that rely on the element positions still return their results. However, such results are unpredictable to the caller unless they know the specific implementation of Set used.

Retrieve by position

For retrieving an element at a specific position, there is the function `elementAt()`. Call it with the integer number as an argument, and you'll receive the collection element at the given position. The first element has the position 0, and the last one is `(size - 1)`.

`elementAt()` is useful for collections that do not provide indexed access, or are not statically known to provide one. In case of `List`, it's more idiomatic to use `indexed access operator (get() or [])`.

```
fun main() {
//sampleStart
    val numbers = linkedSetOf("one", "two", "three", "four", "five")
    println(numbers.elementAt(3))

    val numbersSortedSet = sortedSetOf("one", "two", "three", "four")
    println(numbersSortedSet.elementAt(0)) // elements are stored in the ascending order
//sampleEnd
}
```

There are also useful aliases for retrieving the first and the last element of the collection: `first()` and `last()`.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.first())
    println(numbers.last())
//sampleEnd
}
```

To avoid exceptions when retrieving element with non-existing positions, use safe variations of `elementAt()`:

- `elementAtOrNull()` returns null when the specified position is out of the collection bounds.
- `elementAtOrElse()` additionally takes a lambda function that maps an Int argument to an instance of the collection element type. When called with an out-of-bounds position, the `elementAtOrElse()` returns the result of the lambda on the given value.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.elementAtOrNull(5))
    println(numbers.elementAtOrElse(5) { index -> "The value for index $index is undefined"})
//sampleEnd
}
```

Retrieve by condition

Functions `first()` and `last()` also let you search a collection for elements matching a given predicate. When you call `first()` with a predicate that tests a collection element, you'll receive the first element on which the predicate yields true. In turn, `last()` with a predicate returns the last element matching it.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.first { it.length > 3 })
    println(numbers.last { it.startsWith("f") })
//sampleEnd
}
```

If no elements match the predicate, both functions throw exceptions. To avoid them, use `firstOrNull()` and `lastOrNull()` instead: they return null if no matching elements are found.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.firstOrNull { it.length > 6 })
//sampleEnd
}
```

Use the aliases if their names suit your situation better:

- `find()` instead of `firstOrNull()`
- `findLast()` instead of `lastOrNull()`

```
fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.find { it % 2 == 0 })
    println(numbers.findLast { it % 2 == 0 })
//sampleEnd
}
```

Retrieve with selector

If you need to map the collection before retrieving the element, there is a function `firstNotNullOf()`. It combines 2 actions:

- Maps the collection with the selector function
- Returns the first non-null value in the result

`firstNotNullOf()` throws the `NoSuchElementException` if the resulting collection doesn't have a non-null element. Use the counterpart `firstNotNullOrNull()` to return null in this case.

```
fun main() {
//sampleStart
    val list = listOf<Any>(0, "true", false)
    // Converts each element to string and returns the first one that has required length
    val longEnough = list.firstNotNullOf { item -> item.toString().takeIf { it.length >= 4 } }
    println(longEnough)
//sampleEnd
}
```

Random element

If you need to retrieve an arbitrary element of a collection, call the `random()` function. You can call it without arguments or with a `Random` object as a source of the randomness.

```
fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.random())
//sampleEnd
}
```

On empty collections, `random()` throws an exception. To receive null instead, use `randomOrNull()`.

Check element existence

To check the presence of an element in a collection, use the `contains()` function. It returns true if there is a collection element that equals() the function argument. You can call `contains()` in the operator form with the `in` keyword.

To check the presence of multiple instances together at once, call `containsAll()` with a collection of these instances as an argument.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.contains("four"))
    println("zero" in numbers)

    println(numbers.containsAll(listOf("four", "two")))
    println(numbers.containsAll(listOf("one", "zero")))
//sampleEnd
}
```

Additionally, you can check if the collection contains any elements by calling `isEmpty()` or `isNotEmpty()`.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.isEmpty())
    println(numbers.isNotEmpty())

    val empty = emptyList<String>()
    println(empty.isEmpty())
    println(empty.isNotEmpty())
//sampleEnd
}
```

Ordering

The order of elements is an important aspect of certain collection types. For example, two lists of the same elements are not equal if their elements are ordered differently.

In Kotlin, the orders of objects can be defined in several ways.

First, there is natural order. It is defined for inheritors of the [Comparable](#) interface. Natural order is used for sorting them when no other order is specified.

Most built-in types are comparable:

- Numeric types use the traditional numerical order: 1 is greater than 0; -3.4f is greater than -5f, and so on.
- Char and String use the [lexicographical order](#): b is greater than a; world is greater than hello.

To define a natural order for a user-defined type, make the type an inheritor of Comparable. This requires implementing the `compareTo()` function. `compareTo()` must take another object of the same type as an argument and return an integer value showing which object is greater:

- Positive values show that the receiver object is greater.
- Negative values show that it's less than the argument.
- Zero shows that the objects are equal.

Below is a class for ordering versions that consist of the major and the minor part.

```
class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int {
        if (this.major != other.major) {
            return this.major - other.major
        } else if (this.minor != other.minor) {
            return this.minor - other.minor
        } else return 0
    }
}
```

```

    }
}

fun main() {
    println(Version(1, 2) > Version(1, 3))
    println(Version(2, 0) > Version(1, 5))
}

```

Custom orders let you sort instances of any type in a way you like. Particularly, you can define an order for non-comparable objects or define an order other than natural for a comparable type. To define a custom order for a type, create a [Comparator](#) for it. Comparator contains the compare() function: it takes two instances of a class and returns the integer result of the comparison between them. The result is interpreted in the same way as the result of a compareTo() as is described above.

```

fun main() {
//sampleStart
    val lengthComparator = Comparator { str1: String, str2: String -> str1.length -
str2.length }
    println(listOf("aaa", "bb", "c").sortedWith(lengthComparator))
//sampleEnd
}

```

Having the lengthComparator, you are able to arrange strings by their length instead of the default lexicographical order.

A shorter way to define a Comparator is the [compareBy\(\)](#) function from the standard library. compareBy() takes a lambda function that produces a Comparable value from an instance and defines the custom order as the natural order of the produced values.

With compareBy(), the length comparator from the example above looks like this:

```

fun main() {
//sampleStart
    println(listOf("aaa", "bb", "c").sortedWith(compareBy { it.length }))
//sampleEnd
}

```

The Kotlin collections package provides functions for sorting collections in natural, custom, and even random orders. On this page, we'll describe sorting functions that apply to [read-only](#) collections. These functions return their result as a new collection containing the elements of the original collection in the requested order. To learn about functions for sorting [mutable](#) collections in place, see the [List-specific operations](#).

Natural order

The basic functions [sorted\(\)](#) and [sortedDescending\(\)](#) return elements of a collection sorted into ascending and descending sequence according to their natural order. These functions apply to collections of Comparable elements.

```

fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println("Sorted ascending: ${numbers.sorted()}")
    println("Sorted descending: ${numbers.sortedDescending()}")
//sampleEnd
}

```

Custom orders

For sorting in custom orders or sorting non-comparable objects, there are the functions [sortedBy\(\)](#) and [sortedByDescending\(\)](#). They take a selector function that maps collection elements to Comparable values and sort the collection in natural order of that values.

```

fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    val sortedNumbers = numbers.sortedBy { it.length }
    println("Sorted by length ascending: $sortedNumbers")
    val sortedByLast = numbers.sortedByDescending { it.last() }
    println("Sorted by the last letter descending: $sortedByLast")
//sampleEnd
}

```

To define a custom order for the collection sorting, you can provide your own Comparator. To do this, call the [sortedWith\(\)](#) function passing in your Comparator. With this function, sorting strings by their length looks like this:

```

fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println("Sorted by length ascending: ${numbers.sortedWith(compareBy { it.length })}")
//sampleEnd
}

```

Reverse order

You can retrieve the collection in the reversed order using the [reversed\(\)](#) function.

```

fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.reversed())
//sampleEnd
}

```

`reversed()` returns a new collection with the copies of the elements. So, if you change the original collection later, this won't affect the previously obtained results of `reversed()`.

Another reversing function - `asReversed()`

- returns a reversed view of the same collection instance, so it may be more lightweight and preferable than `reversed()` if the original list is not going to change.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val reversedNumbers = numbers.asReversed()
    println(reversedNumbers)
//sampleEnd
}
```

If the original list is mutable, all its changes reflect in its reversed views and vice versa.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    val reversedNumbers = numbers.asReversed()
    println(reversedNumbers)
    numbers.add("five")
    println(reversedNumbers)
//sampleEnd
}
```

However, if the mutability of the list is unknown or the source is not a list at all, `reversed()` is more preferable since its result is a copy that won't change in the future.

Random order

Finally, there is a function that returns a new List containing the collection elements in a random order - `shuffled()`. You can call it without arguments or with a `Random` object.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.shuffled())
//sampleEnd
}
```

Aggregate operations

Kotlin collections contain functions for commonly used aggregate operations – operations that return a single value based on the collection content. Most of them are well known and work the same way as they do in other languages:

- minOrNull() and maxOrNull() return the smallest and the largest element respectively. On empty collections, they return null.
- average() returns the average value of elements in the collection of numbers.
- sum() returns the sum of elements in the collection of numbers.
- count() returns the number of elements in a collection.

```
fun main() {  
    val numbers = listOf(6, 42, 10, 4)  
  
    println("Count: ${numbers.count()}")  
    println("Max: ${numbers.maxOrNull()}")  
    println("Min: ${numbers.minOrNull()}")  
    println("Average: ${numbers.average()}")  
    println("Sum: ${numbers.sum()}")  
}
```

There are also functions for retrieving the smallest and the largest elements by certain selector function or custom Comparator:

- maxByOrNull() and minByOrNull() take a selector function and return the element for which it returns the largest or the smallest value.
- maxWithOrNull() and minWithOrNull() take a Comparator object and return the largest or smallest element according to that Comparator.

These functions return null on empty collections. There are also alternatives for maxByOrNull() and minByOrNull(): maxOf() and minOf(), which do the same but throw a NoSuchElementException on empty collections.

```
fun main() {  
    //sampleStart  
    val numbers = listOf(5, 42, 10, 4)  
    val min3Remainder = numbers.minByOrNull { it % 3 }  
    println(min3Remainder)  
  
    val strings = listOf("one", "two", "three", "four")  
    val longestString = strings.maxWithOrNull(compareBy { it.length })  
    println(longestString)  
    //sampleEnd  
}
```

Besides regular sum(), there is an advanced summation function sumOf() that takes a selector function and returns the sum of its application to all collection elements. Selector can return different numeric types: Int, Long, Double, UInt, and ULong (also BigInteger and BigDecimal on the JVM).

```

fun main() {
//sampleStart
    val numbers = listOf(5, 42, 10, 4)
    println(numbers.sumOf { it * 2 })
    println(numbers.sumOf { it.toDouble() / 2 })
//sampleEnd
}

```

Fold and reduce

For more specific cases, there are the functions `reduce()` and `fold()` that apply the provided operation to the collection elements sequentially and return the accumulated result. The operation takes two arguments: the previously accumulated value and the collection element.

The difference between the two functions is that `fold()` takes an initial value and uses it as the accumulated value on the first step, whereas the first step of `reduce()` uses the first and the second elements as operation arguments on the first step.

```

fun main() {
//sampleStart
    val numbers = listOf(5, 2, 10, 4)

    val sum = numbers.reduce { sum, element -> sum + element }
    println(sum)
    val sumDoubled = numbers.fold(0) { sum, element -> sum + element * 2 }
    println(sumDoubled)

    //val sumDoubledReduce = numbers.reduce { sum, element -> sum + element * 2 } //incorrect:
    //the first element isn't doubled in the result
    //println(sumDoubledReduce)
//sampleEnd
}

```

The example above shows the difference: `fold()` is used for calculating the sum of doubled elements. If you pass the same function to `reduce()`, it will return another result because it uses the list's first and second elements as arguments on the first step, so the first element won't be doubled.

To apply a function to elements in the reverse order, use functions `reduceRight()` and `foldRight()`. They work in a way similar to `fold()` and `reduce()` but start from the last element and then continue to previous. Note that when folding or reducing right, the operation arguments change their order: first goes the element, and then the accumulated value.

```

fun main() {
//sampleStart
    val numbers = listOf(5, 2, 10, 4)
    val sumDoubledRight = numbers.foldRight(0) { element, sum -> sum + element * 2 }
    println(sumDoubledRight)
//sampleEnd
}

```

```
//sampleEnd  
}
```

You can also apply operations that take element indices as parameters. For this purpose, use functions [reduceIndexed\(\)](#) and [foldIndexed\(\)](#) passing element index as the first argument of the operation.

Finally, there are functions that apply such operations to collection elements from right to left - [reduceRightIndexed\(\)](#) and [foldRightIndexed\(\)](#).

```
fun main() {  
    //sampleStart  
    val numbers = listOf(5, 2, 10, 4)  
    val sumEven = numbers.foldIndexed(0) { idx, sum, element -> if (idx % 2 == 0) sum +  
        element else sum }  
    println(sumEven)  
  
    val sumEvenRight = numbers.foldRightIndexed(0) { idx, element, sum -> if (idx % 2 == 0)  
        sum + element else sum }  
    println(sumEvenRight)  
    //sampleEnd  
}
```

All reduce operations throw an exception on empty collections. To receive null instead, use their `*OrNull()` counterparts:

- [reduceOrNull\(\)](#)
- [reduceRightOrNull\(\)](#)
- [reduceIndexedOrNull\(\)](#)
- [reduceRightIndexedOrNull\(\)](#)

For cases where you want to save intermediate accumulator values, there are functions [runningFold\(\)](#) (or its synonym [scan\(\)](#)) and [runningReduce\(\)](#).

```
fun main() {  
    //sampleStart  
    val numbers = listOf(0, 1, 2, 3, 4, 5)  
    val runningReduceSum = numbers.runningReduce { sum, item -> sum + item }  
    val runningFoldSum = numbers.runningFold(10) { sum, item -> sum + item }  
    //sampleEnd  
    val transform = { index: Int, element: Int -> "N = ${index + 1}: $element" }  
    println(runningReduceSum.mapIndexed(transform).joinToString("\n", "Sum of first N elements  
with runningReduce:\n"))  
    println(runningFoldSum.mapIndexed(transform).joinToString("\n", "Sum of first N elements  
with runningFold:\n"))  
}
```

If you need an index in the operation parameter, use [runningFoldIndexed\(\)](#) or [runningReduceIndexed\(\)](#).

Collection write operations

[Mutable collections](#) support operations for changing the collection contents, for example, adding or removing elements. On this page, we'll describe write operations available for all implementations of `MutableCollection`. For more specific operations available for `List` and `Map`, see [List-specific Operations](#) and [Map Specific Operations](#) respectively.

Adding elements

To add a single element to a list or a set, use the `add()` function. The specified object is appended to the end of the collection.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.add(5)
    println(numbers)
//sampleEnd
}
```

`addAll()` adds every element of the argument object to a list or a set. The argument can be an `Iterable`, a `Sequence`, or an `Array`. The types of the receiver and the argument may be different, for example, you can add all items from a `Set` to a `List`.

When called on lists, `addAll()` adds new elements in the same order as they go in the argument. You can also call `addAll()` specifying an element position as the first argument. The first element of the argument collection will be inserted at this position. Other elements of the argument collection will follow it, shifting the receiver elements to the end.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 5, 6)
    numbers.addAll(arrayOf(7, 8))
    println(numbers)
    numbers.addAll(2, setOf(3, 4))
    println(numbers)
//sampleEnd
}
```

You can also add elements using the in-place version of the [plus operator](#)- `plusAssign` (`+=`) When applied to a mutable collection, `+=` appends the second operand (an element or another collection) to the end of the collection.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two")
```

```

numbers += "three"
println(numbers)
numbers += listOf("four", "five")
println(numbers)
//sampleEnd
}

```

Removing elements

To remove an element from a mutable collection, use the [remove\(\)](#) function. `remove()` accepts the element value and removes one occurrence of this value.

```

fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4, 3)
    numbers.remove(3) // removes the first `3`
    println(numbers)
    numbers.remove(5) // removes nothing
    println(numbers)
//sampleEnd
}

```

For removing multiple elements at once, there are the following functions :

- [removeAll\(\)](#) removes all elements that are present in the argument collection. Alternatively, you can call it with a predicate as an argument; in this case the function removes all elements for which the predicate yields true.
- [retainAll\(\)](#) is the opposite of `removeAll()`: it removes all elements except the ones from the argument collection. When used with a predicate, it leaves only elements that match it.
- [clear\(\)](#) removes all elements from a list and leaves it empty.

```

fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    println(numbers)
    numbers.retainAll { it >= 3 }
    println(numbers)
    numbers.clear()
    println(numbers)

    val numbersSet = mutableSetOf("one", "two", "three", "four")
    numbersSet.removeAll(setOf("one", "two"))
    println(numbersSet)
//sampleEnd
}

```

Another way to remove elements from a collection is with the [minusAssign](#) (-=) operator – the in-place version of [minus](#). The second argument can be a single instance of the element type or another collection. With a single element on the

right-hand side, `-=` removes the first occurrence of it. In turn, if it's a collection, all occurrences of its elements are removed. For example, if a list contains duplicate elements, they are removed at once. The second operand can contain elements that are not present in the collection. Such elements don't affect the operation execution.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "three", "four")
    numbers -= "three"
    println(numbers)
    numbers -= listOf("four", "five")
    //numbers -= listOf("four")      // does the same as above
    println(numbers)
//sampleEnd
}
```

Updating elements

Lists and maps also provide operations for updating elements. They are described in [List-specific Operations](#) and [Map Specific Operations](#). For sets, updating doesn't make sense since it's actually removing an element and adding another one.

List-specific operations

[List](#) is the most popular type of built-in collection in Kotlin. Index access to the elements of lists provides a powerful set of operations for lists.

Retrieve elements by index

Lists support all common operations for element retrieval: `elementAt()`, `first()`, `last()`, and others listed in [Retrieve single elements](#). What is specific for lists is index access to the elements, so the simplest way to read an element is retrieving it by index. That is done with the `get()` function with the index passed in the argument or the shorthand `[index]` syntax.

If the list size is less than the specified index, an exception is thrown. There are two other functions that help you avoid such exceptions:

- `getOrDefault()` lets you provide the function for calculating the default value to return if the index isn't present in the collection.
- `getOrNull()` returns null as the default value.

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

```

val numbers = listOf(1, 2, 3, 4)
println(numbers.get(0))
println(numbers[0])
//numbers.get(5)                                // exception!
println(numbers.getOrNull(5))                   // null
println(numbers.getOrDefault(5, {it}))           // 5
//sampleEnd
}

```

Retrieve list parts

In addition to common operations for [Retrieving Collection Parts](#), lists provide the `subList()` function that returns a view of the specified elements range as a list. Thus, if an element of the original collection changes, it also changes in the previously created sublists and vice versa.

```

fun main() {
//sampleStart
    val numbers = (0..13).toList()
    println(numbers.subList(3, 6))
//sampleEnd
}

```

Find element positions

Linear search

In any lists, you can find the position of an element using the functions `indexOf()` and `lastIndexOf()`. They return the first and the last position of an element equal to the given argument in the list. If there are no such elements, both functions return -1.

```

fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3, 4, 2, 5)
    println(numbers.indexOf(2))
    println(numbers.lastIndexOf(2))
//sampleEnd
}

```

There is also a pair of functions that take a predicate and search for elements matching it:

- `indexOfFirst()` returns the index of the first element matching the predicate or -1 if there are no such elements.
- `indexOfLast()` returns the index of the last element matching the predicate or -1 if there are no such elements.

```

fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    println(numbers.indexOfFirst { it > 2})
    println(numbers.indexOfLast { it % 2 == 1})
//sampleEnd
}

```

Binary search in sorted lists

There is one more way to search elements in lists – [binary search](#). It works significantly faster than other built-in search functions but requires the list to be [sorted](#) in ascending order according to a certain ordering: natural or another one provided in the function parameter. Otherwise, the result is undefined.

To search an element in a sorted list, call the [binarySearch\(\)](#) function passing the value as an argument. If such an element exists, the function returns its index; otherwise, it returns (-insertionPoint - 1) where insertionPoint is the index where this element should be inserted so that the list remains sorted. If there is more than one element with the given value, the search can return any of their indices.

You can also specify an index range to search in: in this case, the function searches only between two provided indices.

```

fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    numbers.sort()
    println(numbers)
    println(numbers.binarySearch("two")) // 3
    println(numbers.binarySearch("z")) // -5
    println(numbers.binarySearch("two", 0, 2)) // -3
//sampleEnd
}

```

Comparator binary search

When list elements aren't Comparable, you should provide a [Comparator](#) to use in the binary search. The list must be sorted in ascending order according to this Comparator. Let's have a look at an example:

```

data class Product(val name: String, val price: Double)

fun main() {
//sampleStart
    val productList = listOf(
        Product("WebStorm", 49.0),
        Product("AppCode", 99.0),
        Product("DotTrace", 129.0),
        Product("ReSharper", 149.0))

```

```

    println(productList.binarySearch(Product("AppCode", 99.0), compareBy<Product> { it.price
}.thenBy { it.name }))  

//sampleEnd
}

```

Here's a list of Product instances that aren't Comparable and a Comparator that defines the order: product p1 precedes product p2 if p1 's price is less than p2 's price. So, having a list sorted ascending according to this order, we use binarySearch() to find the index of the specified Product.

Custom comparators are also handy when a list uses an order different from natural one, for example, a case-insensitive order for String elements.

```

fun main() {
//sampleStart
    val colors = listOf("Blue", "green", "ORANGE", "Red", "yellow")
    println(colors.binarySearch("RED", String.CASE_INSENSITIVE_ORDER)) // 3
//sampleEnd
}

```

Comparison binary search

Binary search with comparison function lets you find elements without providing explicit search values. Instead, it takes a comparison function mapping elements to Int values and searches for the element where the function returns zero. The list must be sorted in the ascending order according to the provided function; in other words, the return values of comparison must grow from one list element to the next one.

```

import kotlin.math.sign
//sampleStart
data class Product(val name: String, val price: Double)

fun priceComparison(product: Product, price: Double) = sign(product.price - price).toInt()

fun main() {
    val productList = listOf(
        Product("WebStorm", 49.0),
        Product("AppCode", 99.0),
        Product("DotTrace", 129.0),
        Product("ReSharper", 149.0))

    println(productList.binarySearch { priceComparison(it, 99.0) })
}
//sampleEnd

```

Both comparator and comparison binary search can be performed for list ranges as well.

List write operations

In addition to the collection modification operations described in [Collection write operations](#), `mutable` lists support specific write operations. Such operations use the index to access elements to broaden the list modification capabilities.

Add

To add elements to a specific position in a list, use `add()` and `addAll()` providing the position for element insertion as an additional argument. All elements that come after the position shift to the right.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "five", "six")
    numbers.add(1, "two")
    numbers.addAll(2, listOf("three", "four"))
    println(numbers)
//sampleEnd
}
```

Update

Lists also offer a function to replace an element at a given position - `set()` and its operator form `[]`. `set()` doesn't change the indexes of other elements.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "five", "three")
    numbers[1] = "two"
    println(numbers)
//sampleEnd
}
```

`fill()` simply replaces all the collection elements with the specified value.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.fill(3)
    println(numbers)
//sampleEnd
}
```

Remove

To remove an element at a specific position from a list, use the `removeAt()` function providing the position as an argument. All indices of elements that come after the element being removed will decrease by one.

```

fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4, 3)
    numbers.removeAt(1)
    println(numbers)
//sampleEnd
}

```

Sort

In [Collection Ordering](#), we describe operations that retrieve collection elements in specific orders. For mutable lists, the standard library offers similar extension functions that perform the same ordering operations in place. When you apply such an operation to a list instance, it changes the order of elements in that exact instance.

The in-place sorting functions have similar names to the functions that apply to read-only lists, but without the ed/d suffix:

- sort* instead of sorted* in the names of all sorting functions: `sort()`, `sortDescending()`, `sortBy()`, and so on.
- `shuffle()` instead of `shuffled()`.
- `reverse()` instead of `reversed()`.

`asReversed()` called on a mutable list returns another mutable list which is a reversed view of the original list. Changes in that view are reflected in the original list. The following example shows sorting functions for mutable lists:

```

fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")

    numbers.sort()
    println("Sort into ascending: $numbers")
    numbers.sortDescending()
    println("Sort into descending: $numbers")

    numbers.sortBy { it.length }
    println("Sort into ascending by length: $numbers")
    numbers.sortByDescending { it.last() }
    println("Sort into descending by the last letter: $numbers")

    numbers.sortWith(compareBy<String> { it.length }.thenBy { it })
    println("Sort by Comparator: $numbers")

    numbers.shuffle()
    println("Shuffle: $numbers")

    numbers.reverse()
    println("Reverse: $numbers")
//sampleEnd
}

```

Set-specific operations

The Kotlin collections package contains extension functions for popular operations on sets: finding intersections, merging, or subtracting collections from each other.

To merge two collections into one, use the `union()` function. It can be used in the infix form `a union b`. Note that for ordered collections the order of the operands is important: in the resulting collection, the elements of the first operand go before the elements of the second.

To find an intersection between two collections (elements present in both of them), use `intersect()`. To find collection elements not present in another collection, use `subtract()`. Both these functions can be called in the infix form as well, for example, `a intersect b`.

```
fun main() {
//sampleStart
    val numbers = setOf("one", "two", "three")

    println(numbers union setOf("four", "five"))
    println(setOf("four", "five") union numbers)

    println(numbers intersect setOf("two", "one"))
    println(numbers subtract setOf("three", "four"))
    println(numbers subtract setOf("four", "three")) // same output
//sampleEnd
}
```

Note that set operations are supported by `List` as well. However, the result of set operations on lists is still a `Set`, so all the duplicate elements are removed.

Map-specific operations

In `maps`, types of both keys and values are user-defined. Key-based access to map entries enables various map-specific processing capabilities from getting a value by key to separate filtering of keys and values. On this page, we provide descriptions of the map processing functions from the standard library.

Retrieve keys and values

For retrieving a value from a map, you must provide its key as an argument of the `get()` function. The shorthand `[key]` syntax is also supported. If the given key is not found, it returns `null`. There is also the function `getValue()` which has slightly different behavior: it throws an exception if the key is not found in the map. Additionally, you have two more options to handle the key absence:

- `getOrElse()` works the same way as for lists: the values for non-existent keys are returned from the given lambda function.

- `getOrDefault()` returns the specified default value if the key is not found.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap.get("one"))
    println(numbersMap["one"])
    println(numbersMap.getOrDefault("four", 10))
    println(numbersMap["five"]) // null
    //numbersMap.getValue("six") // exception!
//sampleEnd
}
```

To perform operations on all keys or all values of a map, you can retrieve them from the properties `keys` and `values` accordingly. `keys` is a set of all map keys and `values` is a collection of all map values.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap.keys)
    println(numbersMap.values)
//sampleEnd
}
```

Filter

You can filter maps with the `filter()` function as well as other collections. When calling `filter()` on a map, pass to it a predicate with a `Pair` as an argument. This enables you to use both the key and the value in the filtering predicate.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
    println(filteredMap)
//sampleEnd
}
```

There are also two specific ways for filtering maps: by keys and by values. For each way, there is a function: `filterKeys()` and `filterValues()`. Both return a new map of entries which match the given predicate. The predicate for `filterKeys()` checks only the element keys, the one for `filterValues()` checks only values.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredKeysMap = numbersMap.filterKeys { it.endsWith("1") }
```

```

val filteredValuesMap = numbersMap.filterValues { it < 10 }

println(filteredKeysMap)
println(filteredValuesMap)
//sampleEnd
}

```

Plus and minus operators

Due to the key access to elements, plus (+) and minus (-) operators work for maps differently than for other collections. plus returns a Map that contains elements of its both operands: a Map on the left and a Pair or another Map on the right. When the right-hand side operand contains entries with keys present in the left-hand side Map, the result map contains the entries from the right side.

```

fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap + Pair("four", 4))
    println(numbersMap + Pair("one", 10))
    println(numbersMap + mapOf("five" to 5, "one" to 11))
//sampleEnd
}

```

minus creates a Map from entries of a Map on the left except those with keys from the right-hand side operand. So, the right-hand side operand can be either a single key or a collection of keys: list, set, and so on.

```

fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap - "one")
    println(numbersMap - listOf("two", "four"))
//sampleEnd
}

```

For details on using plusAssign (+=) and minusAssign (-=) operators on mutable maps, see [Map write operations](#) below.

Map write operations

Mutable maps offer map-specific write operations. These operations let you change the map content using the key-based access to the values.

There are certain rules that define write operations on maps:

- Values can be updated. In turn, keys never change: once you add an entry, its key is constant.

- For each key, there is always a single value associated with it. You can add and remove whole entries.

Below are descriptions of the standard library functions for write operations available on mutable maps.

Add and update entries

To add a new key-value pair to a mutable map, use `put()`. When a new entry is put into a `LinkedHashMap` (the default map implementation), it is added so that it comes last when iterating the map. In sorted maps, the positions of new elements are defined by the order of their keys.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap.put("three", 3)
    println(numbersMap)
//sampleEnd
}
```

To add multiple entries at a time, use `putAll()`. Its argument can be a Map or a group of Pairs: Iterable, Sequence, or Array.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
    numbersMap.putAll(mapOf("four" to 4, "five" to 5))
    println(numbersMap)
//sampleEnd
}
```

Both `put()` and `putAll()` overwrite the values if the given keys already exist in the map. Thus, you can use them to update values of map entries.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    val previousValue = numbersMap.put("one", 11)
    println("value associated with 'one', before: $previousValue, after: ${numbersMap["one"]}")
    println(numbersMap)
//sampleEnd
}
```

You can also add new entries to maps using the shorthand operator form. There are two ways:

- `plusAssign (+=)` operator.
- the `[]` operator alias for `set()`.

```

fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap["three"] = 3      // calls numbersMap.put("three", 3)
    numbersMap += mapOf("four" to 4, "five" to 5)
    println(numbersMap)
//sampleEnd
}

```

When called with the key present in the map, operators overwrite the values of the corresponding entries.

Remove entries

To remove an entry from a mutable map, use the [remove\(\)](#) function. When calling `remove()`, you can pass either a key or a whole key-value-pair. If you specify both the key and value, the element with this key will be removed only if its value matches the second argument.

```

fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
    numbersMap.remove("one")
    println(numbersMap)
    numbersMap.remove("three", 4)           //doesn't remove anything
    println(numbersMap)
//sampleEnd
}

```

You can also remove entries from a mutable map by their keys or values. To do this, call `remove()` on the map's keys or values providing the key or the value of an entry. When called on values, `remove()` removes only the first entry with the given value.

```

fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3, "threeAgain" to 3)
    numbersMap.keys.remove("one")
    println(numbersMap)
    numbersMap.values.remove(3)
    println(numbersMap)
//sampleEnd
}

```

The [minusAssign](#) (`-=`) operator is also available for mutable maps.

```

fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
    numbersMap -= "two"

```

```

println(numbersMap)
numbersMap -= "five"           //doesn't remove anything
println(numbersMap)
//sampleEnd
}

```

Scope functions

The Kotlin standard library contains several functions whose sole purpose is to execute a block of code within the context of an object. When you call such a function on an object with a [lambda expression](#) provided, it forms a temporary scope. In this scope, you can access the object without its name. Such functions are called scope functions. There are five of them: let, run, with, apply, and also.

Basically, these functions do the same: execute a block of code on an object. What's different is how this object becomes available inside the block and what is the result of the whole expression.

Here's a typical usage of a scope function:

```

data class Person(var name: String, var age: Int, var city: String) {
    fun moveTo(newCity: String) { city = newCity }
    fun incrementAge() { age++ }
}

fun main() {
//sampleStart
    Person("Alice", 20, "Amsterdam").let {
        println(it)
        it.moveTo("London")
        it.incrementAge()
        println(it)
    }
//sampleEnd
}

```

If you write the same without let, you'll have to introduce a new variable and repeat its name whenever you use it.

```

data class Person(var name: String, var age: Int, var city: String) {
    fun moveTo(newCity: String) { city = newCity }
    fun incrementAge() { age++ }
}

fun main() {
//sampleStart
    val alice = Person("Alice", 20, "Amsterdam")
    println(alice)
    alice.moveTo("London")
    alice.incrementAge()
    println(alice)
//sampleEnd
}

```

The scope functions do not introduce any new technical capabilities, but they can make your code more concise and readable.

Due to the similar nature of scope functions, choosing the right one for your case can be a bit tricky. The choice mainly depends on your intent and the consistency of use in your project. Below we'll provide detailed descriptions of the distinctions between scope functions and the conventions on their usage.

Function selection

To help you choose the right scope function for your purpose, we provide the table of key differences between them.

Function	Object reference	Return value	Is extension function
let	it	Lambda result	Yes
run	this	Lambda result	Yes
run	-	Lambda result	No: called without the context object
with	this	Lambda result	No: takes the context object as an argument.
apply	this	Context object	Yes
also	it	Context object	Yes

The detailed information about the differences is provided in the dedicated sections below.

Here is a short guide for choosing scope functions depending on the intended purpose:

- Executing a lambda on non-null objects: let
- Introducing an expression as a variable in local scope: let
- Object configuration: apply
- Object configuration and computing the result: run
- Running statements where an expression is required: non-extension run
- Additional effects: also
- Grouping function calls on an object: with

The use cases of different functions overlap, so that you can choose the functions based on the specific conventions used in your project or team.

Although the scope functions are a way of making the code more concise, avoid overusing them: it can decrease your

code readability and lead to errors. Avoid nesting scope functions and be careful when chaining them: it's easy to get confused about the current context object and the value of this or it.

Distinctions

Because the scope functions are all quite similar in nature, it's important to understand the differences between them. There are two main differences between each scope function:

- The way to refer to the context object
- The return value.

Context object: this or it

Inside the lambda of a scope function, the context object is available by a short reference instead of its actual name. Each scope function uses one of two ways to access the context object: as a lambda receiver (this) or as a lambda argument (it). Both provide the same capabilities, so we'll describe the pros and cons of each for different cases and provide recommendations on their use.

```
fun main() {
    val str = "Hello"
    // this
    str.run {
        println("The receiver string length: $length")
        //println("The receiver string length: ${this.length}") // does the same
    }

    // it
    str.let {
        println("The receiver string's length is ${it.length}")
    }
}
```

this

run, with, and apply refer to the context object as a lambda receiver - by keyword this. Hence, in their lambdas, the object is available as it would be in ordinary class functions. In most cases, you can omit this when accessing the members of the receiver object, making the code shorter. On the other hand, if this is omitted, it can be hard to distinguish between the receiver members and external objects or functions. So, having the context object as a receiver (this) is recommended for lambdas that mainly operate on the object members: call its functions or assign properties.

```
data class Person(var name: String, var age: Int = 0, var city: String = "")

fun main() {
//sampleStart
    val adam = Person("Adam").apply {
        age = 20                      // same as this.age = 20 or adam.age = 20
    }
}
```

```

        city = "London"
    }
    println(adam)
//sampleEnd
}

```

it

In turn, let and also have the context object as a lambda argument. If the argument name is not specified, the object is accessed by the implicit default name it. It is shorter than this and expressions with it are usually easier for reading. However, when calling the object functions or properties you don't have the object available implicitly like this. Hence, having the context object as it is better when the object is mostly used as an argument in function calls. It is also better if you use multiple variables in the code block.

```

import kotlin.random.Random

fun writeToFile(message: String) {
    println("INFO: $message")
}

fun main() {
//sampleStart
    fun getRandomInt(): Int {
        return Random.nextInt(100).also {
            writeToFile("getRandomInt() generated value $it")
        }
    }

    val i = getRandomInt()
//sampleEnd
}

```

Additionally, when you pass the context object as an argument, you can provide a custom name for the context object inside the scope.

```

import kotlin.random.Random

fun writeToFile(message: String) {
    println("INFO: $message")
}

fun main() {
//sampleStart
    fun getRandomInt(): Int {
        return Random.nextInt(100).also { value ->
            writeToFile("getRandomInt() generated value $value")
        }
    }

    val i = getRandomInt()
//sampleEnd
}

```

Return value

The scope functions differ by the result they return:

- apply and also return the context object.
- let, run, and with return the lambda result.

These two options let you choose the proper function depending on what you do next in your code.

Context object

The return value of apply and also is the context object itself. Hence, they can be included into call chains as side steps: you can continue chaining function calls on the same object after them.

```
fun main() {  
    //sampleStart  
    val numberList = mutableListOf<Double>()  
    numberList.also { println("Populating the list") }  
        .apply {  
            add(2.71)  
            add(3.14)  
            add(1.0)  
        }  
        .also { println("Sorting the list") }  
        .sort()  
    //sampleEnd  
    println(numberList)  
}
```

They also can be used in return statements of functions returning the context object.

```
import kotlin.random.Random  
  
fun writeToFile(message: String) {  
    println("INFO: $message")  
}  
  
fun main() {  
    //sampleStart  
    fun getRandomInt(): Int {  
        return Random.nextInt(100).also {  
            writeToFile("getRandomInt() generated value $it")  
        }  
    }  
  
    val i = getRandomInt()  
    //sampleEnd  
}
```

Lambda result

let, run, and with return the lambda result. So, you can use them when assigning the result to a variable, chaining

operations on the result, and so on.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    val countEndsWithE = numbers.run {
        add("four")
        add("five")
        count { it.endsWith("e") }
    }
    println("There are $countEndsWithE elements that end with e.")
//sampleEnd
}
```

Additionally, you can ignore the return value and use a scope function to create a temporary scope for variables.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    with(numbers) {
        val firstItem = first()
        val lastItem = last()
        println("First item: $firstItem, last item: $lastItem")
    }
//sampleEnd
}
```

Functions

To help you choose the right scope function for your case, we'll describe them in detail and provide usage recommendations. Technically, functions are interchangeable in many cases, so the examples show the conventions that define the common usage style.

let

The context object is available as an argument (it). The return value is the lambda result.

let can be used to invoke one or more functions on results of call chains. For example, the following code prints the results of two operations on a collection:

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four", "five")
    val resultList = numbers.map { it.length }.filter { it > 3 }
    println(resultList)
//sampleEnd
}
```

With let, you can rewrite it:

```

fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four", "five")
    numbers.map { it.length }.filter { it > 3 }.let {
        println(it)
        // and more function calls if needed
    }
//sampleEnd
}

```

If the code block contains a single function with it as an argument, you can use the method reference (::) instead of the lambda:

```

fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four", "five")
    numbers.map { it.length }.filter { it > 3 }.let(::println)
//sampleEnd
}

```

let is often used for executing a code block only with non-null values. To perform actions on a non-null object, use the safe call operator ?. on it and call let with the actions in its lambda.

```

fun processNonNullString(str: String) {}

fun main() {
//sampleStart
    val str: String? = "Hello"
    //processNonNullString(str)      // compilation error: str can be null
    val length = str?.let {
        println("let() called on $it")
        processNonNullString(it)    // OK: 'it' is not null inside '?.let { }'
        it.length
    }
//sampleEnd
}

```

Another case for using let is introducing local variables with a limited scope for improving code readability. To define a new variable for the context object, provide its name as the lambda argument so that it can be used instead of the default it.

```

fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val modifiedFirstItem = numbers.first().let { firstItem ->
        println("The first item of the list is '$firstItem'")
        if (firstItem.length >= 5) firstItem else "!" + firstItem + "!"
    }.uppercase()
    println("First item after modifications: '$modifiedFirstItem'")
//sampleEnd
}

```

with

A non-extension function: the context object is passed as an argument, but inside the lambda, it's available as a receiver (this). The return value is the lambda result.

We recommend with for calling functions on the context object without providing the lambda result. In the code, with can be read as “with this object, do the following.”

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    with(numbers) {
        println("'with' is called with argument $this")
        println("It contains $size elements")
    }
//sampleEnd
}
```

Another use case for with is introducing a helper object whose properties or functions will be used for calculating a value.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    val firstAndLast = with(numbers) {
        "The first element is ${first()}, "
        " the last element is ${last()}"
    }
    println(firstAndLast)
//sampleEnd
}
```

run

The context object is available as a receiver (this). The return value is the lambda result.

run does the same as with but invokes as let- as an extension function of the context object.

run is useful when your lambda contains both the object initialization and the computation of the return value.

```
class MultiportService(var url: String, var port: Int) {
    fun prepareRequest(): String = "Default request"
    fun query(request: String): String = "Result for query '$request'"
}

fun main() {
//sampleStart
    val service = MultiportService("https://example.kotlinlang.org", 80)

    val result = service.run {
        port = 8080
        query(prepareRequest() + " to port $port")
    }
}
```

```
// the same code written with let() function:
val letResult = service.let {
    it.port = 8080
    it.query(it.prepareRequest() + " to port ${it.port}")
}
//sampleEnd
    println(result)
    println(letResult)
}
```

Besides calling run on a receiver object, you can use it as a non-extension function. Non-extension run lets you execute a block of several statements where an expression is required.

```
fun main() {
//sampleStart
    val hexNumberRegex = run {
        val digits = "0-9"
        val hexDigits = "A-Fa-f"
        val sign = "+-"

        Regex("[${sign}]?[${digits}${hexDigits}]+")
    }

    for (match in hexNumberRegex.findAll("+1234 -FFFF not-a-number")) {
        println(match.value)
    }
//sampleEnd
}
```

apply

The context object is available as a receiver (this). The return value is the object itself.

Use apply for code blocks that don't return a value and mainly operate on the members of the receiver object. The common case for apply is the object configuration. Such calls can be read as “apply the following assignments to the object.”

```
data class Person(var name: String, var age: Int = 0, var city: String = "")

fun main() {
//sampleStart
    val adam = Person("Adam").apply {
        age = 32
        city = "London"
    }
    println(adam)
//sampleEnd
}
```

Having the receiver as the return value, you can easily include apply into call chains for more complex processing.

also

The context object is available as an argument (it). The return value is the object itself.

also is good for performing some actions that take the context object as an argument. Use also for actions that need a reference rather to the object than to its properties and functions, or when you don't want to shadow this reference from an outer scope.

When you see also in the code, you can read it as “and also do the following with the object.”

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    numbers
        .also { println("The list elements before adding new one: $it") }
        .add("four")
//sampleEnd
}
```

takelf and takeUnless

In addition to scope functions, the standard library contains the functions takelf and takeUnless. These functions let you embed checks of the object state in call chains.

When called on an object with a predicate provided, takelf returns this object if it matches the predicate. Otherwise, it returns null. So, takelf is a filtering function for a single object. In turn, takeUnless returns the object if it doesn't match the predicate and null if it does. The object is available as a lambda argument (it).

```
import kotlin.random.*

fun main() {
//sampleStart
    val number = Random.nextInt(100)

    val evenOrNull = number.takeIf { it % 2 == 0 }
    val oddOrNull = number.takeUnless { it % 2 == 0 }
    println("even: $evenOrNull, odd: $oddOrNull")
//sampleEnd
}
```

When chaining other functions after takelf and takeUnless, don't forget to perform the null check or the safe call (?.) because their return value is nullable.

```
fun main() {
//sampleStart
    val str = "Hello"
    val caps = str.takeIf { it.isNotEmpty() }?.uppercase()
    //val caps = str.takeIf { it.isNotEmpty() }.uppercase() //compilation error
    println(caps)
//sampleEnd
}
```

```
}
```

takelf and takeUnless are especially useful together with scope functions. A good case is chaining them with let for running a code block on objects that match the given predicate. To do this, call takelf on the object and then call let with a safe call (?). For objects that don't match the predicate, takelf returns null and let isn't invoked.

```
fun main() {
//sampleStart
    fun displaySubstringPosition(input: String, sub: String) {
        input.indexOf(sub).takeIf { it >= 0 }?.let {
            println("The substring $sub is found in $input.")
            println("Its start position is $it.")
        }
    }

    displaySubstringPosition("010000011", "11")
    displaySubstringPosition("010000011", "12")
//sampleEnd
}
```

This is how the same function looks without the standard library functions:

```
fun main() {
//sampleStart
    fun displaySubstringPosition(input: String, sub: String) {
        val index = input.indexOf(sub)
        if (index >= 0) {
            println("The substring $sub is found in $input.")
            println("Its start position is $index.")
        }
    }

    displaySubstringPosition("010000011", "11")
    displaySubstringPosition("010000011", "12")
//sampleEnd
}
```

Opt-in requirements

The opt-in requirement annotations `@RequiresOptIn` and `@OptIn` 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 it in [YouTrack](#).

`@RequireOptIn` and `@OptIn` annotations were introduced in 1.3.70 to replace previously used `@Experimental` and `@UseExperimental`; at the same time, `-Xopt-in` compiler option replaced `-Xuse-experimental`.

The Kotlin standard library provides a mechanism for requiring and giving explicit consent for using certain elements of APIs. This mechanism lets library developers inform users of their APIs about specific conditions that require opt-in, for example, if an API is in the experimental state and is likely to change in the future.

To prevent potential issues, the compiler warns users of such APIs about these conditions and requires them to opt in before using the API.

Opt in to using API

If a library author marks a declaration from a library's API as [requiring opt-in](#), you should give an explicit consent for using it in your code. There are several ways to opt in to such APIs, all applicable without technical limitations. You are free to choose the way that you find best for your situation.

Propagating opt-in

When you use an API in the code intended for third-party use (a library), you can propagate its opt-in requirement to your API as well. To do this, annotate your declaration with the [opt-in requirement annotation](#) of the API used in its body. This enables you to use the API elements marked with this annotation.

```
// library code
@RequiresOptIn(message = "This API is experimental. It may be changed in the future without
notice.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime // Opt-in requirement annotation

@MyDateTime
class DateProvider // A class requiring opt-in
```

```
// client code
fun getYear(): Int {
    val dateProvider: DateProvider // Error: DateProvider requires opt-in
    // ...
}

@MyDateTime
fun getDate(): Date {
    val dateProvider: DateProvider // OK: the function requires opt-in as well
    // ...
}

fun displayDate() {
    println(getDate()) // error: getDate() requires opt-in
}
```

As you can see in this example, the annotated function appears to be a part of the @MyDateTime API. So, such an opt-in propagates the opt-in requirement to the client code; its clients will see the same warning message and be required to consent as well. To use multiple APIs that require opt-in, mark the declaration with all their opt-in requirement

annotations.

Non-propagating opt-in

In modules that don't expose their own API, such as applications, you can opt in to using APIs without propagating the opt-in requirement to your code. In this case, mark your declaration with `@OptIn` passing the opt-in requirement annotation as its argument:

```
// library code
@RequiresOptIn(message = "This API is experimental. It may be changed in the future without
notice.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime // Opt-in requirement annotation

@MyDateTime
class DateProvider // A class requiring opt-in
```

```
//client code
@OptIn(MyDateTime::class)
fun getDate(): Date { // Uses DateProvider; doesn't propagate the opt-in requirement
    val dateProvider: DateProvider
    // ...
}

fun displayDate() {
    println(getDate()) // OK: opt-in is not required
}
```

When somebody calls the function `getDate()`, they won't be informed about the opt-in requirements for APIs used in its body.

To use an API that requires opt-in in all functions and classes in a file, add the file-level annotation `@file:OptIn` to the top of the file before the package specification and imports.

```
//client code
@file:OptIn(MyDateTime::class)
```

Module-wide opt-in

If you don't want to annotate every usage of APIs that require opt-in, you can opt in to them for your whole module. To opt in to using an API in a module, compile it with the argument `-Xopt-in`, specifying the fully qualified name of the opt-in requirement annotation of the API you use: `-Xopt-in=org.mylibrary.OptInAnnotation`. Compiling with this argument has the same effect as if every declaration in the module had the annotation `@OptIn(OptInAnnotation::class)`.

If you build your module with Gradle, you can add arguments like this:

Groovy

```
tasks.withType(KotlinCompile).configureEach {  
    kotlinOptions {  
        freeCompilerArgs += "-Xopt-in=org.mylibrary.OptInAnnotation"  
    }  
}
```

Kotlin

```
tasks.withType<KotlinCompile>().configureEach {  
    kotlinOptions.freeCompilerArgs += "-Xopt-in=org.mylibrary.OptInAnnotation"  
}
```

If your Gradle module is a multiplatform module, use the `useExperimentalAnnotation` method:

Groovy

```
sourceSets {  
    all {  
        languageSettings {  
            useExperimentalAnnotation('org.mylibrary.OptInAnnotation')  
        }  
    }  
}
```

Kotlin

```
sourceSets {  
    all {  
        languageSettings.useExperimentalAnnotation("org.mylibrary.OptInAnnotation")  
    }  
}
```

For Maven, it would be:

```
<build>  
    <plugins>  
        <plugin>  
            <groupId>org.jetbrains.kotlin</groupId>  
            <artifactId>kotlin-maven-plugin</artifactId>  
            <version>${kotlin.version}</version>  
            <executions>...</executions>  
            <configuration>  
                <args>  
                    <arg>-Xopt-in=org.mylibrary.OptInAnnotation</arg>  
                </args>  
            </configuration>
```

```
</plugin>
</plugins>
</build>
```

To opt in to multiple APIs on the module level, add one of the described arguments for each opt-in requirement marker used in your module.

Require opt-in for API

Create opt-in requirement annotations

If you want to require explicit consent to using your module's API, create an annotation class to use as an opt-in requirement annotation. This class must be annotated with `@RequiresOptIn`:

```
@RequiresOptIn
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime
```

Opt-in requirement annotations must meet several requirements:

- BINARY retention
- No EXPRESSION and FILE among targets
- No parameters.

An opt-in requirement can have one of two severity levels:

- `RequiresOptIn.Level.ERROR`. Opt-in is mandatory. Otherwise, the code that uses marked API won't compile.
Default level.
- `RequiresOptIn.Level.WARNING`. Opt-in is not mandatory, but advisable. Without it, the compiler raises a warning.

To set the desired level, specify the `level` parameter of the `@RequiresOptIn` annotation.

Additionally, you can provide a message to inform API users about special condition of using the API. The compiler will show it to users that use the API without opt-in.

```
@RequiresOptIn(level = RequiresOptIn.Level.WARNING, message = "This API is experimental. It can
be incompatibly changed in the future.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime
```

If you publish multiple independent features that require opt-in, declare an annotation for each. This makes the use of API safer for your clients: they can use only the features that they explicitly accept. This also lets you remove the opt-in

requirements from the features independently.

Mark API elements

To require an opt-in to using an API element, annotate its declaration with an opt-in requirement annotation:

```
@MyDateTime  
class DateProvider  
  
@MyDateTime  
fun getTime(): Time {}
```

Opt-in requirements for pre-stable APIs

If you use opt-in requirements for features that are not stable yet, carefully handle the API graduation to avoid breaking the client code.

Once your pre-stable API graduates and is released in a stable state, remove its opt-in requirement annotations from declarations. The clients will be able to use them without restriction. However, you should leave the annotation classes in modules so that the existing client code remains compatible.

To let the API users update their modules accordingly (remove the annotations from their code and recompile), mark the annotations as `@Deprecated` and provide the explanation in the deprecation message.

```
@Deprecated("This opt-in requirement is not used anymore. Remove its usages from your code.")  
@RequiresOptIn  
annotation class ExperimentalDateTime
```

Experimental status of the opt-in requirements

The opt-in requirement mechanism is currently [experimental](#). This means that in future releases it may be changed in ways that make it incompatible.

To make the users of annotations `@OptIn` and `@RequiresOptIn` aware of their experimental status, the compiler raises warnings when compiling the code with these annotations:

This class can only be used with the compiler argument '`-Xopt-in=kotlin.RequiresOptIn`'

To remove the warnings, add the compiler argument `-Xopt-in=kotlin.RequiresOptIn`.

Coroutines guide

Kotlin, as a language, provides only minimal low-level APIs in its standard library to enable various other libraries to

utilize coroutines. Unlike many other languages with similar capabilities, `async` and `await` are not keywords in Kotlin and are not even part of its standard library. Moreover, Kotlin's concept of suspending function provides a safer and less error-prone abstraction for asynchronous operations than futures and promises.

`kotlinx.coroutines` is a rich library for coroutines developed by JetBrains. It contains a number of high-level coroutine-enabled primitives that this guide covers, including `launch`, `async` and others.

This is a guide on core features of `kotlinx.coroutines` with a series of examples, divided up into different topics.

In order to use coroutines as well as follow the examples in this guide, you need to add a dependency on the `kotlinx-coroutines-core` module as explained [in the project README](#).

Table of contents

- [Coroutines basics](#)
- [Hands-on: Intro to coroutines and channels](#)
- [Cancellation and timeouts](#)
- [Composing suspending functions](#)
- [Coroutine context and dispatchers](#)
- [Asynchronous Flow](#)
- [Channels](#)
- [Coroutine exceptions handling](#)
- [Shared mutable state and concurrency](#)
- [Select expression \(experimental\)](#)
- [Tutorial: Debug coroutines using IntelliJ IDEA](#)
- [Tutorial: Debug Kotlin Flow using IntelliJ IDEA](#)

Additional references

- [Guide to UI programming with coroutines](#)
- [Coroutines design document \(KEEP\)](#)
- [Full `kotlinx.coroutines` API reference](#)

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, 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 it, 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 regarding Coroutines and the different possibilities, check out the reference guide.

Coroutines basics

This section covers basic coroutine concepts.

Your first coroutine

A coroutine is an instance of suspendable computation. It is conceptually similar to a thread, in the sense that it takes

a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one.

Coroutines can be thought of as light-weight threads, but there is a number of important differences that make their real-life usage very different from threads.

Run the following code to get to your first working coroutine:

```
import kotlinx.coroutines.*  
  
//sampleStart  
fun main() = runBlocking { // this: CoroutineScope  
    launch { // launch a new coroutine and continue  
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)  
        println("World!") // print after delay  
    }  
    println("Hello") // main coroutine continues while a previous one is delayed  
}  
//sampleEnd
```

You can get the full code [here](#).

You will see the following result:

```
Hello  
World!
```

Let's dissect what this code does.

`launch` is a coroutine builder. It launches a new coroutine concurrently with the rest of the code, which continues to work independently. That's why Hello has been printed first.

`delay` is a special suspending function. It suspends the coroutine for a specific time. Suspending a coroutine does not block the underlying thread, but allows other coroutines to run and use the underlying thread for their code.

`runBlocking` is also a coroutine builder that bridges the non-coroutine world of a regular `fun main()` and the code with coroutines inside of `runBlocking { ... }` curly braces. This is highlighted in an IDE by this: `CoroutineScope` hint right after the `runBlocking` opening curly brace.

If you remove or forget `runBlocking` in this code, you'll get an error on the `launch` call, since `launch` is declared only in the `CoroutineScope`:

```
Unresolved reference: launch
```

The name of `runBlocking` means that the thread that runs it (in this case — the main thread) gets blocked for the duration of the call, until all the coroutines inside `runBlocking { ... }` complete their execution. You will often see

`runBlocking` used like that at the very top-level of the application and quite rarely inside the real code, as threads are expensive resources and blocking them is inefficient and is often not desired.

Structured concurrency

Coroutines follow a principle of structured concurrency which means that new coroutines can be only launched in a specific `CoroutineScope` which delimits the lifetime of the coroutine. The above example shows that `runBlocking` establishes the corresponding scope and that is why the previous example waits until `World!` is printed after a second's delay and only then exits.

In the real application, you will be launching a lot of coroutines. Structured concurrency ensures that they are not lost and do not leak. An outer scope cannot complete until all its children coroutines complete. Structured concurrency also ensures that any errors in the code are properly reported and are never lost.

Extract function refactoring

Let's extract the block of code inside `launch { ... }` into a separate function. When you perform "Extract function" refactoring on this code, you get a new function with the `suspend` modifier. This is your first suspending function. Suspending functions can be used inside coroutines just like regular functions, but their additional feature is that they can, in turn, use other suspending functions (like `delay` in this example) to suspend execution of a coroutine.

```
import kotlinx.coroutines.*  
  
//sampleStart  
fun main() = runBlocking { // this: CoroutineScope  
    launch { doWorld() }  
    println("Hello")  
}  
  
// this is your first suspending function  
suspend fun doWorld() {  
    delay(1000L)  
    println("World!")  
}  
//sampleEnd
```

You can get the full code [here](#).

Scope builder

In addition to the coroutine scope provided by different builders, it is possible to declare your own scope using the `CoroutineScope` builder. It creates a coroutine scope and does not complete until all launched children complete.

`runBlocking` and `CoroutineScope` builders may look similar because they both wait for their body and all its children to complete. The main difference is that the `runBlocking` method blocks the current thread for waiting, while

`coroutineScope` just suspends, releasing the underlying thread for other usages. Because of that difference, `runBlocking` is a regular function and `coroutineScope` is a suspending function.

You can use `coroutineScope` from any suspending function. For example, you can move the concurrent printing of Hello and World into a suspend fun `doWorld()` function:

```
import kotlinx.coroutines.*  
  
//sampleStart  
fun main() = runBlocking {  
    doWorld()  
}  
  
suspend fun doWorld() = coroutineScope { // this: CoroutineScope  
    launch {  
        delay(1000L)  
        println("World!")  
    }  
    println("Hello")  
}  
//sampleEnd
```

You can get the full code [here](#).

This code also prints:

```
Hello  
World!
```

Scope builder and concurrency

A `coroutineScope` builder can be used inside any suspending function to perform multiple concurrent operations. Let's launch two concurrent coroutines inside a `doWorld` suspending function:

```
import kotlinx.coroutines.*  
  
//sampleStart  
// Sequentially executes doWorld followed by "Done"  
fun main() = runBlocking {  
    doWorld()  
    println("Done")  
}  
  
// Concurrently executes both sections  
suspend fun doWorld() = coroutineScope { // this: CoroutineScope  
    launch {  
        delay(2000L)  
        println("World 2")  
    }  
}
```

```
    }
    launch {
        delay(1000L)
        println("World 1")
    }
    println("Hello")
}
//sampleEnd
```

You can get the full code [here](#).

Both pieces of code inside launch { ... } blocks execute concurrently, with World 1 printed first, after a second from start, and World 2 printed next, after two seconds from start. A `coroutineScope` in doWorld completes only after both are complete, so doWorld returns and allows Done string to be printed only after that:

```
Hello
World 1
World 2
Done
```

An explicit job

A `launch` coroutine builder returns a `Job` object that is a handle to the launched coroutine and can be used to explicitly wait for its completion. For example, you can wait for completion of the child coroutine and then print "Done" string:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch { // launch a new coroutine and keep a reference to its Job
        delay(1000L)
        println("World!")
    }
    println("Hello")
    job.join() // wait until child coroutine completes
    println("Done")
}
//sampleEnd
}
```

You can get the full code [here](#).

This code produces:

```
Hello
```

World!
Done

Coroutines ARE light-weight

Run the following code:

```
import kotlinx.coroutines.*  
  
//sampleStart  
fun main() = runBlocking {  
    repeat(100_000) { // launch a lot of coroutines  
        launch {  
            delay(5000L)  
            print(".")  
        }  
    }  
}  
//sampleEnd
```

You can get the full code [here](#).

It launches 100K coroutines and, after 5 seconds, each coroutine prints a dot.

Now, try that with threads (remove runBlocking, replace launch with thread, and replace delay with Thread.sleep). What would happen? (Most likely your code will produce some sort of out-of-memory error)

Cancellation and timeouts

This section covers coroutine cancellation and timeouts.

Cancelling coroutine execution

In a long-running application you might need fine-grained control on your background coroutines. For example, a user might have closed the page that launched a coroutine and now its result is no longer needed and its operation can be cancelled. The `launch` function returns a `Job` that can be used to cancel the running coroutine:

```
import kotlinx.coroutines.*  
  
fun main() = runBlocking {  
    //sampleStart  
    val job = launch {  
        repeat(1000) { i ->
```

```

        println("job: I'm sleeping $i ...")
        delay(500L)
    }
}

delay(1300L) // delay a bit
println("main: I'm tired of waiting!")
job.cancel() // cancels the job
job.join() // waits for job's completion
println("main: Now I can quit.")
//sampleEnd
}

```

You can get the full code [here](#).

It produces the following output:

```

job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
main: Now I can quit.

```

As soon as main invokes job.cancel, we don't see any output from the other coroutine because it was cancelled. There is also a `Job` extension function `cancelAndJoin` that combines `cancel` and `join` invocations.

Cancellation is cooperative

Coroutine cancellation is cooperative. A coroutine code has to cooperate to be cancellable. All the suspending functions in `kotlinx.coroutines` are cancellable. They check for cancellation of coroutine and throw `CancellationException` when cancelled. However, if a coroutine is working in a computation and does not check for cancellation, then it cannot be cancelled, like the following example shows:

```

import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val startTime = System.currentTimeMillis()
    val job = launch(Dispatchers.Default) {
        var nextPrintTime = startTime
        var i = 0
        while (i < 5) { // computation loop, just wastes CPU
            // print a message twice a second
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("job: I'm sleeping ${i++} ...")
                nextPrintTime += 500L
            }
        }
    }
    delay(1300L) // delay a bit
}

```

```

    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // cancels the job and waits for its completion
    println("main: Now I can quit.")
//sampleEnd
}

```

You can get the full code [here](#).

Run it to see that it continues to print "I'm sleeping" even after cancellation until the job completes by itself after five iterations.

Making computation code cancellable

There are two approaches to making computation code cancellable. The first one is to periodically invoke a suspending function that checks for cancellation. There is a [yield](#) function that is a good choice for that purpose. The other one is to explicitly check the cancellation status. Let us try the latter approach.

Replace while (*i* < 5) in the previous example with while (*isActive*) and rerun it.

```

import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val startTime = System.currentTimeMillis()
    val job = launch(Dispatchers.Default) {
        var nextPrintTime = startTime
        var i = 0
        while (isActive) { // cancellable computation loop
            // print a message twice a second
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("job: I'm sleeping ${i++} ...")
                nextPrintTime += 500L
            }
        }
    }
    delay(1300L) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // cancels the job and waits for its completion
    println("main: Now I can quit.")
//sampleEnd
}

```

You can get the full code [here](#).

As you can see, now this loop is cancelled. [isActive](#) is an extension property available inside the coroutine via the [CoroutineScope](#) object.

Closing resources with finally

Cancellable suspending functions throw [CancellationException](#) on cancellation which can be handled in the usual way. For example, try {...} finally {...} expression and Kotlin use function execute their finalization actions normally when a coroutine is cancelled:

```
import kotlinx.coroutines.*  
  
fun main() = runBlocking {  
    //sampleStart  
    val job = launch {  
        try {  
            repeat(1000) { i ->  
                println("job: I'm sleeping $i ...")  
                delay(500L)  
            }  
        } finally {  
            println("job: I'm running finally")  
        }  
    }  
    delay(1300L) // delay a bit  
    println("main: I'm tired of waiting!")  
    job.cancelAndJoin() // cancels the job and waits for its completion  
    println("main: Now I can quit.")  
    //sampleEnd  
}
```

You can get the full code [here](#).

Both `join` and `cancelAndJoin` wait for all finalization actions to complete, so the example above produces the following output:

```
job: I'm sleeping 0 ...  
job: I'm sleeping 1 ...  
job: I'm sleeping 2 ...  
main: I'm tired of waiting!  
job: I'm running finally  
main: Now I can quit.
```

Run non-cancellable block

Any attempt to use a suspending function in the finally block of the previous example causes [CancellationException](#), because the coroutine running this code is cancelled. Usually, this is not a problem, since all well-behaving closing operations (closing a file, cancelling a job, or closing any kind of a communication channel) are usually non-blocking and do not involve any suspending functions. However, in the rare case when you need to suspend in a cancelled coroutine you can wrap the corresponding code in `withContext(NonCancellable) {...}` using [withContext](#) function and [NonCancellable](#) context as the following example shows:

```

import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch {
        try {
            repeat(1000) { i ->
                println("job: I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            withContext(NonCancellable) {
                println("job: I'm running finally")
                delay(1000L)
                println("job: And I've just delayed for 1 sec because I'm non-cancellable")
            }
        }
    }
    delay(1300L) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // cancels the job and waits for its completion
    println("main: Now I can quit.")
//sampleEnd
}

```

You can get the full code [here](#).

Timeout

The most obvious practical reason to cancel execution of a coroutine is because its execution time has exceeded some timeout. While you can manually track the reference to the corresponding `Job` and launch a separate coroutine to cancel the tracked one after delay, there is a ready to use `withTimeout` function that does it. Look at the following example:

```

import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    withTimeout(1300L) {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    }
//sampleEnd
}

```

You can get the full code [here](#).

It produces the following output:

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
Exception in thread "main" kotlinx.coroutines.TimeoutCancellationException: Timed out waiting
for 1300 ms
```

The `TimeoutCancellationException` that is thrown by `withTimeout` is a subclass of `CancellationException`. We have not seen its stack trace printed on the console before. That is because inside a cancelled coroutine `CancellationException` is considered to be a normal reason for coroutine completion. However, in this example we have used `withTimeout` right inside the main function.

Since cancellation is just an exception, all resources are closed in the usual way. You can wrap the code with timeout in a `try {...} catch (e: TimeoutCancellationException) {...}` block if you need to do some additional action specifically on any kind of timeout or use the `withTimeoutOrNull` function that is similar to `withTimeout` but returns null on timeout instead of throwing an exception:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val result = withTimeoutOrNull(1300L) {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
        "Done" // will get cancelled before it produces this result
    }
    println("Result is $result")
//sampleEnd
}
```

You can get the full code [here](#).

There is no longer an exception when running this code:

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
Result is null
```

Asynchronous timeout and resources

The timeout event in `withTimeout` is asynchronous with respect to the code running in its block and may happen at any time, even right before the return from inside of the timeout block. Keep this in mind if you open or acquire some resource inside the block that needs closing or release outside of the block.

For example, here we imitate a closeable resource with the `Resource` class, that simply keeps track of how many times it was created by incrementing the acquired counter and decrementing this counter from its `close` function. Let us run a lot of coroutines with the small timeout try acquire this resource from inside of the `withTimeout` block after a bit of delay and release it from outside.

```
import kotlinx.coroutines.*  
  
//sampleStart  
var acquired = 0  
  
class Resource {  
    init { acquired++ } // Acquire the resource  
    fun close() { acquired-- } // Release the resource  
}  
  
fun main() {  
    runBlocking {  
        repeat(100_000) { // Launch 100K coroutines  
            launch {  
                val resource = withTimeout(60) { // Timeout of 60 ms  
                    delay(50) // Delay for 50 ms  
                    Resource() // Acquire a resource and return it from withTimeout block  
                }  
                resource.close() // Release the resource  
            }  
        }  
        // Outside of runBlocking all coroutines have completed  
        println(acquired) // Print the number of resources still acquired  
    }  
    //sampleEnd
```

You can get the full code [here](#).

If you run the above code you'll see that it does not always print zero, though it may depend on the timings of your machine you may need to tweak timeouts in this example to actually see non-zero values.

Note, that incrementing and decrementing acquired counter here from 100K coroutines is completely safe, since it always happens from the same main thread. More on that will be explained in the next chapter on coroutine context.

To workaround this problem you can store a reference to the resource in the variable as opposed to returning it from the `withTimeout` block.

```
import kotlinx.coroutines.*  
  
var acquired = 0  
  
class Resource {  
    init { acquired++ } // Acquire the resource  
    fun close() { acquired-- } // Release the resource  
}  
  
fun main() {  
    //sampleStart  
    runBlocking {  
        repeat(100_000) { // Launch 100K coroutines  
            launch {  
                var resource: Resource? = null // Not acquired yet  
                try {  
                    withTimeout(60) { // Timeout of 60 ms  
                        delay(50) // Delay for 50 ms  
                        resource = Resource() // Store a resource to the variable if acquired  
                    }  
                    // We can do something else with the resource here  
                } finally {  
                    resource?.close() // Release the resource if it was acquired  
                }  
            }  
        }  
        // Outside of runBlocking all coroutines have completed  
        println(acquired) // Print the number of resources still acquired  
    }  
    //sampleEnd  
}
```

You can get the full code [here](#).

This example always prints zero. Resources do not leak.

Composing suspending functions

This section covers various approaches to composition of suspending functions.

Sequential by default

Assume that we have two suspending functions defined elsewhere that do something useful like some kind of remote service call or computation. We just pretend they are useful, but actually each one just delays for a second for the

purpose of this example:

```
suspend fun doSomethingUsefulOne(): Int {  
    delay(1000L) // pretend we are doing something useful here  
    return 13  
}  
  
suspend fun doSomethingUsefulTwo(): Int {  
    delay(1000L) // pretend we are doing something useful here, too  
    return 29  
}
```

What do we do if we need them to be invoked sequentially — first `doSomethingUsefulOne` and then `doSomethingUsefulTwo`, and compute the sum of their results? In practice, we do this if we use the result of the first function to make a decision on whether we need to invoke the second one or to decide on how to invoke it.

We use a normal sequential invocation, because the code in the coroutine, just like in the regular code, is sequential by default. The following example demonstrates it by measuring the total time it takes to execute both suspending functions:

```
import kotlinx.coroutines.*  
import kotlin.system.*  
  
fun main() = runBlocking<Unit> {  
    //sampleStart  
    val time = measureTimeMillis {  
        val one = doSomethingUsefulOne()  
        val two = doSomethingUsefulTwo()  
        println("The answer is ${one + two}")  
    }  
    println("Completed in $time ms")  
    //sampleEnd  
}  
  
suspend fun doSomethingUsefulOne(): Int {  
    delay(1000L) // pretend we are doing something useful here  
    return 13  
}  
  
suspend fun doSomethingUsefulTwo(): Int {  
    delay(1000L) // pretend we are doing something useful here, too  
    return 29  
}
```

You can get the full code [here](#).

It produces something like this:

```
The answer is 42  
Completed in 2017 ms
```

Concurrent using `async`

What if there are no dependencies between invocations of `doSomethingUsefulOne` and `doSomethingUsefulTwo` and we want to get the answer faster, by doing both concurrently? This is where `async` comes to help.

Conceptually, `async` is just like `launch`. It starts a separate coroutine which is a light-weight thread that works concurrently with all the other coroutines. The difference is that `launch` returns a `Job` and does not carry any resulting value, while `async` returns a `Deferred` — a light-weight non-blocking future that represents a promise to provide a result later. You can use `.await()` on a deferred value to get its eventual result, but `Deferred` is also a `Job`, so you can cancel it if needed.

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
    //sampleStart
    val time = measureTimeMillis {
        val one = async { doSomethingUsefulOne() }
        val two = async { doSomethingUsefulTwo() }
        println("The answer is ${one.await() + two.await()}")
    }
    println("Completed in $time ms")
    //sampleEnd
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

You can get the full code [here](#).

It produces something like this:

```
The answer is 42
Completed in 1017 ms
```

This is twice as fast, because the two coroutines execute concurrently. Note that concurrency with coroutines is always explicit.

Lazily started async

Optionally, `async` can be made lazy by setting its start parameter to `CoroutineStart.LAZY`. In this mode it only starts the coroutine when its result is required by `await`, or if its Job's `start` function is invoked. Run the following example:

```
import kotlinx.coroutines.*  
import kotlin.system.*  
  
fun main() = runBlocking<Unit> {  
    //sampleStart  
    val time = measureTimeMillis {  
        val one = async(start = CoroutineStart.LAZY) { doSomethingUsefulOne() }  
        val two = async(start = CoroutineStart.LAZY) { doSomethingUsefulTwo() }  
        // some computation  
        one.start() // start the first one  
        two.start() // start the second one  
        println("The answer is ${one.await() + two.await()}")  
    }  
    println("Completed in $time ms")  
    //sampleEnd  
}  
  
suspend fun doSomethingUsefulOne(): Int {  
    delay(1000L) // pretend we are doing something useful here  
    return 13  
}  
  
suspend fun doSomethingUsefulTwo(): Int {  
    delay(1000L) // pretend we are doing something useful here, too  
    return 29  
}
```

You can get the full code [here](#).

It produces something like this:

```
The answer is 42  
Completed in 1017 ms
```

So, here the two coroutines are defined but not executed as in the previous example, but the control is given to the programmer on when exactly to start the execution by calling `start`. We first start one, then start two, and then await for the individual coroutines to finish.

Note that if we just call `await` in `println` without first calling `start` on individual coroutines, this will lead to sequential behavior, since `await` starts the coroutine execution and waits for its finish, which is not the intended use-case for laziness. The use-case for `async(start = CoroutineStart.LAZY)` is a replacement for the standard lazy function in cases when computation of the value involves suspending functions.

Async-style functions

We can define async-style functions that invoke `doSomethingUsefulOne` and `doSomethingUsefulTwo` asynchronously using the `async` coroutine builder using a `GlobalScope` reference to opt-out of the structured concurrency. We name such functions with the "...Async" suffix to highlight the fact that they only start asynchronous computation and one needs to use the resulting deferred value to get the result.

`GlobalScope` is a delicate API that can backfire in non-trivial ways, one of which will be explained below, so you must explicitly opt-in into using `GlobalScope` with `@OptIn(DelicateCoroutinesApi::class)`.

```
// The result type of somethingUsefulOneAsync is Deferred<Int>
@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
}

// The result type of somethingUsefulTwoAsync is Deferred<Int>
@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulTwoAsync() = GlobalScope.async {
    doSomethingUsefulTwo()
}
```

Note that these `xxxAsync` functions are not suspending functions. They can be used from anywhere. However, their use always implies asynchronous (here meaning concurrent) execution of their action with the invoking code.

The following example shows their use outside of coroutine:

```
import kotlinx.coroutines.*
import kotlin.system.*

//sampleStart
// note that we don't have `runBlocking` to the right of `main` in this example
fun main() {
    val time = measureTimeMillis {
        // we can initiate async actions outside of a coroutine
        val one = somethingUsefulOneAsync()
        val two = somethingUsefulTwoAsync()
        // but waiting for a result must involve either suspending or blocking.
        // here we use `runBlocking { ... }` to block the main thread while waiting for the
        // result
        runBlocking {
            println("The answer is ${one.await() + two.await()}")
        }
    }
    println("Completed in $time ms")
}
//sampleEnd

@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
```

```

}

@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulTwoAsync() = GlobalScope.async {
    doSomethingUsefulTwo()
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}

```

You can get the full code [here](#).

This programming style with `async` functions is provided here only for illustration, because it is a popular style in other programming languages. Using this style with Kotlin coroutines is strongly discouraged for the reasons explained below.

Consider what happens if between the `val one = somethingUsefulOneAsync()` line and `one.await()` expression there is some logic error in the code, and the program throws an exception, and the operation that was being performed by the program aborts. Normally, a global error-handler could catch this exception, log and report the error for developers, but the program could otherwise continue doing other operations. However, here we have `somethingUsefulOneAsync` still running in the background, even though the operation that initiated it was aborted. This problem does not happen with structured concurrency, as shown in the section below.

Structured concurrency with `async`

Let us take the [Concurrent using `async`](#) example and extract a function that concurrently performs `doSomethingUsefulOne` and `doSomethingUsefulTwo` and returns the sum of their results. Because the `async` coroutine builder is defined as an extension on `CoroutineScope`, we need to have it in the scope and that is what the `coroutineScope` function provides:

```

suspend fun concurrentSum(): Int = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    one.await() + two.await()
}

```

This way, if something goes wrong inside the code of the `concurrentSum` function, and it throws an exception, all the coroutines that were launched in its scope will be cancelled.

```

import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        println("The answer is ${concurrentSum()}")
    }
    println("Completed in $time ms")
//sampleEnd
}

suspend fun concurrentSum(): Int = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    one.await() + two.await()
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}

```

You can get the full code [here](#).

We still have concurrent execution of both operations, as evident from the output of the above main function:

```

The answer is 42
Completed in 1017 ms

```

Cancellation is always propagated through coroutines hierarchy:

```

import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    try {
        failedConcurrentSum()
    } catch(e: ArithmeticException) {
        println("Computation failed with ArithmeticException")
    }
}

suspend fun failedConcurrentSum(): Int = coroutineScope {
    val one = async<Int> {
        try {
            delay(Long.MAX_VALUE) // Emulates very long computation
        } catch(e: CancellationException) {
            println("Computation was cancelled")
        }
        return@async 42
    }
    one.await()
}

```

```

    42
    } finally {
        println("First child was cancelled")
    }
}
val two = async<Int> {
    println("Second child throws an exception")
    throw ArithmeticException()
}
one.await() + two.await()
}

```

You can get the full code [here](#).

Note how both the first `async` and the awaiting parent are cancelled on failure of one of the children (namely, `two`):

```

Second child throws an exception
First child was cancelled
Computation failed with ArithmeticException

```

Coroutine context and dispatchers

Coroutines always execute in some context represented by a value of the [CoroutineContext](#) type, defined in the Kotlin standard library.

The coroutine context is a set of various elements. The main elements are the [Job](#) of the coroutine, which we've seen before, and its dispatcher, which is covered in this section.

Dispatchers and threads

The coroutine context includes a coroutine dispatcher ([see CoroutineDispatcher](#)) that determines what thread or threads the corresponding coroutine uses for its execution. The coroutine dispatcher can confine coroutine execution to a specific thread, dispatch it to a thread pool, or let it run unconfined.

All coroutine builders like `launch` and `async` accept an optional [CoroutineContext](#) parameter that can be used to explicitly specify the dispatcher for the new coroutine and other context elements.

Try the following example:

```

import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    launch { // context of the parent, main runBlocking coroutine

```

```

        println("main runBlocking      : I'm working in thread
${Thread.currentThread().name}")
    }
    launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
        println("Unconfined          : I'm working in thread
${Thread.currentThread().name}")
    }
    launch(Dispatchers.Default) { // will get dispatched to DefaultDispatcher
        println("Default            : I'm working in thread
${Thread.currentThread().name}")
    }
    launch(newSingleThreadContext("MyOwnThread")) { // will get its own new thread
        println("newSingleThreadContext: I'm working in thread ${Thread.currentThread().name}")
    }
}
//sampleEnd
}

```

You can get the full code [here](#).

It produces the following output (maybe in different order):

```

Unconfined      : I'm working in thread main
Default         : I'm working in thread DefaultDispatcher-worker-1
newSingleThreadContext: I'm working in thread MyOwnThread
main runBlocking   : I'm working in thread main

```

When `launch { ... }` is used without parameters, it inherits the context (and thus dispatcher) from the [CoroutineScope](#) it is being launched from. In this case, it inherits the context of the main `runBlocking` coroutine which runs in the main thread.

[Dispatchers.Unconfined](#) is a special dispatcher that also appears to run in the main thread, but it is, in fact, a different mechanism that is explained later.

The default dispatcher that is used when no other dispatcher is explicitly specified in the scope. It is represented by [Dispatchers.Default](#) and uses a shared background pool of threads.

[newSingleThreadContext](#) creates a thread for the coroutine to run. A dedicated thread is a very expensive resource. In a real application it must be either released, when no longer needed, using the [close](#) function, or stored in a top-level variable and reused throughout the application.

Unconfined vs confined dispatcher

The [Dispatchers.Unconfined](#) coroutine dispatcher starts a coroutine in the caller thread, but only until the first suspension point. After suspension it resumes the coroutine in the thread that is fully determined by the suspending function that was invoked. The unconfined dispatcher is appropriate for coroutines which neither consume CPU time nor update any shared data (like UI) confined to a specific thread.

On the other side, the dispatcher is inherited from the outer `CoroutineScope` by default. The default dispatcher for the `runBlocking` coroutine, in particular, is confined to the invoker thread, so inheriting it has the effect of confining execution to this thread with predictable FIFO scheduling.

```
import kotlinx.coroutines.*  
  
fun main() = runBlocking<Unit> {  
    //sampleStart  
    launch(Dispatchers.Unconfined) { // not confined -- will work with main thread  
        println("Unconfined      : I'm working in thread ${Thread.currentThread().name}")  
        delay(500)  
        println("Unconfined      : After delay in thread ${Thread.currentThread().name}")  
    }  
    launch { // context of the parent, main runBlocking coroutine  
        println("main runBlocking: I'm working in thread ${Thread.currentThread().name}")  
        delay(1000)  
        println("main runBlocking: After delay in thread ${Thread.currentThread().name}")  
    }  
    //sampleEnd  
}
```

You can get the full code [here](#).

Produces the output:

```
Unconfined      : I'm working in thread main  
main runBlocking: I'm working in thread main  
Unconfined      : After delay in thread kotlinx.coroutines.DefaultExecutor  
main runBlocking: After delay in thread main
```

So, the coroutine with the context inherited from `runBlocking {...}` continues to execute in the main thread, while the unconfined one resumes in the default executor thread that the `delay` function is using.

The unconfined dispatcher is an advanced mechanism that can be helpful in certain corner cases where dispatching of a coroutine for its execution later is not needed or produces undesirable side-effects, because some operation in a coroutine must be performed right away. The unconfined dispatcher should not be used in general code.

Debugging coroutines and threads

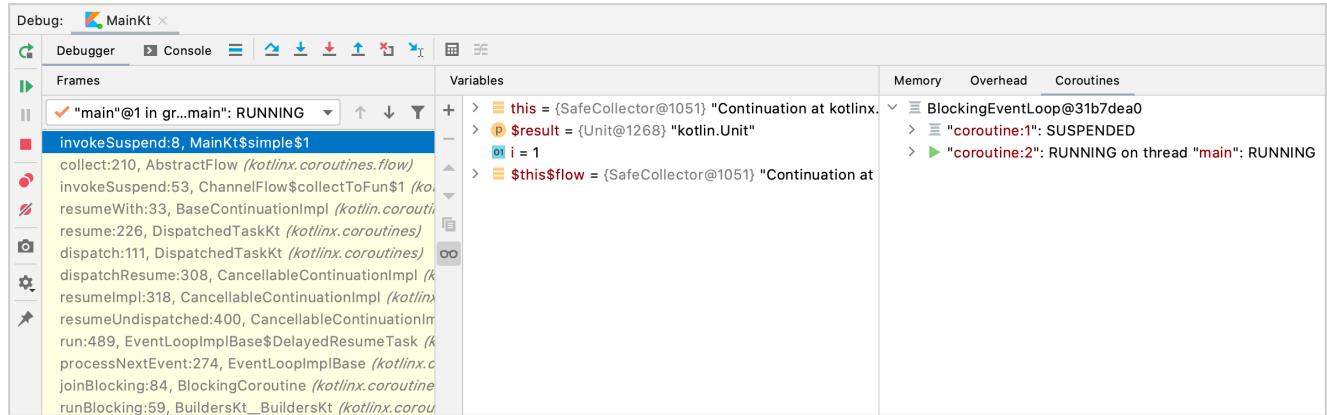
Coroutines can suspend on one thread and resume on another thread. Even with a single-threaded dispatcher it might be hard to figure out what the coroutine was doing, where, and when if you don't have special tooling.

Debugging with IDEA

The Coroutine Debugger of the Kotlin plugin simplifies debugging coroutines in IntelliJ IDEA.

Debugging works for versions 1.3.8 or later of kotlinx-coroutines-core.

The Debug tool window contains the Coroutines tab. In this tab, you can find information about both currently running and suspended coroutines. The coroutines are grouped by the dispatcher they are running on.



Debugging coroutines

With the coroutine debugger, you can:

- Check the state of each coroutine.
- See the values of local and captured variables for both running and suspended coroutines.
- See a full coroutine creation stack, as well as a call stack inside the coroutine. The stack includes all frames with variable values, even those that would be lost during standard debugging.
- Get a full report that contains the state of each coroutine and its stack. To obtain it, right-click inside the Coroutines tab, and then click Get Coroutines Dump.

To start coroutine debugging, you just need to set breakpoints and run the application in debug mode.

Learn more about coroutines debugging in the [tutorial](#).

Debugging using logging

Another approach to debugging applications with threads without Coroutine Debugger is to print the thread name in the log file on each log statement. This feature is universally supported by logging frameworks. When using coroutines, the thread name alone does not give much of a context, so kotlinx.coroutines includes debugging facilities to make it easier.

Run the following code with -Dkotlinx.coroutines.debug JVM option:

```

import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() = runBlocking<Unit> {
//sampleStart
    val a = async {
        log("I'm computing a piece of the answer")
        6
    }
    val b = async {
        log("I'm computing another piece of the answer")
        7
    }
    log("The answer is ${a.await() * b.await()}")
//sampleEnd
}

```

You can get the full code [here](#).

There are three coroutines. The main coroutine (#1) inside `runBlocking` and two coroutines computing the deferred values a (#2) and b (#3). They are all executing in the context of `runBlocking` and are confined to the main thread. The output of this code is:

```

[main @coroutine#2] I'm computing a piece of the answer
[main @coroutine#3] I'm computing another piece of the answer
[main @coroutine#1] The answer is 42

```

The `log` function prints the name of the thread in square brackets, and you can see that it is the main thread with the identifier of the currently executing coroutine appended to it. This identifier is consecutively assigned to all created coroutines when the debugging mode is on.

Debugging mode is also turned on when JVM is run with `-ea` option. You can read more about debugging facilities in the documentation of the [DEBUG_PROPERTY_NAME](#) property.

Jumping between threads

Run the following code with the `-Dkotlinx.coroutines.debug` JVM option (see [debug](#)):

```

import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() {
//sampleStart

```

```

newSingleThreadContext("Ctx1").use { ctx1 ->
    newSingleThreadContext("Ctx2").use { ctx2 ->
        runBlocking(ctx1) {
            log("Started in ctx1")
            withContext(ctx2) {
                log("Working in ctx2")
            }
            log("Back to ctx1")
        }
    }
}
//sampleEnd
}

```

You can get the full code [here](#).

It demonstrates several new techniques. One is using `runBlocking` with an explicitly specified context, and the other one is using the `withContext` function to change the context of a coroutine while still staying in the same coroutine, as you can see in the output below:

```
[Ctx1 @coroutine#1] Started in ctx1
[Ctx2 @coroutine#1] Working in ctx2
[Ctx1 @coroutine#1] Back to ctx1
```

Note that this example also uses the `use` function from the Kotlin standard library to release threads created with `newSingleThreadContext` when they are no longer needed.

Job in the context

The coroutine's `Job` is part of its context, and can be retrieved from it using `coroutineContext[Job]` expression:

```

import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    println("My job is ${coroutineContext[Job]}")
//sampleEnd
}

```

You can get the full code [here](#).

In the `debug mode`, it outputs something like this:

My job is "coroutine#1":BlockingCoroutine{Active}@6d311334

Note that `isActive` in `CoroutineScope` is just a convenient shortcut for `coroutineContext[Job]?.isActive == true`.

Children of a coroutine

When a coroutine is launched in the [CoroutineScope](#) of another coroutine, it inherits its context via [CoroutineScope.coroutineContext](#) and the [Job](#) of the new coroutine becomes a child of the parent coroutine's job. When the parent coroutine is cancelled, all its children are recursively cancelled, too.

However, this parent-child relation can be explicitly overridden in one of two ways:

1. When a different scope is explicitly specified when launching a coroutine (for example, `GlobalScope.launch`), then it does not inherit a Job from the parent scope.
2. When a different Job object is passed as the context for the new coroutine (as shown in the example below), then it overrides the Job of the parent scope.

In both cases, the launched coroutine is not tied to the scope it was launched from and operates independently.

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    //sampleStart
    // launch a coroutine to process some kind of incoming request
    val request = launch {
        // it spawns two other jobs
        launch(Job()) {
            println("job1: I run in my own Job and execute independently!")
            delay(1000)
            println("job1: I am not affected by cancellation of the request")
        }
        // and the other inherits the parent context
        launch {
            delay(100)
            println("job2: I am a child of the request coroutine")
            delay(1000)
            println("job2: I will not execute this line if my parent request is cancelled")
        }
    }
    delay(500)
    request.cancel() // cancel processing of the request
    delay(1000) // delay a second to see what happens
    println("main: Who has survived request cancellation?")
    //sampleEnd
}
```

You can get the full code [here](#).

The output of this code is:

```
job1: I run in my own Job and execute independently!
```

```
job2: I am a child of the request coroutine
job1: I am not affected by cancellation of the request
main: Who has survived request cancellation?
```

Parental responsibilities

A parent coroutine always waits for completion of all its children. A parent does not have to explicitly track all the children it launches, and it does not have to use `Job.join` to wait for them at the end:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    // launch a coroutine to process some kind of incoming request
    val request = launch {
        repeat(3) { i -> // launch a few children jobs
            launch {
                delay((i + 1) * 200L) // variable delay 200ms, 400ms, 600ms
                println("Coroutine $i is done")
            }
        }
        println("request: I'm done and I don't explicitly join my children that are still active")
    }
    request.join() // wait for completion of the request, including all its children
    println("Now processing of the request is complete")
//sampleEnd
}
```

You can get the full code [here](#).

The result is going to be:

```
request: I'm done and I don't explicitly join my children that are still active
Coroutine 0 is done
Coroutine 1 is done
Coroutine 2 is done
Now processing of the request is complete
```

Naming coroutines for debugging

Automatically assigned ids are good when coroutines log often and you just need to correlate log records coming from the same coroutine. However, when a coroutine is tied to the processing of a specific request or doing some specific background task, it is better to name it explicitly for debugging purposes. The `CoroutineName` context element serves the same purpose as the thread name. It is included in the thread name that is executing this coroutine when the

debugging mode is turned on.

The following example demonstrates this concept:

```
import kotlinx.coroutines.*  
  
fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")  
  
fun main() = runBlocking(CoroutineName("main")) {  
    //sampleStart  
    log("Started main coroutine")  
    // run two background value computations  
    val v1 = async(CoroutineName("v1coroutine")) {  
        delay(500)  
        log("Computing v1")  
        252  
    }  
    val v2 = async(CoroutineName("v2coroutine")) {  
        delay(1000)  
        log("Computing v2")  
        6  
    }  
    log("The answer for v1 / v2 = ${v1.await() / v2.await()}")  
    //sampleEnd  
}
```

You can get the full code [here](#).

The output it produces with -Dkotlinx.coroutines.debug JVM option is similar to:

```
[main @main#1] Started main coroutine  
[main @v1coroutine#2] Computing v1  
[main @v2coroutine#3] Computing v2  
[main @main#1] The answer for v1 / v2 = 42
```

Combining context elements

Sometimes we need to define multiple elements for a coroutine context. We can use the + operator for that. For example, we can launch a coroutine with an explicitly specified dispatcher and an explicitly specified name at the same time:

```
import kotlinx.coroutines.*  
  
fun main() = runBlocking<Unit> {  
    //sampleStart  
    launch(Dispatchers.Default + CoroutineName("test")) {  
        println("I'm working in thread ${Thread.currentThread().name}")  
    }  
}
```

```
//sampleEnd  
}
```

You can get the full code [here](#).

The output of this code with the `-Dkotlinx.coroutines.debug` JVM option is:

```
I'm working in thread DefaultDispatcher-worker-1 @test#2
```

Coroutine scope

Let us put our knowledge about contexts, children and jobs together. Assume that our application has an object with a lifecycle, but that object is not a coroutine. For example, we are writing an Android application and launch various coroutines in the context of an Android activity to perform asynchronous operations to fetch and update data, do animations, etc. All of these coroutines must be cancelled when the activity is destroyed to avoid memory leaks. We, of course, can manipulate contexts and jobs manually to tie the lifecycles of the activity and its coroutines, but `kotlinx.coroutines` provides an abstraction encapsulating that: `CoroutineScope`. You should be already familiar with the coroutine scope as all coroutine builders are declared as extensions on it.

We manage the lifecycles of our coroutines by creating an instance of `CoroutineScope` tied to the lifecycle of our activity. A `CoroutineScope` instance can be created by the `CoroutineScope()` or `MainScope()` factory functions. The former creates a general-purpose scope, while the latter creates a scope for UI applications and uses `Dispatchers.Main` as the default dispatcher:

```
class Activity {  
    private val mainScope = MainScope()  
  
    fun destroy() {  
        mainScope.cancel()  
    }  
    // to be continued ...
```

Now, we can launch coroutines in the scope of this `Activity` using the defined scope. For the demo, we launch ten coroutines that delay for a different time:

```
// class Activity continues  
fun doSomething() {  
    // launch ten coroutines for a demo, each working for a different time  
    repeat(10) { i ->  
        mainScope.launch {  
            delay((i + 1) * 200L) // variable delay 200ms, 400ms, ... etc  
            println("Coroutine $i is done")  
        }  
    }  
}
```

```
} // class Activity ends
```

In our main function we create the activity, call our test doSomething function, and destroy the activity after 500ms. This cancels all the coroutines that were launched from doSomething. We can see that because after the destruction of the activity no more messages are printed, even if we wait a little longer.

```
import kotlinx.coroutines.*  
  
class Activity {  
    private val mainScope = CoroutineScope(Dispatchers.Default) // use Default for test  
    purposes  
  
    fun destroy() {  
        mainScope.cancel()  
    }  
  
    fun doSomething() {  
        // launch ten coroutines for a demo, each working for a different time  
        repeat(10) { i ->  
            mainScope.launch {  
                delay((i + 1) * 200L) // variable delay 200ms, 400ms, ... etc  
                println("Coroutine $i is done")  
            }  
        }  
    }  
} // class Activity ends  
  
fun main() = runBlocking<Unit> {  
    //sampleStart  
    val activity = Activity()  
    activity.doSomething() // run test function  
    println("Launched coroutines")  
    delay(500L) // delay for half a second  
    println("Destroying activity!")  
    activity.destroy() // cancels all coroutines  
    delay(1000) // visually confirm that they don't work  
    //sampleEnd  
}
```

You can get the full code [here](#).

The output of this example is:

```
Launched coroutines  
Coroutine 0 is done  
Coroutine 1 is done  
Destroying activity!
```

As you can see, only the first two coroutines print a message and the others are cancelled by a single invocation of job.cancel() in Activity.destroy().

Note, that Android has first-party support for coroutine scope in all entities with the lifecycle. See [the corresponding documentation](#).

Thread-local data

Sometimes it is convenient to have an ability to pass some thread-local data to or between coroutines. However, since they are not bound to any particular thread, this will likely lead to boilerplate if done manually.

For [ThreadLocal](#), the [asContextElement](#) extension function is here for the rescue. It creates an additional context element which keeps the value of the given ThreadLocal and restores it every time the coroutine switches its context.

It is easy to demonstrate it in action:

```
import kotlinx.coroutines.*  
  
val threadLocal = ThreadLocal<String?>() // declare thread-local variable  
  
fun main() = runBlocking<Unit> {  
    //sampleStart  
    threadLocal.set("main")  
    println("Pre-main, current thread: ${Thread.currentThread()}, thread local value:  
        '${threadLocal.get()}'")  
    val job = launch(Dispatchers.Default + threadLocal.asContextElement(value = "launch")) {  
        println("Launch start, current thread: ${Thread.currentThread()}, thread local value:  
            '${threadLocal.get()}'")  
        yield()  
        println("After yield, current thread: ${Thread.currentThread()}, thread local value:  
            '${threadLocal.get()}'")  
    }  
    job.join()  
    println("Post-main, current thread: ${Thread.currentThread()}, thread local value:  
        '${threadLocal.get()}'")  
    //sampleEnd  
}
```

You can get the full code [here](#).

In this example we launch a new coroutine in a background thread pool using [Dispatchers.Default](#), so it works on a different thread from the thread pool, but it still has the value of the thread local variable that we specified using `threadLocal.asContextElement(value = "launch")`, no matter which thread the coroutine is executed on. Thus, the output (with [debug](#)) is:

```
Pre-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'main'  
Launch start, current thread: Thread[DefaultDispatcher-worker-1 @coroutine#2,5,main], thread  
local value: 'launch'  
After yield, current thread: Thread[DefaultDispatcher-worker-2 @coroutine#2,5,main], thread  
local value: 'launch'  
Post-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'main'
```

It's easy to forget to set the corresponding context element. The thread-local variable accessed from the coroutine may then have an unexpected value, if the thread running the coroutine is different. To avoid such situations, it is recommended to use the [ensurePresent](#) method and fail-fast on improper usages.

ThreadLocal has first-class support and can be used with any primitive kotlinx.coroutines provides. It has one key limitation, though: when a thread-local is mutated, a new value is not propagated to the coroutine caller (because a context element cannot track all ThreadLocal object accesses), and the updated value is lost on the next suspension. Use [withContext](#) to update the value of the thread-local in a coroutine, see [asContextElement](#) for more details.

Alternatively, a value can be stored in a mutable box like class Counter(var i: Int), which is, in turn, stored in a thread-local variable. However, in this case you are fully responsible to synchronize potentially concurrent modifications to the variable in this mutable box.

For advanced usage, for example for integration with logging MDC, transactional contexts or any other libraries which internally use thread-locals for passing data, see the documentation of the [ThreadContextElement](#) interface that should be implemented.

Asynchronous Flow

A suspending function asynchronously returns a single value, but how can we return multiple asynchronously computed values? This is where Kotlin Flows come in.

Representing multiple values

Multiple values can be represented in Kotlin using [collections](#). For example, we can have a simple function that returns a [List](#) of three numbers and then print them all using [forEach](#):

```
fun simple(): List<Int> = listOf(1, 2, 3)

fun main() {
    simple().forEach { value -> println(value) }
}
```

You can get the full code from [here](#).

This code outputs:

```
1
2
3
```

Sequences

If we are computing the numbers with some CPU-consuming blocking code (each computation taking 100ms), then we can represent the numbers using a Sequence:

```
fun simple(): Sequence<Int> = sequence { // sequence builder
    for (i in 1..3) {
        Thread.sleep(100) // pretend we are computing it
        yield(i) // yield next value
    }
}

fun main() {
    simple().forEach { value -> println(value) }
}
```

You can get the full code from [here](#).

This code outputs the same numbers, but it waits 100ms before printing each one.

Suspending functions

However, this computation blocks the main thread that is running the code. When these values are computed by asynchronous code we can mark the simple function with a suspend modifier, so that it can perform its work without blocking and return the result as a list:

```
import kotlinx.coroutines.*

//sampleStart
suspend fun simple(): List<Int> {
    delay(1000) // pretend we are doing something asynchronous here
    return listOf(1, 2, 3)
}

fun main() = runBlocking<Unit> {
    simple().forEach { value -> println(value) }
}
//sampleEnd
```

You can get the full code from [here](#).

This code prints the numbers after waiting for a second.

Flows

Using the `List<Int>` result type, means we can only return all the values at once. To represent the stream of values that

are being asynchronously computed, we can use a `Flow<Int>` type just like we would use the `Sequence<Int>` type for synchronously computed values:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow { // flow builder
    for (i in 1..3) {
        delay(100) // pretend we are doing something useful here
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
    // Launch a concurrent coroutine to check if the main thread is blocked
    launch {
        for (k in 1..3) {
            println("I'm not blocked $k")
            delay(100)
        }
    }
    // Collect the flow
    simple().collect { value -> println(value) }
}
//sampleEnd
```

You can get the full code from [here](#).

This code waits 100ms before printing each number without blocking the main thread. This is verified by printing "I'm not blocked" every 100ms from a separate coroutine that is running in the main thread:

```
I'm not blocked 1
1
I'm not blocked 2
2
I'm not blocked 3
3
```

Notice the following differences in the code with the `Flow` from the earlier examples:

- A builder function for `Flow` type is called `flow`.
- Code inside the `flow { ... }` builder block can suspend.
- The `simple` function is no longer marked with `suspend` modifier.
- Values are emitted from the `flow` using `emit` function.
- Values are collected from the `flow` using `collect` function.

We can replace `delay` with `Thread.sleep` in the body of `foo`'s flow { ... } and see that the main thread is blocked in this case.

Flows are cold

Flows are cold streams similar to sequences — the code inside a `flow` builder does not run until the flow is collected. This becomes clear in the following example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    println("Flow started")
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    println("Calling simple function...")
    val flow = simple()
    println("Calling collect...")
    flow.collect { value -> println(value) }
    println("Calling collect again...")
    flow.collect { value -> println(value) }
}
//sampleEnd
```

You can get the full code from [here](#).

Which prints:

```
Calling simple function...
Calling collect...
Flow started
1
2
3
Calling collect again...
Flow started
1
2
3
```

This is a key reason the `simple` function (which returns a flow) is not marked with `suspend` modifier. By itself, `simple()`

call returns quickly and does not wait for anything. The flow starts every time it is collected, that is why we see "Flow started" when we call collect again.

Flow cancellation basics

Flow adheres to the general cooperative cancellation of coroutines. As usual, flow collection can be cancelled when the flow is suspended in a cancellable suspending function (like `delay`). The following example shows how the flow gets cancelled on a timeout when running in a `withTimeoutOrNull` block and stops executing its code:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100)
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    withTimeoutOrNull(250) { // Timeout after 250ms
        simple().collect { value -> println(value) }
    }
    println("Done")
}
//sampleEnd
```

You can get the full code from [here](#).

Notice how only two numbers get emitted by the flow in the simple function, producing the following output:

```
Emitting 1
1
Emitting 2
2
Done
```

See [Flow cancellation checks](#) section for more details.

Flow builders

The flow { ... } builder from the previous examples is the most basic one. There are other builders for easier declaration of flows:

- `flowOf` builder that defines a flow emitting a fixed set of values.
- Various collections and sequences can be converted to flows using `.asFlow()` extension functions.

So, the example that prints the numbers from 1 to 3 from a flow can be written as:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    // Convert an integer range to a flow
    (1..3).asFlow().collect { value -> println(value) }
//sampleEnd
}
```

You can get the full code from [here](#).

Intermediate flow operators

Flows can be transformed with operators, just as you would with collections and sequences. Intermediate operators are applied to an upstream flow and return a downstream flow. These operators are cold, just like flows are. A call to such an operator is not a suspending function itself. It works quickly, returning the definition of a new transformed flow.

The basic operators have familiar names like `map` and `filter`. The important difference to sequences is that blocks of code inside these operators can call suspending functions.

For example, a flow of incoming requests can be mapped to the results with the `map` operator, even when performing a request is a long-running operation that is implemented by a suspending function:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
suspend fun performRequest(request: Int): String {
    delay(1000) // imitate long-running asynchronous work
    return "response $request"
}

fun main() = runBlocking<Unit> {
    (1..3).asFlow() // a flow of requests
        .map { request -> performRequest(request) }
        .collect { response -> println(response) }
}
//sampleEnd
```

You can get the full code from [here](#).

It produces the following three lines, each line appearing after each second:

```
response 1  
response 2  
response 3
```

Transform operator

Among the flow transformation operators, the most general one is called `transform`. It can be used to imitate simple transformations like `map` and `filter`, as well as implement more complex transformations. Using the transform operator, we can `emit` arbitrary values an arbitrary number of times.

For example, using transform we can emit a string before performing a long-running asynchronous request and follow it with a response:

```
import kotlinx.coroutines.*  
import kotlinx.coroutines.flow.*  
  
suspend fun performRequest(request: Int): String {  
    delay(1000) // imitate long-running asynchronous work  
    return "response $request"  
}  
  
fun main() = runBlocking<Unit> {  
    //sampleStart  
    (1..3).asFlow() // a flow of requests  
        .transform { request ->  
            emit("Making request $request")  
            emit(performRequest(request))  
        }  
        .collect { response -> println(response) }  
    //sampleEnd  
}
```

You can get the full code from [here](#).

The output of this code is:

```
Making request 1  
response 1  
Making request 2  
response 2  
Making request 3  
response 3
```

Size-limiting operators

Size-limiting intermediate operators like `take` cancel the execution of the flow when the corresponding limit is reached. Cancellation in coroutines is always performed by throwing an exception, so that all the resource-management functions (like `try { ... } finally { ... }` blocks) operate normally in case of cancellation:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun numbers(): Flow<Int> = flow {
    try {
        emit(1)
        emit(2)
        println("This line will not execute")
        emit(3)
    } finally {
        println("Finally in numbers")
    }
}

fun main() = runBlocking<Unit> {
    numbers()
        .take(2) // take only the first two
        .collect { value -> println(value) }
}
//sampleEnd
```

You can get the full code from [here](#).

The output of this code clearly shows that the execution of the `flow { ... }` body in the `numbers()` function stopped after emitting the second number:

```
1
2
Finally in numbers
```

Terminal flow operators

Terminal operators on flows are suspending functions that start a collection of the flow. The `collect` operator is the most basic one, but there are other terminal operators, which can make it easier:

- Conversion to various collections like `toList` and `toSet`.
- Operators to get the `first` value and to ensure that a flow emits a `single` value.
- Reducing a flow to a value with `reduce` and `fold`.

For example:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    val sum = (1..5).asFlow()
        .map { it * it } // squares of numbers from 1 to 5
        .reduce { a, b -> a + b } // sum them (terminal operator)
    println(sum)
//sampleEnd
}
```

You can get the full code from [here](#).

Prints a single number:

```
55
```

Flows are sequential

Each individual collection of a flow is performed sequentially unless special operators that operate on multiple flows are used. The collection works directly in the coroutine that calls a terminal operator. No new coroutines are launched by default. Each emitted value is processed by all the intermediate operators from upstream to downstream and is then delivered to the terminal operator after.

See the following example that filters the even integers and maps them to strings:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    (1..5).asFlow()
        .filter {
            println("Filter $it")
            it % 2 == 0
        }
        .map {
            println("Map $it")
            "string $it"
        }.collect {
            println("Collect $it")
        }
//sampleEnd
}
```

You can get the full code from [here](#).

Producing:

```
Filter 1
Filter 2
Map 2
Collect string 2
Filter 3
Filter 4
Map 4
Collect string 4
Filter 5
```

Flow context

Collection of a flow always happens in the context of the calling coroutine. For example, if there is a simple flow, then the following code runs in the context specified by the author of this code, regardless of the implementation details of the simple flow:

```
withContext(context) {
    simple().collect { value ->
        println(value) // run in the specified context
    }
}
```

This property of a flow is called context preservation.

So, by default, code in the flow { ... } builder runs in the context that is provided by a collector of the corresponding flow. For example, consider the implementation of a simple function that prints the thread it is called on and emits three numbers:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

//sampleStart
fun simple(): Flow<Int> = flow {
    log("Started simple flow")
    for (i in 1..3) {
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    simple().collect { value -> log("Collected $value") }
}
```

```
//sampleEnd
```

You can get the full code from [here](#).

Running this code produces:

```
[main @coroutine#1] Started simple flow
[main @coroutine#1] Collected 1
[main @coroutine#1] Collected 2
[main @coroutine#1] Collected 3
```

Since `simple().collect` is called from the main thread, the body of `simple`'s flow is also called in the main thread. This is the perfect default for fast-running or asynchronous code that does not care about the execution context and does not block the caller.

Wrong emission withContext

However, the long-running CPU-consuming code might need to be executed in the context of `Dispatchers.Default` and UI-updating code might need to be executed in the context of `Dispatchers.Main`. Usually, `withContext` is used to change the context in the code using Kotlin coroutines, but code in the flow `{ ... }` builder has to honor the context preservation property and is not allowed to `emit` from a different context.

Try running the following code:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    // The WRONG way to change context for CPU-consuming code in flow builder
    kotlinx.coroutines.withContext(Dispatchers.Default) {
        for (i in 1..3) {
            Thread.sleep(100) // pretend we are computing it in CPU-consuming way
            emit(i) // emit next value
        }
    }
}

fun main() = runBlocking<Unit> {
    simple().collect { value -> println(value) }
}
//sampleEnd
```

You can get the full code from [here](#).

This code produces the following exception:

```
Exception in thread "main" java.lang.IllegalStateException: Flow invariant is violated:  
    Flow was collected in [CoroutineId(1), "coroutine#1":BlockingCoroutine{Active}@05511c7f8,  
    BlockingEventLoop@2eac3323],  
    but emission happened in [CoroutineId(1), "coroutine#1":DispatchedCoroutine{Active}@2dae0000,  
    Dispatchers.Default].  
    Please refer to 'flow' documentation or use 'flowOn' instead  
    at ...
```

flowOn operator

The exception refers to the [flowOn](#) function that shall be used to change the context of the flow emission. The correct way to change the context of a flow is shown in the example below, which also prints the names of the corresponding threads to show how it all works:

```
import kotlinx.coroutines.*  
import kotlinx.coroutines.flow.*  
  
fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")  
  
//sampleStart  
fun simple(): Flow<Int> = flow {  
    for (i in 1..3) {  
        Thread.sleep(100) // pretend we are computing it in CPU-consuming way  
        log("Emitting $i")  
        emit(i) // emit next value  
    }  
}.flowOn(Dispatchers.Default) // RIGHT way to change context for CPU-consuming code in flow  
builder  
  
fun main() = runBlocking<Unit> {  
    simple().collect { value ->  
        log("Collected $value")  
    }  
}  
//sampleEnd
```

You can get the full code from [here](#).

Notice how `flow { ... }` works in the background thread, while collection happens in the main thread:

Another thing to observe here is that the [flowOn](#) operator has changed the default sequential nature of the flow. Now collection happens in one coroutine ("coroutine#1") and emission happens in another coroutine ("coroutine#2") that is running in another thread concurrently with the collecting coroutine. The [flowOn](#) operator creates another coroutine for an upstream flow when it has to change the [CoroutineDispatcher](#) in its context.

Buffering

Running different parts of a flow in different coroutines can be helpful from the standpoint of the overall time it takes to collect the flow, especially when long-running asynchronous operations are involved. For example, consider a case when the emission by a simple flow is slow, taking 100 ms to produce an element; and collector is also slow, taking 300 ms to process an element. Let's see how long it takes to collect such a flow with three numbers:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // pretend we are asynchronously waiting 100 ms
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
    val time = measureTimeMillis {
        simple().collect { value ->
            delay(300) // pretend we are processing it for 300 ms
            println(value)
        }
    }
    println("Collected in $time ms")
}
//sampleEnd
```

You can get the full code from [here](#).

It produces something like this, with the whole collection taking around 1200 ms (three numbers, 400 ms for each):

```
1
2
3
Collected in 1220 ms
```

We can use a `buffer` operator on a flow to run emitting code of the simple flow concurrently with collecting code, as opposed to running them sequentially:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // pretend we are asynchronously waiting 100 ms
    }
}
```

```

        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        simple()
            .buffer() // buffer emissions, don't wait
            .collect { value ->
                delay(300) // pretend we are processing it for 300 ms
                println(value)
            }
    }
    println("Collected in $time ms")
//sampleEnd
}

```

You can get the full code from [here](#).

It produces the same numbers just faster, as we have effectively created a processing pipeline, having to only wait 100 ms for the first number and then spending only 300 ms to process each number. This way it takes around 1000 ms to run:

```

1
2
3
Collected in 1071 ms

```

Note that the `flowOn` operator uses the same buffering mechanism when it has to change a `CoroutineDispatcher`, but here we explicitly request buffering without changing the execution context.

Conflation

When a flow represents partial results of the operation or operation status updates, it may not be necessary to process each value, but instead, only most recent ones. In this case, the `conflate` operator can be used to skip intermediate values when a collector is too slow to process them. Building on the previous example:

```

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // pretend we are asynchronously waiting 100 ms
        emit(i) // emit next value
    }
}

```

```

}

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        simple()
            .conflate() // conflate emissions, don't process each one
            .collect { value ->
                delay(300) // pretend we are processing it for 300 ms
                println(value)
            }
    }
    println("Collected in $time ms")
//sampleEnd
}

```

You can get the full code from [here](#).

We see that while the first number was still being processed the second, and third were already produced, so the second one was conflated and only the most recent (the third one) was delivered to the collector:

```

1
3
Collected in 758 ms

```

Processing the latest value

Conflation is one way to speed up processing when both the emitter and collector are slow. It does it by dropping emitted values. The other way is to cancel a slow collector and restart it every time a new value is emitted. There is a family of xxxLatest operators that perform the same essential logic of a xxx operator, but cancel the code in their block on a new value. Let's try changing `conflate` to `collectLatest` in the previous example:

```

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // pretend we are asynchronously waiting 100 ms
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        simple()
            .collectLatest { value -> // cancel & restart on the latest value
                println("Collecting $value")
                delay(300) // pretend we are processing it for 300 ms
            }
    }
    println("Collected in $time ms")
//sampleEnd
}

```

```

        println("Done $value")
    }
}
println("Collected in $time ms")
//sampleEnd
}

```

You can get the full code from [here](#).

Since the body of `collectLatest` takes 300 ms, but new values are emitted every 100 ms, we see that the block is run on every value, but completes only for the last value:

```

Collecting 1
Collecting 2
Collecting 3
Done 3
Collected in 741 ms

```

Composing multiple flows

There are lots of ways to compose multiple flows.

Zip

Just like the `Sequence.zip` extension function in the Kotlin standard library, flows have a `zip` operator that combines the corresponding values of two flows:

```

import kotlinox.coroutines.*
import kotlinox.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    val nums = (1..3).asFlow() // numbers 1..3
    val strs = flowOf("one", "two", "three") // strings
    nums.zip(strs) { a, b -> "$a -> $b" } // compose a single string
        .collect { println(it) } // collect and print
//sampleEnd
}

```

You can get the full code from [here](#).

This example prints:

```
1 -> one
2 -> two
3 -> three
```

Combine

When flow represents the most recent value of a variable or operation (see also the related section on [conflation](#)), it might be needed to perform a computation that depends on the most recent values of the corresponding flows and to recompute it whenever any of the upstream flows emit a value. The corresponding family of operators is called [combine](#).

For example, if the numbers in the previous example update every 300ms, but strings update every 400 ms, then zipping them using the `zip` operator will still produce the same result, albeit results that are printed every 400 ms:

We use a `onEach` intermediate operator in this example to delay each element and make the code that emits sample flows more declarative and shorter.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    val nums = (1..3).asFlow().onEach { delay(300) } // numbers 1..3 every 300 ms
    val strs = flowOf("one", "two", "three").onEach { delay(400) } // strings every 400 ms
    val startTime = System.currentTimeMillis() // remember the start time
    nums.zip(strs) { a, b -> "$a -> $b" } // compose a single string with "zip"
        .collect { value -> // collect and print
            println("$value at ${System.currentTimeMillis() - startTime} ms from start")
        }
//sampleEnd
}
```

You can get the full code from [here](#).

However, when using a `combine` operator here instead of a `zip`:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    val nums = (1..3).asFlow().onEach { delay(300) } // numbers 1..3 every 300 ms
    val strs = flowOf("one", "two", "three").onEach { delay(400) } // strings every 400 ms
    val startTime = System.currentTimeMillis() // remember the start time
    nums.combine(strs) { a, b -> "$a -> $b" } // compose a single string with "combine"
```

```

    .collect { value -> // collect and print
        println("$value at ${System.currentTimeMillis() - startTime} ms from start")
    }
//sampleEnd
}

```

You can get the full code from [here](#).

We get quite a different output, where a line is printed at each emission from either nums or strs flows:

```

1 -> one at 452 ms from start
2 -> one at 651 ms from start
2 -> two at 854 ms from start
3 -> two at 952 ms from start
3 -> three at 1256 ms from start

```

Flattening flows

Flows represent asynchronously received sequences of values, so it is quite easy to get in a situation where each value triggers a request for another sequence of values. For example, we can have the following function that returns a flow of two strings 500 ms apart:

```

fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // wait 500 ms
    emit("$i: Second")
}

```

Now if we have a flow of three integers and call `requestFlow` for each of them like this:

```
(1..3).asFlow().map { requestFlow(it) }
```

Then we end up with a flow of flows (`Flow<Flow<String>>`) that needs to be flattened into a single flow for further processing. Collections and sequences have `flatten` and `flatMap` operators for this. However, due to the asynchronous nature of flows they call for different modes of flattening, as such, there is a family of flattening operators on flows.

flatMapConcat

Concatenating mode is implemented by `flatMapConcat` and `flattenConcat` operators. They are the most direct analogues of the corresponding sequence operators. They wait for the inner flow to complete before starting to collect the next one as the following example shows:

```
import kotlinx.coroutines.*
```

```

import kotlinx.coroutines.*

fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // wait 500 ms
    emit("$i: Second")
}

fun main() = runBlocking<Unit> {
//sampleStart
    val startTime = System.currentTimeMillis() // remember the start time
    (1..3).asFlow().onEach { delay(100) } // a number every 100 ms
        .flatMapConcat { requestFlow(it) }
        .collect { value -> // collect and print
            println("$value at ${System.currentTimeMillis()} ms from start")
        }
//sampleEnd
}

```

You can get the full code from [here](#).

The sequential nature of `flatMapConcat` is clearly seen in the output:

```

1: First at 121 ms from start
1: Second at 622 ms from start
2: First at 727 ms from start
2: Second at 1227 ms from start
3: First at 1328 ms from start
3: Second at 1829 ms from start

```

flatMapMerge

Another flattening mode is to concurrently collect all the incoming flows and merge their values into a single flow so that values are emitted as soon as possible. It is implemented by `flatMapMerge` and `flattenMerge` operators. They both accept an optional concurrency parameter that limits the number of concurrent flows that are collected at the same time (it is equal to `DEFAULT_CONCURRENCY` by default).

```

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // wait 500 ms
    emit("$i: Second")
}

fun main() = runBlocking<Unit> {
//sampleStart
    val startTime = System.currentTimeMillis() // remember the start time
    (1..3).asFlow().onEach { delay(100) } // a number every 100 ms

```

```

        .flatMapMerge { requestFlow(it) }
        .collect { value -> // collect and print
            println("$value at ${System.currentTimeMillis() - startTime} ms from start")
        }
    //sampleEnd
}

```

You can get the full code from [here](#).

The concurrent nature of `flatMapMerge` is obvious:

```

1: First at 136 ms from start
2: First at 231 ms from start
3: First at 333 ms from start
1: Second at 639 ms from start
2: Second at 732 ms from start
3: Second at 833 ms from start

```

Note that the `flatMapMerge` calls its block of code (`{ requestFlow(it) }` in this example) sequentially, but collects the resulting flows concurrently, it is the equivalent of performing a sequential map `{ requestFlow(it) }` first and then calling `flattenMerge` on the result.

flatMapLatest

In a similar way to the `collectLatest` operator, that was shown in "Processing the latest value" section, there is the corresponding "Latest" flattening mode where a collection of the previous flow is cancelled as soon as new flow is emitted. It is implemented by the `flatMapLatest` operator.

```

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // wait 500 ms
    emit("$i: Second")
}

fun main() = runBlocking<Unit> {
    //sampleStart
    val startTime = System.currentTimeMillis() // remember the start time
    (1..3).asFlow().onEach { delay(100) } // a number every 100 ms
        .flatMapLatest { requestFlow(it) }
        .collect { value -> // collect and print
            println("$value at ${System.currentTimeMillis() - startTime} ms from start")
        }
    //sampleEnd
}

```

You can get the full code from [here](#).

The output here in this example is a good demonstration of how `flatMapLatest` works:

```
1: First at 142 ms from start
2: First at 322 ms from start
3: First at 425 ms from start
3: Second at 931 ms from start
```

Note that `flatMapLatest` cancels all the code in its block (`{ requestFlow(it) }` in this example) on a new value. It makes no difference in this particular example, because the call to `requestFlow` itself is fast, not-suspending, and cannot be cancelled. However, it would show up if we were to use suspending functions like `delay` in there.

Flow exceptions

Flow collection can complete with an exception when an emitter or code inside the operators throw an exception.

There are several ways to handle these exceptions.

Collector try and catch

A collector can use Kotlin's `try/catch` block to handle exceptions:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
    try {
        simple().collect { value ->
            println(value)
            check(value <= 1) { "Collected $value" }
        }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
//sampleEnd
```

You can get the full code from [here](#).

This code successfully catches an exception in `collect` terminal operator and, as we see, no more values are emitted after that:

```
Emitting 1
1
Emitting 2
2
Caught java.lang.IllegalStateException: Collected 2
```

Everything is caught

The previous example actually catches any exception happening in the emitter or in any intermediate or terminal operators. For example, let's change the code so that emitted values are mapped to strings, but the corresponding code produces an exception:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<String> =
    flow {
        for (i in 1..3) {
            println("Emitting $i")
            emit(i) // emit next value
        }
    }
    .map { value ->
        check(value <= 1) { "Crashed on $value" }
        "string $value"
    }

fun main() = runBlocking<Unit> {
    try {
        simple().collect { value -> println(value) }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
//sampleEnd
```

You can get the full code from [here](#).

This exception is still caught and collection is stopped:

```
Emitting 1
```

```
string 1
Emitting 2
Caught java.lang.IllegalStateException: Crashed on 2
```

Exception transparency

But how can code of the emitter encapsulate its exception handling behavior?

Flows must be transparent to exceptions and it is a violation of the exception transparency to `emit` values in the flow { ... } builder from inside of a try/catch block. This guarantees that a collector throwing an exception can always catch it using try/catch as in the previous example.

The emitter can use a `catch` operator that preserves this exception transparency and allows encapsulation of its exception handling. The body of the catch operator can analyze an exception and react to it in different ways depending on which exception was caught:

- Exceptions can be rethrown using `throw`.
- Exceptions can be turned into emission of values using `emit` from the body of `catch`.
- Exceptions can be ignored, logged, or processed by some other code.

For example, let us emit the text on catching an exception:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun simple(): Flow<String> =
    flow {
        for (i in 1..3) {
            println("Emitting $i")
            emit(i) // emit next value
        }
    }
    .map { value ->
        check(value <= 1) { "Crashed on $value" }
        "string $value"
    }

fun main() = runBlocking<Unit> {
//sampleStart
    simple()
        .catch { e -> emit("Caught $e") } // emit on exception
        .collect { value -> println(value) }
//sampleEnd
}
```

You can get the full code from [here](#).

The output of the example is the same, even though we do not have try/catch around the code anymore.

Transparent catch

The `catch` intermediate operator, honoring exception transparency, catches only upstream exceptions (that is an exception from all the operators above `catch`, but not below it). If the block in `collect { ... }` (placed below `catch`) throws an exception then it escapes:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    simple()
        .catch { e -> println("Caught $e") } // does not catch downstream exceptions
        .collect { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
}
//sampleEnd
```

You can get the full code from [here](#).

A "Caught ..." message is not printed despite there being a `catch` operator:

```
Emitting 1
1
Emitting 2
Exception in thread "main" java.lang.IllegalStateException: Collected 2
at ...
```

Catching declaratively

We can combine the declarative nature of the `catch` operator with a desire to handle all the exceptions, by moving the body of the `collect` operator into `onEach` and putting it before the `catch` operator. Collection of this flow must be triggered by a call to `collect()` without parameters:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
```

```

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    simple()
        .onEach { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
        .catch { e -> println("Caught $e") }
        .collect()
//sampleEnd
}

```

You can get the full code from [here](#).

Now we can see that a "Caught ..." message is printed and so we can catch all the exceptions without explicitly using a try/catch block:

```

Emitting 1
1
Emitting 2
Caught java.lang.IllegalStateException: Collected 2

```

Flow completion

When flow collection completes (normally or exceptionally) it may need to execute an action. As you may have already noticed, it can be done in two ways: imperative or declarative.

Imperative finally block

In addition to try/ catch, a collector can also use a finally block to execute an action upon collect completion.

```

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
    try {
        simple().collect { value -> println(value) }
    } finally {

```

```
        println("Done")
    }
}
//sampleEnd
```

You can get the full code from [here](#).

This code prints three numbers produced by the simple flow followed by a "Done" string:

```
1
2
3
Done
```

Declarative handling

For the declarative approach, flow has `onCompletion` intermediate operator that is invoked when the flow has completely collected.

The previous example can be rewritten using an `onCompletion` operator and produces the same output:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
//sampleStart
    simple()
        .onCompletion { println("Done") }
        .collect { value -> println(value) }
//sampleEnd
}
```

You can get the full code from [here](#).

The key advantage of `onCompletion` is a nullable `Throwable` parameter of the lambda that can be used to determine whether the flow collection was completed normally or exceptionally. In the following example the simple flow throws an exception after emitting the number 1:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
```

```

    emit(1)
    throw RuntimeException()
}

fun main() = runBlocking<Unit> {
    simple()
        .onCompletion { cause -> if (cause != null) println("Flow completed exceptionally") }
        .catch { cause -> println("Caught exception") }
        .collect { value -> println(value) }
}
//sampleEnd

```

You can get the full code from [here](#).

As you may expect, it prints:

```

1
Flow completed exceptionally
Caught exception

```

The `onCompletion` operator, unlike `catch`, does not handle the exception. As we can see from the above example code, the exception still flows downstream. It will be delivered to further `onCompletion` operators and can be handled with a `catch` operator.

Successful completion

Another difference with `catch` operator is that `onCompletion` sees all exceptions and receives a null exception only on successful completion of the upstream flow (without cancellation or failure).

```

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
    simple()
        .onCompletion { cause -> println("Flow completed with $cause") }
        .collect { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
}
//sampleEnd

```

You can get the full code from [here](#).

We can see the completion cause is not null, because the flow was aborted due to downstream exception:

```
1
Flow completed with java.lang.IllegalStateException: Collected 2
Exception in thread "main" java.lang.IllegalStateException: Collected 2
```

Imperative versus declarative

Now we know how to collect flow, and handle its completion and exceptions in both imperative and declarative ways. The natural question here is, which approach is preferred and why? As a library, we do not advocate for any particular approach and believe that both options are valid and should be selected according to your own preferences and code style.

Launching flow

It is easy to use flows to represent asynchronous events that are coming from some source. In this case, we need an analogue of the `addEventListener` function that registers a piece of code with a reaction for incoming events and continues further work. The `onEach` operator can serve this role. However, `onEach` is an intermediate operator. We also need a terminal operator to collect the flow. Otherwise, just calling `onEach` has no effect.

If we use the `collect` terminal operator after `onEach`, then the code after it will wait until the flow is collected:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
// Imitate a flow of events
fun events(): Flow<Int> = (1..3).asFlow().onEach { delay(100) }

fun main() = runBlocking<Unit> {
    events()
        .onEach { event -> println("Event: $event") }
        .collect() // <--- Collecting the flow waits
    println("Done")
}
//sampleEnd
```

You can get the full code from [here](#).

As you can see, it prints:

```
Event: 1
Event: 2
Event: 3
```

Done

The `launchIn` terminal operator comes in handy here. By replacing `collect` with `launchIn` we can launch a collection of the flow in a separate coroutine, so that execution of further code immediately continues:

```
import kotlinx.coroutines.*  
import kotlinx.coroutines.flow.*  
  
// Imitate a flow of events  
fun events(): Flow<Int> = (1..3).asFlow().onEach { delay(100) }  
  
//sampleStart  
fun main() = runBlocking<Unit> {  
    events()  
        .onEach { event -> println("Event: $event") }  
        .launchIn(this) // <-- Launching the flow in a separate coroutine  
    println("Done")  
}  
//sampleEnd
```

You can get the full code from [here](#).

It prints:

```
Done  
Event: 1  
Event: 2  
Event: 3
```

The required parameter to `launchIn` must specify a `CoroutineScope` in which the coroutine to collect the flow is launched. In the above example this scope comes from the `runBlocking` coroutine builder, so while the flow is running, this `runBlocking` scope waits for completion of its child coroutine and keeps the main function from returning and terminating this example.

In actual applications a scope will come from an entity with a limited lifetime. As soon as the lifetime of this entity is terminated the corresponding scope is cancelled, cancelling the collection of the corresponding flow. This way the pair of `onEach { ... }.launchIn(scope)` works like the `addEventListener`. However, there is no need for the corresponding `removeEventListener` function, as cancellation and structured concurrency serve this purpose.

Note that `launchIn` also returns a `Job`, which can be used to `cancel` the corresponding flow collection coroutine only without cancelling the whole scope or to `join` it.

Flow cancellation checks

For convenience, the `flow` builder performs additional `ensureActive` checks for cancellation on each emitted value. It means that a busy loop emitting from a `flow { ... }` is cancellable:

```

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun foo(): Flow<Int> = flow {
    for (i in 1..5) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    foo().collect { value ->
        if (value == 3) cancel()
        println(value)
    }
}
//sampleEnd

```

You can get the full code from [here](#).

We get only numbers up to 3 and a `CancellationException` after trying to emit number 4:

```

Emitting 1
1
Emitting 2
2
Emitting 3
3
Emitting 4
Exception in thread "main" kotlinx.coroutines.JobCancellationException: BlockingCoroutine was
cancelled; job="coroutine#1":BlockingCoroutine{Cancelled}@6d7b4f4c

```

However, most other flow operators do not do additional cancellation checks on their own for performance reasons. For example, if you use `IntRange.asFlow` extension to write the same busy loop and don't suspend anywhere, then there are no checks for cancellation:

```

import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun main() = runBlocking<Unit> {
    (1..5).asFlow().collect { value ->
        if (value == 3) cancel()
        println(value)
    }
}
//sampleEnd

```

You can get the full code from [here](#).

All numbers from 1 to 5 are collected and cancellation gets detected only before return from runBlocking:

```
1
2
3
4
5
Exception in thread "main" kotlinx.coroutines.JobCancellationException: BlockingCoroutine was
cancelled; job="coroutine#1":BlockingCoroutine{Cancelled}@3327bd23
```

Making busy flow cancellable

In the case where you have a busy loop with coroutines you must explicitly check for cancellation. You can add `.onEach { currentCoroutineContext().ensureActive() }`, but there is a ready-to-use `cancellable` operator provided to do that:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun main() = runBlocking<Unit> {
    (1..5).asFlow().cancellable().collect { value ->
        if (value == 3) cancel()
        println(value)
    }
}
//sampleEnd
```

You can get the full code from [here](#).

With the cancellable operator only the numbers from 1 to 3 are collected:

```
1
2
3
Exception in thread "main" kotlinx.coroutines.JobCancellationException: BlockingCoroutine was
cancelled; job="coroutine#1":BlockingCoroutine{Cancelled}@5ec0a365
```

Flow and Reactive Streams

For those who are familiar with [Reactive Streams](#) or reactive frameworks such as RxJava and project Reactor, design of the Flow may look very familiar.

Indeed, its design was inspired by Reactive Streams and its various implementations. But Flow main goal is to have as simple design as possible, be Kotlin and suspension friendly and respect structured concurrency. Achieving this goal would be impossible without reactive pioneers and their tremendous work. You can read the complete story in [Reactive Streams and Kotlin Flows](#) article.

While being different, conceptually, Flow is a reactive stream and it is possible to convert it to the reactive (spec and TCK compliant) Publisher and vice versa. Such converters are provided by `kotlinx.coroutines` out-of-the-box and can be found in corresponding reactive modules (`kotlinx-coroutines-reactive` for Reactive Streams, `kotlinx-coroutines-reactor` for Project Reactor and `kotlinx-coroutines-rx2/ kotlinx-coroutines-rx3` for RxJava2/RxJava3). Integration modules include conversions from and to Flow, integration with Reactor's Context and suspension-friendly ways to work with various reactive entities.

Channels

Deferred values provide a convenient way to transfer a single value between coroutines. Channels provide a way to transfer a stream of values.

Channel basics

A [Channel](#) is conceptually very similar to `BlockingQueue`. One key difference is that instead of a blocking `put` operation it has a suspending `send`, and instead of a blocking `take` operation it has a suspending `receive`.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val channel = Channel<Int>()
    launch {
        // this might be heavy CPU-consuming computation or async logic, we'll just send five
        squares
        for (x in 1..5) channel.send(x * x)
    }
    // here we print five received integers:
    repeat(5) { println(channel.receive()) }
    println("Done!")
//sampleEnd
}
```

You can get the full code [here](#).

The output of this code is:

```
1  
4  
9  
16  
25  
Done!
```

Closing and iteration over channels

Unlike a queue, a channel can be closed to indicate that no more elements are coming. On the receiver side it is convenient to use a regular for loop to receive elements from the channel.

Conceptually, a close is like sending a special close token to the channel. The iteration stops as soon as this close token is received, so there is a guarantee that all previously sent elements before the close are received:

```
import kotlinx.coroutines.*  
import kotlinx.coroutines.channels.*  
  
fun main() = runBlocking {  
    //sampleStart  
    val channel = Channel<Int>()  
    launch {  
        for (x in 1..5) channel.send(x * x)  
        channel.close() // we're done sending  
    }  
    // here we print received values using `for` loop (until the channel is closed)  
    for (y in channel) println(y)  
    println("Done!")  
    //sampleEnd  
}
```

You can get the full code [here](#).

Building channel producers

The pattern where a coroutine is producing a sequence of elements is quite common. This is a part of producer-consumer pattern that is often found in concurrent code. You could abstract such a producer into a function that takes channel as its parameter, but this goes contrary to common sense that results must be returned from functions.

There is a convenient coroutine builder named produce that makes it easy to do it right on producer side, and an extension function consumeEach, that replaces a for loop on the consumer side:

```
import kotlinx.coroutines.*  
import kotlinx.coroutines.channels.*  
  
fun CoroutineScope.produceSquares(): ReceiveChannel<Int> = produce {
```

```

        for (x in 1..5) send(x * x)
    }

fun main() = runBlocking {
//sampleStart
    val squares = produceSquares()
    squares.consumeEach { println(it) }
    println("Done!")
//sampleEnd
}

```

You can get the full code [here](#).

Pipelines

A pipeline is a pattern where one coroutine is producing, possibly infinite, stream of values:

```

fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // infinite stream of integers starting from 1
}

```

And another coroutine or coroutines are consuming that stream, doing some processing, and producing some other results. In the example below, the numbers are just squared:

```

fun CoroutineScope.square(numbers: ReceiveChannel<Int>): ReceiveChannel<Int> = produce {
    for (x in numbers) send(x * x)
}

```

The main code starts and connects the whole pipeline:

```

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val numbers = produceNumbers() // produces integers from 1 and on
    val squares = square(numbers) // squares integers
    repeat(5) {
        println(squares.receive()) // print first five
    }
    println("Done!") // we are done
    coroutineContext.cancelChildren() // cancel children coroutines
//sampleEnd
}

fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // infinite stream of integers starting from 1
}

```

```

    }

fun CoroutineScope.square(numbers: ReceiveChannel<Int>): ReceiveChannel<Int> = produce {
    for (x in numbers) send(x * x)
}

```

You can get the full code [here](#).

All functions that create coroutines are defined as extensions on `CoroutineScope`, so that we can rely on `structured concurrency` to make sure that we don't have lingering global coroutines in our application.

Prime numbers with pipeline

Let's take pipelines to the extreme with an example that generates prime numbers using a pipeline of coroutines. We start with an infinite sequence of numbers.

```

fun CoroutineScope.numbersFrom(start: Int) = produce<Int> {
    var x = start
    while (true) send(x++) // infinite stream of integers from start
}

```

The following pipeline stage filters an incoming stream of numbers, removing all the numbers that are divisible by the given prime number:

```

fun CoroutineScope.filter(numbers: ReceiveChannel<Int>, prime: Int) = produce<Int> {
    for (x in numbers) if (x % prime != 0) send(x)
}

```

Now we build our pipeline by starting a stream of numbers from 2, taking a prime number from the current channel, and launching new pipeline stage for each prime number found:

```
numbersFrom(2) -> filter(2) -> filter(3) -> filter(5) -> filter(7) ...
```

The following example prints the first ten prime numbers, running the whole pipeline in the context of the main thread. Since all the coroutines are launched in the scope of the main `runBlocking` coroutine we don't have to keep an explicit list of all the coroutines we have started. We use `cancelChildren` extension function to cancel all the children coroutines after we have printed the first ten prime numbers.

```

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart

```

```

var cur = numbersFrom(2)
repeat(10) {
    val prime = cur.receive()
    println(prime)
    cur = filter(cur, prime)
}
coroutineContext.cancelChildren() // cancel all children to let main finish
//sampleEnd
}

fun CoroutineScope.numbersFrom(start: Int) = produce<Int> {
    var x = start
    while (true) send(x++) // infinite stream of integers from start
}

fun CoroutineScope.filter(numbers: ReceiveChannel<Int>, prime: Int) = produce<Int> {
    for (x in numbers) if (x % prime != 0) send(x)
}

```

You can get the full code [here](#).

The output of this code is:

```

2
3
5
7
11
13
17
19
23
29

```

Note that you can build the same pipeline using `iterator` coroutine builder from the standard library. Replace `produce` with `iterator`, `send` with `yield`, `receive` with `next`, `ReceiveChannel` with `Iterator`, and get rid of the coroutine scope. You will not need `runBlocking` either. However, the benefit of a pipeline that uses channels as shown above is that it can actually use multiple CPU cores if you run it in `Dispatchers.Default` context.

Anyway, this is an extremely impractical way to find prime numbers. In practice, pipelines do involve some other suspending invocations (like asynchronous calls to remote services) and these pipelines cannot be built using `sequence`/`iterator`, because they do not allow arbitrary suspension, unlike `produce`, which is fully asynchronous.

Fan-out

Multiple coroutines may receive from the same channel, distributing work between themselves. Let us start with a producer coroutine that is periodically producing integers (ten numbers per second):

```

fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1 // start from 1
    while (true) {
        send(x++) // produce next
        delay(100) // wait 0.1s
    }
}

```

Then we can have several processor coroutines. In this example, they just print their id and received number:

```

fun CoroutineScope.launchProcessor(id: Int, channel: ReceiveChannel<Int>) = launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
}

```

Now let us launch five processors and let them work for almost a second. See what happens:

```

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking<Unit> {
//sampleStart
    val producer = produceNumbers()
    repeat(5) { launchProcessor(it, producer) }
    delay(950)
    producer.cancel() // cancel producer coroutine and thus kill them all
//sampleEnd
}

fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1 // start from 1
    while (true) {
        send(x++) // produce next
        delay(100) // wait 0.1s
    }
}

fun CoroutineScope.launchProcessor(id: Int, channel: ReceiveChannel<Int>) = launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
}

```

You can get the full code [here](#).

The output will be similar to the the following one, albeit the processor ids that receive each specific integer may be different:

```
Processor #2 received 1
```

```
Processor #4 received 2
Processor #0 received 3
Processor #1 received 4
Processor #3 received 5
Processor #2 received 6
Processor #4 received 7
Processor #0 received 8
Processor #1 received 9
Processor #3 received 10
```

Note that cancelling a producer coroutine closes its channel, thus eventually terminating iteration over the channel that processor coroutines are doing.

Also, pay attention to how we explicitly iterate over channel with for loop to perform fan-out in launchProcessor code. Unlike consumeEach, this for loop pattern is perfectly safe to use from multiple coroutines. If one of the processor coroutines fails, then others would still be processing the channel, while a processor that is written via consumeEach always consumes (cancels) the underlying channel on its normal or abnormal completion.

Fan-in

Multiple coroutines may send to the same channel. For example, let us have a channel of strings, and a suspending function that repeatedly sends a specified string to this channel with a specified delay:

```
suspend fun sendString(channel: SendChannel<String>, s: String, time: Long) {
    while (true) {
        delay(time)
        channel.send(s)
    }
}
```

Now, let us see what happens if we launch a couple of coroutines sending strings (in this example we launch them in the context of the main thread as main coroutine's children):

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val channel = Channel<String>()
    launch { sendString(channel, "foo", 200L) }
    launch { sendString(channel, "BAR!", 500L) }
    repeat(6) { // receive first six
        println(channel.receive())
    }
    coroutineContext.cancelChildren() // cancel all children to let main finish
//sampleEnd
}

suspend fun sendString(channel: SendChannel<String>, s: String, time: Long) {
    while (true) {
        delay(time)
```

```
        channel.send(s)
    }
}
```

You can get the full code [here](#).

The output is:

```
foo
foo
BAR!
foo
foo
BAR!
```

Buffered channels

The channels shown so far had no buffer. Unbuffered channels transfer elements when sender and receiver meet each other (aka rendezvous). If send is invoked first, then it is suspended until receive is invoked, if receive is invoked first, it is suspended until send is invoked.

Both `Channel()` factory function and `produce` builder take an optional capacity parameter to specify buffer size. Buffer allows senders to send multiple elements before suspending, similar to the `BlockingQueue` with a specified capacity, which blocks when buffer is full.

Take a look at the behavior of the following code:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking<Unit> {
//sampleStart
    val channel = Channel<Int>(4) // create buffered channel
    val sender = launch { // launch sender coroutine
        repeat(10) {
            println("Sending $it") // print before sending each element
            channel.send(it) // will suspend when buffer is full
        }
    }
    // don't receive anything... just wait....
    delay(1000)
    sender.cancel() // cancel sender coroutine
//sampleEnd
}
```

You can get the full code [here](#).

It prints "sending" five times using a buffered channel with capacity of four:

```
Sending 0
Sending 1
Sending 2
Sending 3
Sending 4
```

The first four elements are added to the buffer and the sender suspends when trying to send the fifth one.

Channels are fair

Send and receive operations to channels are fair with respect to the order of their invocation from multiple coroutines. They are served in first-in first-out order, e.g. the first coroutine to invoke receive gets the element. In the following example two coroutines "ping" and "pong" are receiving the "ball" object from the shared "table" channel.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

//sampleStart
data class Ball(var hits: Int)

fun main() = runBlocking {
    val table = Channel<Ball>() // a shared table
    launch { player("ping", table) }
    launch { player("pong", table) }
    table.send(Ball(0)) // serve the ball
    delay(1000) // delay 1 second
    coroutineContext.cancelChildren() // game over, cancel them
}

suspend fun player(name: String, table: Channel<Ball>) {
    for (ball in table) { // receive the ball in a loop
        ball.hits++
        println("$name $ball")
        delay(300) // wait a bit
        table.send(ball) // send the ball back
    }
}
//sampleEnd
```

You can get the full code [here](#).

The "ping" coroutine is started first, so it is the first one to receive the ball. Even though "ping" coroutine immediately

starts receiving the ball again after sending it back to the table, the ball gets received by the "pong" coroutine, because it was already waiting for it:

```
ping Ball(hits=1)
pong Ball(hits=2)
ping Ball(hits=3)
pong Ball(hits=4)
```

Note that sometimes channels may produce executions that look unfair due to the nature of the executor that is being used. See [this issue](#) for details.

Ticker channels

Ticker channel is a special rendezvous channel that produces Unit every time given delay passes since last consumption from this channel. Though it may seem to be useless standalone, it is a useful building block to create complex time-based [produce](#) pipelines and operators that do windowing and other time-dependent processing. Ticker channel can be used in [select](#) to perform "on tick" action.

To create such channel use a factory method [ticker](#). To indicate that no further elements are needed use [ReceiveChannel.cancel](#) method on it.

Now let's see how it works in practice:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking<Unit> {
    val tickerChannel = ticker(delayMillis = 100, initialDelayMillis = 0) // create ticker channel
    var nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
    println("Initial element is available immediately: $nextElement") // no initial delay

    nextElement = withTimeoutOrNull(50) { tickerChannel.receive() } // all subsequent elements have 100ms delay
    println("Next element is not ready in 50 ms: $nextElement")

    nextElement = withTimeoutOrNull(60) { tickerChannel.receive() }
    println("Next element is ready in 100 ms: $nextElement")

    // Emulate large consumption delays
    println("Consumer pauses for 150ms")
    delay(150)
    // Next element is available immediately
    nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
    println("Next element is available immediately after large consumer delay: $nextElement")
    // Note that the pause between `receive` calls is taken into account and next element arrives faster
    nextElement = withTimeoutOrNull(60) { tickerChannel.receive() }
    println("Next element is ready in 50ms after consumer pause in 150ms: $nextElement")

    tickerChannel.cancel() // indicate that no more elements are needed
}
```

You can get the full code [here](#).

It prints following lines:

```
Initial element is available immediately: kotlin.Unit
Next element is not ready in 50 ms: null
Next element is ready in 100 ms: kotlin.Unit
Consumer pauses for 150ms
Next element is available immediately after large consumer delay: kotlin.Unit
Next element is ready in 50ms after consumer pause in 150ms: kotlin.Unit
```

Note that `ticker` is aware of possible consumer pauses and, by default, adjusts next produced element delay if a pause occurs, trying to maintain a fixed rate of produced elements.

Optionally, a mode parameter equal to `TickerMode.FIXED_DELAY` can be specified to maintain a fixed delay between elements.

Coroutine exceptions handling

This section covers exception handling and cancellation on exceptions. We already know that a cancelled coroutine throws `CancellationException` in suspension points and that it is ignored by the coroutines' machinery. Here we look at what happens if an exception is thrown during cancellation or multiple children of the same coroutine throw an exception.

Exception propagation

Coroutine builders come in two flavors: propagating exceptions automatically (`launch` and `actor`) or exposing them to users (`async` and `produce`). When these builders are used to create a root coroutine, that is not a child of another coroutine, the former builders treat exceptions as uncaught exceptions, similar to Java's `Thread.uncaughtExceptionHandler`, while the latter are relying on the user to consume the final exception, for example via `await` or `receive` (`produce` and `receive` are covered later in `Channels` section).

It can be demonstrated by a simple example that creates root coroutines using the `GlobalScope`:

`GlobalScope` is a delicate API that can backfire in non-trivial ways. Creating a root coroutine for the whole application is one of the rare legitimate uses for `GlobalScope`, so you must explicitly opt-in into using `GlobalScope` with `@OptIn(DelicateCoroutinesApi::class)`.

```

import kotlinx.coroutines.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
    val job = GlobalScope.launch { // root coroutine with launch
        println("Throwing exception from launch")
        throw IndexOutOfBoundsException() // Will be printed to the console by
Thread.defaultUncaughtExceptionHandler
    }
    job.join()
    println("Joined failed job")
    val deferred = GlobalScope.async { // root coroutine with async
        println("Throwing exception from async")
        throw ArithmeticException() // Nothing is printed, relying on user to call await
    }
    try {
        deferred.await()
        println("Unreached")
    } catch (e: ArithmeticException) {
        println("Caught ArithmeticException")
    }
}

```

You can get the full code [here](#).

The output of this code is (with debug):

```

Throwing exception from launch
Exception in thread "DefaultDispatcher-worker-2 @coroutine#2"
java.lang.IndexOutOfBoundsException
Joined failed job
Throwing exception from async
Caught ArithmeticException

```

CoroutineExceptionHandler

It is possible to customize the default behavior of printing uncaught exceptions to the console.

CoroutineExceptionHandler context element on a root coroutine can be used as generic catch block for this root coroutine and all its children where custom exception handling may take place. It is similar to Thread.uncaughtExceptionHandler. You cannot recover from the exception in the CoroutineExceptionHandler. The coroutine had already completed with the corresponding exception when the handler is called. Normally, the handler is used to log the exception, show some kind of error message, terminate, and/or restart the application.

On JVM it is possible to redefine global exception handler for all coroutines by registering CoroutineExceptionHandler via ServiceLoader. Global exception handler is similar to Thread.defaultUncaughtExceptionHandler which is used when no more specific handlers are registered. On Android, uncaughtExceptionPreHandler is installed as a global coroutine exception handler.

`CoroutineExceptionHandler` is invoked only on uncaught exceptions — exceptions that were not handled in any other way. In particular, all children coroutines (coroutines created in the context of another `Job`) delegate handling of their exceptions to their parent coroutine, which also delegates to the parent, and so on until the root, so the `CoroutineExceptionHandler` installed in their context is never used. In addition to that, `async` builder always catches all exceptions and represents them in the resulting `Deferred` object, so its `CoroutineExceptionHandler` has no effect either.

Coroutines running in supervision scope do not propagate exceptions to their parent and are excluded from this rule. A further [Supervision](#) section of this document gives more details.

```
import kotlinx.coroutines.*  
  
@OptIn(DelicateCoroutinesApi::class)  
fun main() = runBlocking {  
    //sampleStart  
    val handler = CoroutineExceptionHandler { _, exception ->  
        println("CoroutineExceptionHandler got $exception")  
    }  
    val job = GlobalScope.launch(handler) { // root coroutine, running in GlobalScope  
        throw AssertionError()  
    }  
    val deferred = GlobalScope.async(handler) { // also root, but async instead of launch  
        throw ArithmeticException() // Nothing will be printed, relying on user to call  
    deferred.await()  
    }  
    joinAll(job, deferred)  
    //sampleEnd  
}
```

You can get the full code [here](#).

The output of this code is:

```
CoroutineExceptionHandler got java.lang.AssertionError
```

Cancellation and exceptions

Cancellation is closely related to exceptions. Coroutines internally use `CancellationException` for cancellation, these exceptions are ignored by all handlers, so they should be used only as the source of additional debug information, which can be obtained by catch block. When a coroutine is cancelled using `Job.cancel`, it terminates, but it does not cancel its parent.

```
import kotlinx.coroutines.*
```

```

fun main() = runBlocking {
//sampleStart
    val job = launch {
        val child = launch {
            try {
                delay(Long.MAX_VALUE)
            } finally {
                println("Child is cancelled")
            }
        }
        yield()
        println("Cancelling child")
        child.cancel()
        child.join()
        yield()
        println("Parent is not cancelled")
    }
    job.join()
//sampleEnd
}

```

You can get the full code [here](#).

The output of this code is:

```

Cancelling child
Child is cancelled
Parent is not cancelled

```

If a coroutine encounters an exception other than `CancellationException`, it cancels its parent with that exception. This behaviour cannot be overridden and is used to provide stable coroutines hierarchies for [structured concurrency](#). `CoroutineExceptionHandler` implementation is not used for child coroutines.

In these examples, `CoroutineExceptionHandler` is always installed to a coroutine that is created in `GlobalScope`. It does not make sense to install an exception handler to a coroutine that is launched in the scope of the main `runBlocking`, since the main coroutine is going to be always cancelled when its child completes with exception despite the installed handler.

The original exception is handled by the parent only when all its children terminate, which is demonstrated by the following example.

```

import kotlinx.coroutines.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->

```

```

        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) {
        launch { // the first child
            try {
                delay(Long.MAX_VALUE)
            } finally {
                withContext(NonCancellable) {
                    println("Children are cancelled, but exception is not handled until all
children terminate")
                    delay(100)
                    println("The first child finished its non cancellable block")
                }
            }
        }
        launch { // the second child
            delay(10)
            println("Second child throws an exception")
            throw ArithmeticException()
        }
    }
    job.join()
//sampleEnd
}

```

You can get the full code [here](#).

The output of this code is:

```

Second child throws an exception
Children are cancelled, but exception is not handled until all children terminate
The first child finished its non cancellable block
CoroutineExceptionHandler got java.lang.ArithmetricException

```

Exceptions aggregation

When multiple children of a coroutine fail with an exception, the general rule is "the first exception wins", so the first exception gets handled. All additional exceptions that happen after the first one are attached to the first exception as suppressed ones.

```

import kotlinx.coroutines.*
import java.io.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception with suppressed
${exception.suppressed.contentToString()}")
    }
}

```

```

val job = GlobalScope.launch(handler) {
    launch {
        try {
            delay(Long.MAX_VALUE) // it gets cancelled when another sibling fails with
IOException
        } finally {
            throw ArithmeticException() // the second exception
        }
    }
    launch {
        delay(100)
        throw IOException() // the first exception
    }
    delay(Long.MAX_VALUE)
}
job.join()
}

```

You can get the full code [here](#).

Note: This above code will work properly only on JDK7+ that supports suppressed exceptions

The output of this code is:

```

CoroutineExceptionHandler got java.io.IOException with suppressed
[java.lang.ArithmaticException]

```

Note that this mechanism currently only works on Java version 1.7+. The JS and Native restrictions are temporary and will be lifted in the future.

Cancellation exceptions are transparent and are unwrapped by default:

```

import kotlinx.coroutines.*
import java.io.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) {
        val inner = launch { // all this stack of coroutines will get cancelled
            launch {
                launch {
                    throw IOException() // the original exception
                }
            }
        }
    }
}

```

```

        }
    }
    try {
        inner.join()
    } catch (e: CancellationException) {
        println("Rethrowing CancellationException with original cause")
        throw e // cancellation exception is rethrown, yet the original IOException gets to
the handler
    }
}
job.join()
//sampleEnd
}

```

You can get the full code [here](#).

The output of this code is:

```

Rethrowing CancellationException with original cause
CoroutineExceptionHandler got java.io.IOException

```

Supervision

As we have studied before, cancellation is a bidirectional relationship propagating through the whole hierarchy of coroutines. Let us take a look at the case when unidirectional cancellation is required.

A good example of such a requirement is a UI component with the job defined in its scope. If any of the UI's child tasks have failed, it is not always necessary to cancel (effectively kill) the whole UI component, but if UI component is destroyed (and its job is cancelled), then it is necessary to fail all child jobs as their results are no longer needed.

Another example is a server process that spawns multiple child jobs and needs to supervise their execution, tracking their failures and only restarting the failed ones.

Supervision job

The `SupervisorJob` can be used for these purposes. It is similar to a regular `Job` with the only exception that cancellation is propagated only downwards. This can easily be demonstrated using the following example:

```

import kotlinx.coroutines.*

fun main() = runBlocking {
    val supervisor = SupervisorJob()
    with(CoroutineScope(coroutineContext + supervisor)) {
        // launch the first child -- its exception is ignored for this example (don't do this
in practice!)
        val firstChild = launch(CoroutineExceptionHandler { _, _ -> }) {
            println("The first child is failing")
        }
    }
}

```

```

        throw AssertionError("The first child is cancelled")
    }
    // launch the second child
    val secondChild = launch {
        firstChild.join()
        // Cancellation of the first child is not propagated to the second child
        println("The first child is cancelled: ${firstChild.isCancelled}, but the second
one is still active")
        try {
            delay(Long.MAX_VALUE)
        } finally {
            // But cancellation of the supervisor is propagated
            println("The second child is cancelled because the supervisor was cancelled")
        }
    }
    // wait until the first child fails & completes
    firstChild.join()
    println(" Cancelling the supervisor")
    supervisor.cancel()
    secondChild.join()
}
}

```

You can get the full code [here](#).

The output of this code is:

```

The first child is failing
The first child is cancelled: true, but the second one is still active
Cancelling the supervisor
The second child is cancelled because the supervisor was cancelled

```

Supervision scope

Instead of `coroutineScope`, we can use `supervisorScope` for scoped concurrency. It propagates the cancellation in one direction only and cancels all its children only if it failed itself. It also waits for all children before completion just like `coroutineScope` does.

```

import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
    try {
        supervisorScope {
            val child = launch {
                try {
                    println("The child is sleeping")
                    delay(Long.MAX_VALUE)
                } finally {
                    println("The child is cancelled")
                }
            }
        }
    }
}

```

```

        }
        // Give our child a chance to execute and print using yield
        yield()
        println("Throwing an exception from the scope")
        throw AssertionError()
    }
} catch(e: AssertionError) {
    println("Caught an assertion error")
}
}

```

You can get the full code [here](#).

The output of this code is:

```

The child is sleeping
Throwing an exception from the scope
The child is cancelled
Caught an assertion error

```

Exceptions in supervised coroutines

Another crucial difference between regular and supervisor jobs is exception handling. Every child should handle its exceptions by itself via the exception handling mechanism. This difference comes from the fact that child's failure does not propagate to the parent. It means that coroutines launched directly inside the `supervisorScope` do use the `CoroutineExceptionHandler` that is installed in their scope in the same way as root coroutines do (see the `CoroutineExceptionHandler` section for details).

```

import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    supervisorScope {
        val child = launch(handler) {
            println("The child throws an exception")
            throw AssertionError()
        }
        println("The scope is completing")
    }
    println("The scope is completed")
}

```

You can get the full code [here](#).

The output of this code is:

```
The scope is completing
The child throws an exception
CoroutineExceptionHandler got java.lang.AssertionError
The scope is completed
```

Shared mutable state and concurrency

Coroutines can be executed parallelly using a multi-threaded dispatcher like the `Dispatchers.Default`. It presents all the usual parallelism problems. The main problem being synchronization of access to shared mutable state. Some solutions to this problem in the land of coroutines are similar to the solutions in the multi-threaded world, but others are unique.

The problem

Let us launch a hundred coroutines all doing the same action thousand times. We'll also measure their completion time for further comparisons:

```
suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100 // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}
```

We start with a very simple action that increments a shared mutable variable using multi-threaded `Dispatchers.Default`.

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100 // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}
```

```

        }
    }
}
println("Completed ${n * k} actions in $time ms")
}

//sampleStart
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
//sampleEnd

```

You can get the full code [here](#).

What does it print at the end? It is highly unlikely to ever print "Counter = 100000", because a hundred coroutines increment the counter concurrently from multiple threads without any synchronization.

Volatiles are of no help

There is a common misconception that making a variable volatile solves concurrency problem. Let us try it:

```

import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100 // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
@Volatile // in Kotlin `volatile` is an annotation
var counter = 0

```

```

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
//sampleEnd

```

You can get the full code [here](#).

This code works slower, but we still don't get "Counter = 100000" at the end, because volatile variables guarantee linearizable (this is a technical term for "atomic") reads and writes to the corresponding variable, but do not provide atomicity of larger actions (increment in our case).

Thread-safe data structures

The general solution that works both for threads and for coroutines is to use a thread-safe (aka synchronized, linearizable, or atomic) data structure that provides all the necessary synchronization for the corresponding operations that needs to be performed on a shared state. In the case of a simple counter we can use AtomicInteger class which has atomic incrementAndGet operations:

```

import kotlinx.coroutines.*
import java.util.concurrent.atomic.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100 // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val counter = AtomicInteger()

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter.incrementAndGet()
        }
    }
}

```

```

    }
    println("Counter = $counter")
}
//sampleEnd

```

You can get the full code [here](#).

This is the fastest solution for this particular problem. It works for plain counters, collections, queues and other standard data structures and basic operations on them. However, it does not easily scale to complex state or to complex operations that do not have ready-to-use thread-safe implementations.

Thread confinement fine-grained

Thread confinement is an approach to the problem of shared mutable state where all access to the particular shared state is confined to a single thread. It is typically used in UI applications, where all UI state is confined to the single event-dispatch/application thread. It is easy to apply with coroutines by using a single-threaded context.

```

import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100 // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            // confine each increment to a single-threaded context
            withContext(counterContext) {
                counter++
            }
        }
    }
    println("Counter = $counter")
}

```

```
}
```

```
//sampleEnd
```

You can get the full code [here](#).

This code works very slowly, because it does fine-grained thread-confinement. Each individual increment switches from multi-threaded `Dispatchers.Default` context to the single-threaded context using `withContext(counterContext)` block.

Thread confinement coarse-grained

In practice, thread confinement is performed in large chunks, e.g. big pieces of state-updating business logic are confined to the single thread. The following example does it like that, running each coroutine in the single-threaded context to start with.

```
import kotlinx.coroutines.*
import kotlin.system.*
```

```
suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100 // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}
```

```
//sampleStart
val counterContext = newSingleThreadContext("CounterContext")
var counter = 0
```

```
fun main() = runBlocking {
    // confine everything to a single-threaded context
    withContext(counterContext) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

You can get the full code [here](#).

This now works much faster and produces correct result.

Mutual exclusion

Mutual exclusion solution to the problem is to protect all modifications of the shared state with a critical section that is never executed concurrently. In a blocking world you'd typically use synchronized or ReentrantLock for that.

Coroutine's alternative is called Mutex. It has lock and unlock functions to delimit a critical section. The key difference is that Mutex.lock() is a suspending function. It does not block a thread.

There is also withLock extension function that conveniently represents mutex.lock(); try { ... } finally { mutex.unlock() } pattern:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.sync.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100 // number of coroutines to launch
    val k = 1000 // times an action is repeated by each coroutine
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val mutex = Mutex()
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            // protect each increment with lock
            mutex.withLock {
                counter++
            }
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

You can get the full code [here](#).

The locking in this example is fine-grained, so it pays the price. However, it is a good choice for some situations where you absolutely must modify some shared state periodically, but there is no natural thread that this state is confined to.

Actors

An [actor](#) is an entity made up of a combination of a coroutine, the state that is confined and encapsulated into this coroutine, and a channel to communicate with other coroutines. A simple actor can be written as a function, but an actor with a complex state is better suited for a class.

There is an [actor](#) coroutine builder that conveniently combines actor's mailbox channel into its scope to receive messages from and combines the send channel into the resulting job object, so that a single reference to the actor can be carried around as its handle.

The first step of using an actor is to define a class of messages that an actor is going to process. Kotlin's [sealed classes](#) are well suited for that purpose. We define CounterMsg sealed class with IncCounter message to increment a counter and GetCounter message to get its value. The later needs to send a response. A [CompletableDeferred](#) communication primitive, that represents a single value that will be known (communicated) in the future, is used here for that purpose.

```
// Message types for counterActor
sealed class CounterMsg
object IncCounter : CounterMsg() // one-way message to increment counter
class GetCounter(val response: CompletableDeferred<Int>) : CounterMsg() // a request with reply
```

Then we define a function that launches an actor using an [actor](#) coroutine builder:

```
// This function launches a new counter actor
fun CoroutineScope.counterActor() = actor<CounterMsg> {
    var counter = 0 // actor state
    for (msg in channel) { // iterate over incoming messages
        when (msg) {
            is IncCounter -> counter++
            is GetCounter -> msg.response.complete(counter)
        }
    }
}
```

The main code is straightforward:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
```

```

val n = 100 // number of coroutines to launch
val k = 1000 // times an action is repeated by each coroutine
val time = measureTimeMillis {
    coroutineScope { // scope for coroutines
        repeat(n) {
            launch {
                repeat(k) { action() }
            }
        }
    }
}
println("Completed ${n * k} actions in $time ms")
}

// Message types for counterActor
sealed class CounterMsg
object IncCounter : CounterMsg() // one-way message to increment counter
class GetCounter(val response: CompletableDeferred<Int>) : CounterMsg() // a request with reply

// This function launches a new counter actor
fun CoroutineScope.counterActor() = actor<CounterMsg> {
    var counter = 0 // actor state
    for (msg in channel) { // iterate over incoming messages
        when (msg) {
            is IncCounter -> counter++
            is GetCounter -> msg.response.complete(counter)
        }
    }
}

//sampleStart
fun main() = runBlocking<Unit> {
    val counter = counterActor() // create the actor
    withContext(Dispatchers.Default) {
        massiveRun {
            counter.send(IncCounter)
        }
    }
    // send a message to get a counter value from an actor
    val response = CompletableDeferred<Int>()
    counter.send(GetCounter(response))
    println("Counter = ${response.await()}")
    counter.close() // shutdown the actor
}
//sampleEnd

```

You can get the full code [here](#).

It does not matter (for correctness) what context the actor itself is executed in. An actor is a coroutine and a coroutine is executed sequentially, so confinement of the state to the specific coroutine works as a solution to the problem of shared mutable state. Indeed, actors may modify their own private state, but can only affect each other through messages (avoiding the need for any locks).

Actor is more efficient than locking under load, because in this case it always has work to do and it does not have to

switch to a different context at all.

Note that an `actor` coroutine builder is a dual of `produce` coroutine builder. An actor is associated with the channel that it receives messages from, while a producer is associated with the channel that it sends elements to.

Select expression (experimental)

Select expression makes it possible to await multiple suspending functions simultaneously and select the first one that becomes available.

Select expressions are an experimental feature of `kotlinx.coroutines`. Their API is expected to evolve in the upcoming updates of the `kotlinx.coroutines` library with potentially breaking changes.

Selecting from channels

Let us have two producers of strings: `fizz` and `buzz`. The `fizz` produces "Fizz" string every 300 ms:

```
fun CoroutineScope.fizz() = produce<String> {
    while (true) { // sends "Fizz" every 300 ms
        delay(300)
        send("Fizz")
    }
}
```

And the `buzz` produces "Buzz!" string every 500 ms:

```
fun CoroutineScope.buzz() = produce<String> {
    while (true) { // sends "Buzz!" every 500 ms
        delay(500)
        send("Buzz!")
    }
}
```

Using `receive` suspending function we can receive either from one channel or the other. But `select` expression allows us to receive from both simultaneously using its `onReceive` clauses:

```
suspend fun selectFizzBuzz(fizz: ReceiveChannel<String>, buzz: ReceiveChannel<String>) {
    select<Unit> { // <Unit> means that this select expression does not produce any result
        fizz.onReceive { value -> // this is the first select clause
            println("fizz -> '$value'")
        }
    }
}
```

```

        buzz.onReceive { value -> // this is the second select clause
            println("buzz -> '$value'")
        }
    }
}

```

Let us run it all seven times:

```

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.fizz() = produce<String> {
    while (true) { // sends "Fizz" every 300 ms
        delay(300)
        send("Fizz")
    }
}

fun CoroutineScope.buzz() = produce<String> {
    while (true) { // sends "Buzz!" every 500 ms
        delay(500)
        send("Buzz!")
    }
}

suspend fun selectFizzBuzz(fizz: ReceiveChannel<String>, buzz: ReceiveChannel<String>) {
    select<Unit> { // <Unit> means that this select expression does not produce any result
        fizz.onReceive { value -> // this is the first select clause
            println("fizz -> '$value'")
        }
        buzz.onReceive { value -> // this is the second select clause
            println("buzz -> '$value'")
        }
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val fizz = fizz()
    val buzz = buzz()
    repeat(7) {
        selectFizzBuzz(fizz, buzz)
    }
    coroutineContext.cancelChildren() // cancel fizz & buzz coroutines
//sampleEnd
}

```

You can get the full code [here](#).

The result of this code is:

```
fizz -> 'Fizz'
```

```
buzz -> 'Buzz!'
fizz -> 'Fizz'
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
buzz -> 'Buzz!'
```

Selecting on close

The `onReceive` clause in `select` fails when the channel is closed causing the corresponding `select` to throw an exception. We can use `onReceiveCatching` clause to perform a specific action when the channel is closed. The following example also shows that `select` is an expression that returns the result of its selected clause:

```
suspend fun selectAorB(a: ReceiveChannel<String>, b: ReceiveChannel<String>): String =
    select<String> {
        a.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "a -> '$value'"
            } else {
                "Channel 'a' is closed"
            }
        }
        b.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "b -> '$value'"
            } else {
                "Channel 'b' is closed"
            }
        }
    }
```

Let's use it with channel a that produces "Hello" string four times and channel b that produces "World" four times:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

suspend fun selectAorB(a: ReceiveChannel<String>, b: ReceiveChannel<String>): String =
    select<String> {
        a.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "a -> '$value'"
            } else {
                "Channel 'a' is closed"
            }
        }
        b.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "b -> '$value'"
            } else {
                "Channel 'b' is closed"
            }
        }
    }
```

```

        } else {
            "Channel 'b' is closed"
        }
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val a = produce<String> {
        repeat(4) { send("Hello $it") }
    }
    val b = produce<String> {
        repeat(4) { send("World $it") }
    }
    repeat(8) { // print first eight results
        println(selectAorB(a, b))
    }
    coroutineContext.cancelChildren()
//sampleEnd
}

```

You can get the full code [here](#).

The result of this code is quite interesting, so we'll analyze it in more detail:

```

a -> 'Hello 0'
a -> 'Hello 1'
b -> 'World 0'
a -> 'Hello 2'
a -> 'Hello 3'
b -> 'World 1'
Channel 'a' is closed
Channel 'a' is closed

```

There are couple of observations to make out of it.

First of all, select is biased to the first clause. When several clauses are selectable at the same time, the first one among them gets selected. Here, both channels are constantly producing strings, so a channel, being the first clause in select, wins. However, because we are using unbuffered channel, the a gets suspended from time to time on its `send` invocation and gives a chance for b to send, too.

The second observation, is that `onReceiveCatching` gets immediately selected when the channel is already closed.

Selecting to send

Select expression has `onSend` clause that can be used for a great good in combination with a biased nature of selection.

Let us write an example of producer of integers that sends its values to a side channel when the consumers on its

primary channel cannot keep up with it:

```
fun CoroutineScope.produceNumbers(side: SendChannel<Int>) = produce<Int> {
    for (num in 1..10) { // produce 10 numbers from 1 to 10
        delay(100) // every 100 ms
        select<Unit> {
            onSend(num) {} // Send to the primary channel
            side.onSend(num) {} // or to the side channel
        }
    }
}
```

Consumer is going to be quite slow, taking 250 ms to process each number:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.produceNumbers(side: SendChannel<Int>) = produce<Int> {
    for (num in 1..10) { // produce 10 numbers from 1 to 10
        delay(100) // every 100 ms
        select<Unit> {
            onSend(num) {} // Send to the primary channel
            side.onSend(num) {} // or to the side channel
        }
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val side = Channel<Int>() // allocate side channel
    launch { // this is a very fast consumer for the side channel
        side.consumeEach { println("Side channel has $it") }
    }
    produceNumbers(side).consumeEach {
        println("Consuming $it")
        delay(250) // let us digest the consumed number properly, do not hurry
    }
    println("Done consuming")
    coroutineContext.cancelChildren()
//sampleEnd
}
```

You can get the full code [here](#).

So let us see what happens:

```
Consuming 1
Side channel has 2
Side channel has 3
Consuming 4
Side channel has 5
```

```
Side channel has 6
Consuming 7
Side channel has 8
Side channel has 9
Consuming 10
Done consuming
```

Selecting deferred values

Deferred values can be selected using `onAwait` clause. Let us start with an `async` function that returns a deferred string value after a random delay:

```
fun CoroutineScope.asyncString(time: Int) = async {
    delay(time.toLong())
    "Waited for $time ms"
}
```

Let us start a dozen of them with a random delay.

```
fun CoroutineScope.asyncStringsList(): List<Deferred<String>> {
    val random = Random(3)
    return List(12) { asyncString(random.nextInt(1000)) }
}
```

Now the main function awaits for the first of them to complete and counts the number of deferred values that are still active. Note that we've used here the fact that `select` expression is a Kotlin DSL, so we can provide clauses for it using an arbitrary code. In this case we iterate over a list of deferred values to provide `onAwait` clause for each deferred value.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.selects.*
import java.util.*

fun CoroutineScope.asyncString(time: Int) = async {
    delay(time.toLong())
    "Waited for $time ms"
}

fun CoroutineScope.asyncStringsList(): List<Deferred<String>> {
    val random = Random(3)
    return List(12) { asyncString(random.nextInt(1000)) }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val list = asyncStringsList()
    val result = select<String> {
        list.withIndex().forEach { (index, deferred) ->
            deferred.onAwait { answer ->
                "Deferred $index produced answer '$answer'"
            }
        }
    }
//sampleEnd
}
```

```

        }
    }
}

println(result)
val countActive = list.count { it.isActive }
println("$countActive coroutines are still active")
//sampleEnd
}

```

You can get the full code [here](#).

The output is:

```

Deferred 4 produced answer 'Waited for 128 ms'
11 coroutines are still active

```

Switch over a channel of deferred values

Let us write a channel producer function that consumes a channel of deferred string values, waits for each received deferred value, but only until the next deferred value comes over or the channel is closed. This example puts together `onReceiveCatching` and `onAwait` clauses in the same `select`:

```

fun CoroutineScope.switchMapDeferreds(input: ReceiveChannel<Deferred<String>>) =
produce<String> {
    var current = input.receive() // start with first received deferred value
    while (isActive) { // loop while not cancelled/closed
        val next = select<Deferred<String>?> { // return next deferred value from this select
            or null
            input.onReceiveCatching { update ->
                update.getOrNull()
            }
            current.onAwait { value ->
                send(value) // send value that current deferred has produced
                input.receiveCatching().getOrNull() // and use the next deferred from the
                input channel
            }
        }
        if (next == null) {
            println("Channel was closed")
            break // out of loop
        } else {
            current = next
        }
    }
}

```

To test it, we'll use a simple `async` function that resolves to a specified string after a specified time:

```

fun CoroutineScope.asyncString(str: String, time: Long) = async {
    delay(time)
    str
}

```

The main function just launches a coroutine to print results of switchMapDeferreds and sends some test data to it:

```

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.switchMapDeferreds(input: ReceiveChannel<Deferred<String>>) =
produce<String> {
    var current = input.receive() // start with first received deferred value
    while (isActive) { // loop while not cancelled/closed
        val next = select<Deferred<String>?> { // return next deferred value from this select
            or null
            input.onReceiveCatching { update ->
                update.getOrNull()
            }
            current.onAwait { value ->
                send(value) // send value that current deferred has produced
                input.receiveCatching().getOrNull() // and use the next deferred from the
                input channel
            }
        }
        if (next == null) {
            println("Channel was closed")
            break // out of loop
        } else {
            current = next
        }
    }
}

fun CoroutineScope.asyncString(str: String, time: Long) = async {
    delay(time)
    str
}

fun main() = runBlocking<Unit> {
//sampleStart
    val chan = Channel<Deferred<String>>() // the channel for test
    launch { // launch printing coroutine
        for (s in switchMapDeferreds(chan))
            println(s) // print each received string
    }
    chan.send(asyncString("BEGIN", 100))
    delay(200) // enough time for "BEGIN" to be produced
    chan.send(asyncString("Slow", 500))
    delay(100) // not enough time to produce slow
    chan.send(asyncString("Replace", 100))
    delay(500) // give it time before the last one
    chan.send(asyncString("END", 500))
    delay(1000) // give it time to process
    chan.close() // close the channel ...
    delay(500) // and wait some time to let it finish
}

```

```
//sampleEnd  
}
```

You can get the full code [here](#).

The result of this code:

```
BEGIN  
Replace  
END  
Channel was closed
```

Debug coroutines using IntelliJ IDEA – tutorial

This tutorial demonstrates how to create Kotlin coroutines and debug them using IntelliJ IDEA.

The tutorial assumes you have prior knowledge of the [coroutines](#) concept.

Debugging works for kotlinx-coroutines-core version 1.3.8 or later.

Create coroutines

1. Open a Kotlin project in IntelliJ IDEA. If you don't have a project, [create one](#).
2. Open the main.kt file in src/main/kotlin.

The src directory contains Kotlin source files and resources. The main.kt file contains sample code that will print Hello World!.

3. Change code in the main() function:

- Use the [runBlocking\(\)](#) block to wrap a coroutine.
- Use the [async\(\)](#) function to create coroutines that compute deferred values a and b.
- Use the [await\(\)](#) function to await the computation result.
- Use the [println\(\)](#) function to print computing status and the result of multiplication to the output.

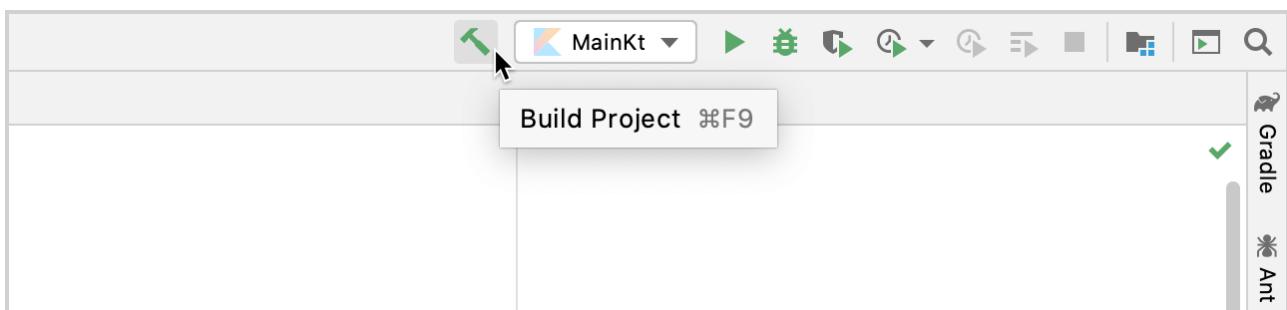
```
import kotlinx.coroutines.*  
  
fun main() = runBlocking<Unit> {
```

```

val a = async {
    println("I'm computing part of the answer")
    6
}
val b = async {
    println("I'm computing another part of the answer")
    7
}
    println("The answer is ${a.await() * b.await()}")
}

```

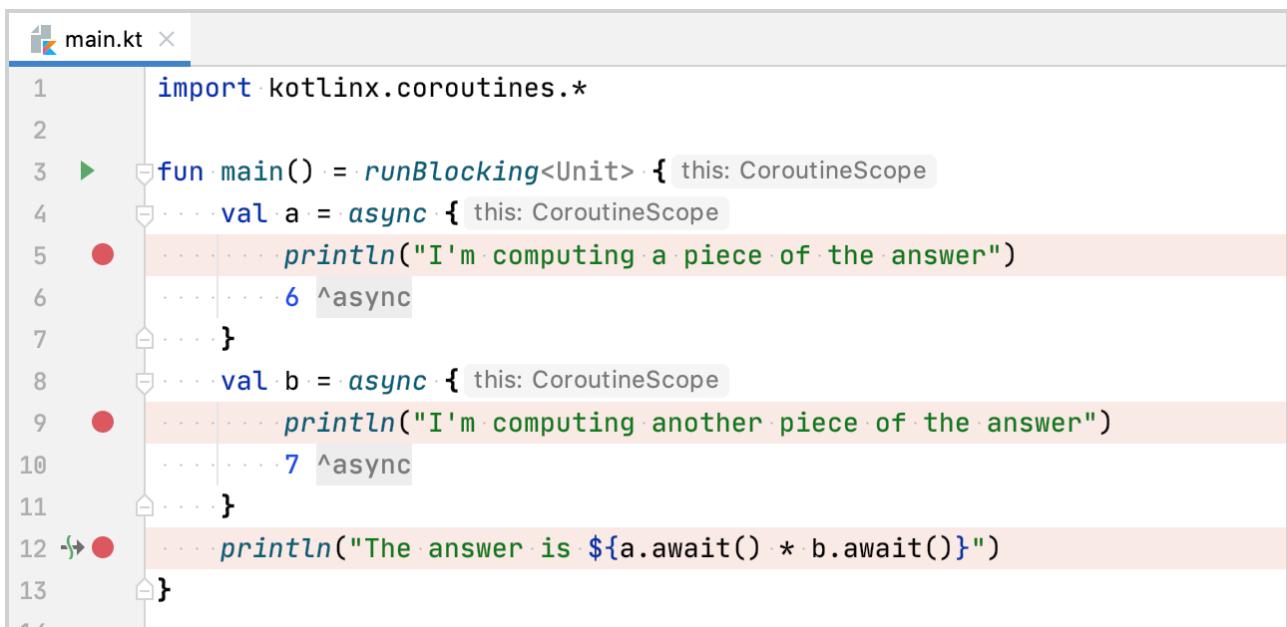
- Build the code by clicking Build Project.



Build an application

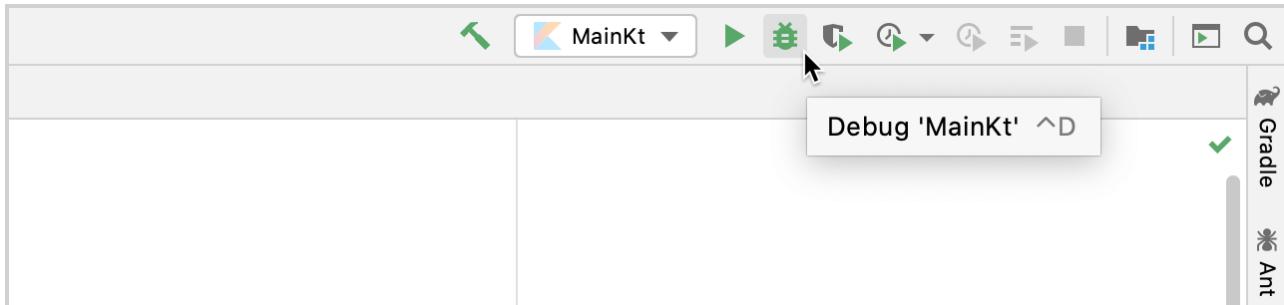
Debug coroutines

- Set breakpoints at the lines with the `println()` function call:



Build a console application

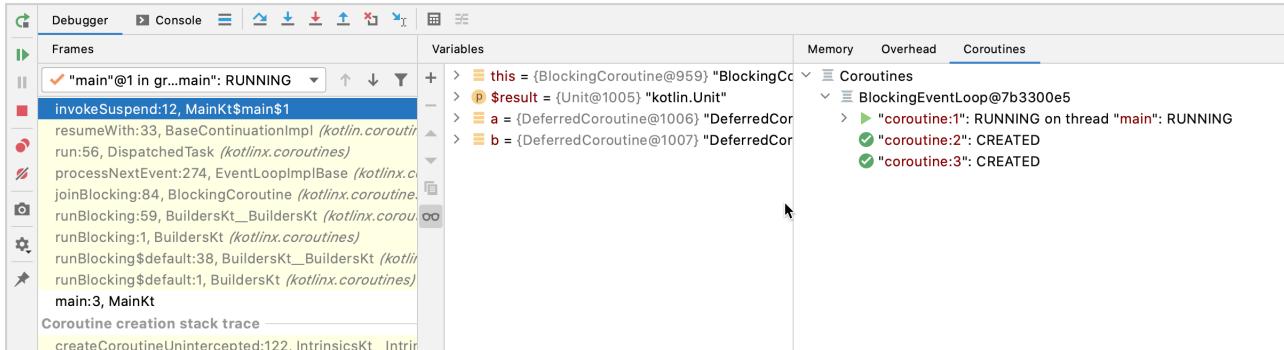
2. Run the code in debug mode by clicking Debug next to the run configuration at the top of the screen.



Build a console application

The Debug tool window appears:

- The Frames tab contains the call stack.
- The Variables tab contains variables in the current context.
- The Coroutines tab contains information on running or suspended coroutines. It shows that there are three coroutines. The first one has the RUNNING status, and the other two have the CREATED status.



Debug the coroutine

3. Resume the debugger session by clicking Resume program in the Debug tool window:

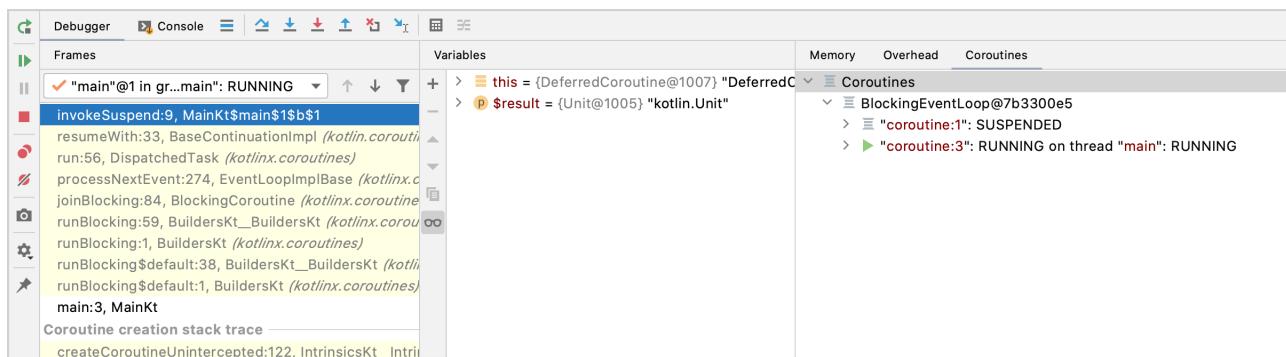


Debug the coroutine

Now the Coroutines tab shows the following:

- The first coroutine has the SUSPENDED status – it is waiting for the values so it can multiply them.
- The second coroutine is calculating the a value – it has the RUNNING status.
- The third coroutine has the CREATED status and isn't calculating the value of b.

4. Resume the debugger session by clicking Resume program in the Debug tool window:



Build a console application

Now the Coroutines tab shows the following:

- The first coroutine has the SUSPENDED status – it is waiting for the values so it can multiply them.
- The second coroutine has computed its value and disappeared.
- The third coroutine is calculating the value of b – it has the RUNNING status.

Using IntelliJ IDEA debugger, you can dig deeper into each coroutine to debug your code.

Debug Kotlin Flow using IntelliJ IDEA – tutorial

This tutorial demonstrates how to create Kotlin Flow and debug it using IntelliJ IDEA.

The tutorial assumes you have prior knowledge of the [coroutines](#) and [Kotlin Flow](#) concepts.

Debugging works for kotlinx-coroutines-core version 1.3.8 or later.

Create a Kotlin flow

Create a Kotlin flow with a slow emitter and a slow collector:

1. Open a Kotlin project in IntelliJ IDEA. If you don't have a project, [create one](#).
2. Open the main.kt file in src/main/kotlin.

The src directory contains Kotlin source files and resources. The main.kt file contains sample code that will print Hello World!.

3. Create the simple() function that returns a flow of three numbers:

- Use the delay() function to imitate CPU-consuming blocking code. It suspends the coroutine for 100 ms without blocking the thread.
- Produce the values in the for loop using the emit() function.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

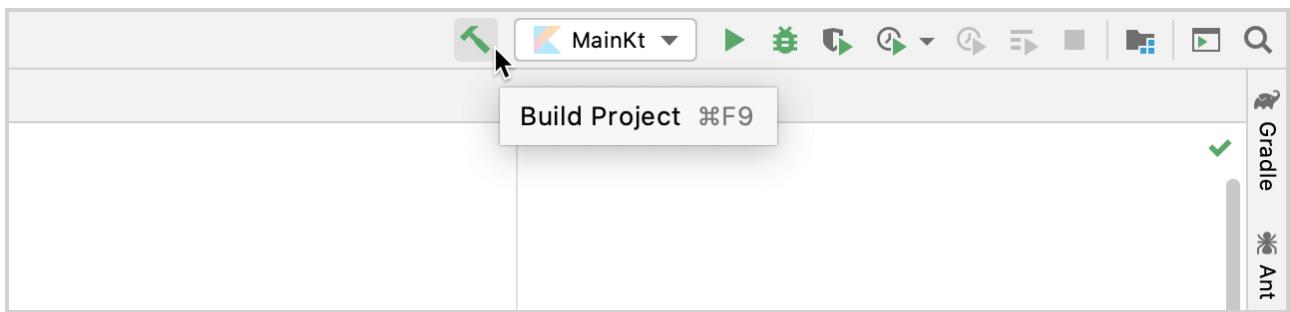
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}
```

4. Change the code in the main() function:

- Use the runBlocking() block to wrap a coroutine.
- Collect the emitted values using the collect() function.
- Use the delay() function to imitate CPU-consuming code. It suspends the coroutine for 300 ms without blocking the thread.
- Print the collected value from the flow using the println() function.

```
fun main() = runBlocking {
    simple()
        .collect { value ->
            delay(300)
            println(value)
        }
}
```

5. Build the code by clicking Build Project.



Build an application

Debug the coroutine

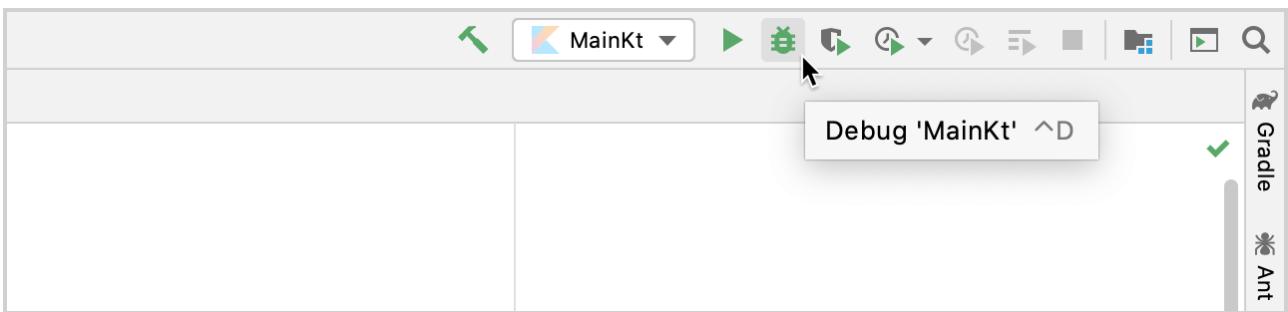
1. Set a breakpoint at the at the line where the emit() function is called:

A screenshot of the main.kt code editor in Android Studio. The code uses Kotlin Coroutines. A red circular breakpoint is set on the line 'emit(i)' in the 'simple()' function. The code is as follows:

```
1 import kotlinx.coroutines.*  
2 import kotlinx.coroutines.flow.*  
3  
4 fun simple(): Flow<Int> = flow { this: FlowCollector<Int>  
5     for (i in 1..3) {  
6         delay( timeMillis: 100)  
7         emit(i)  
8     }  
9 }  
10  
11 fun main() = runBlocking { this: CoroutineScope  
12     simple()  
13         .collect { value ->  
14             delay( timeMillis: 300)  
15             println(value)  
16         }  
17 }  
18
```

Build a console application

2. Run the code in debug mode by clicking Debug next to the run configuration at the top of the screen.



Build a console application

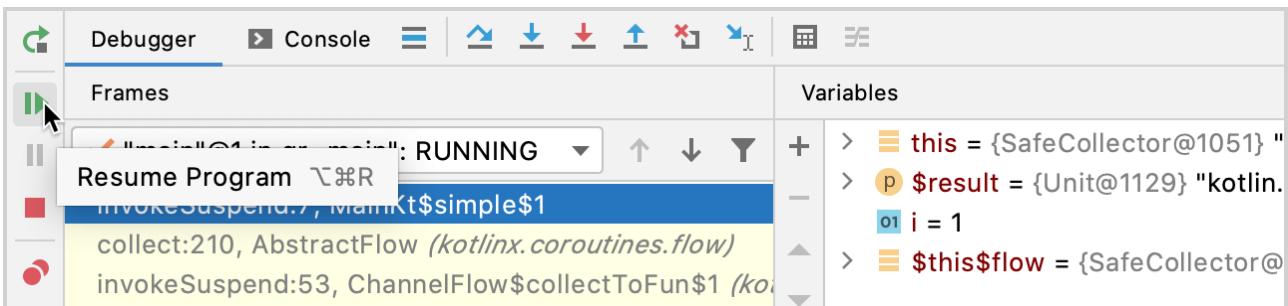
The Debug tool window appears:

- The Frames tab contains the call stack.
- The Variables tab contains variables in the current context. It tells us that the flow is emitting the first value.
- The Coroutines tab contains information on running or suspended coroutines.



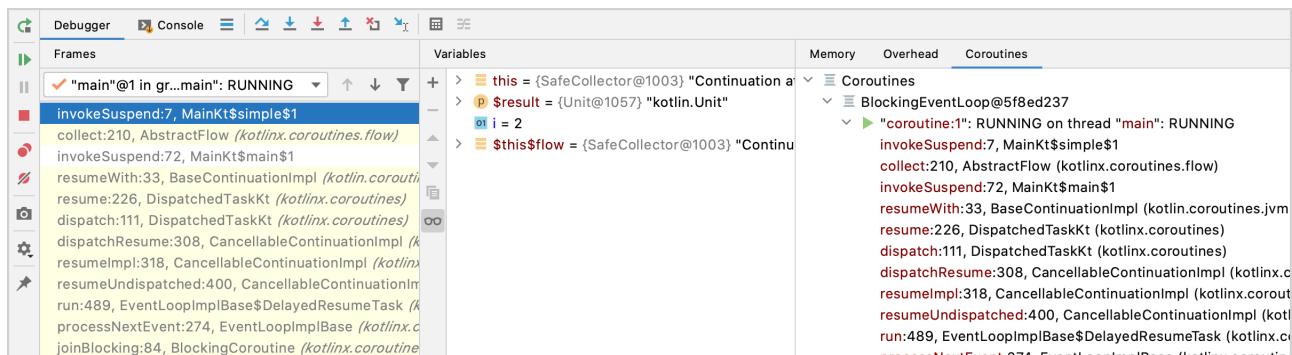
Debug the coroutine

3. Resume the debugger session by clicking Resume program in the Debug tool window. The program stops at the same breakpoint.



Debug the coroutine

Now the flow emits the second value.



Debug the coroutine

Add a concurrently running coroutine

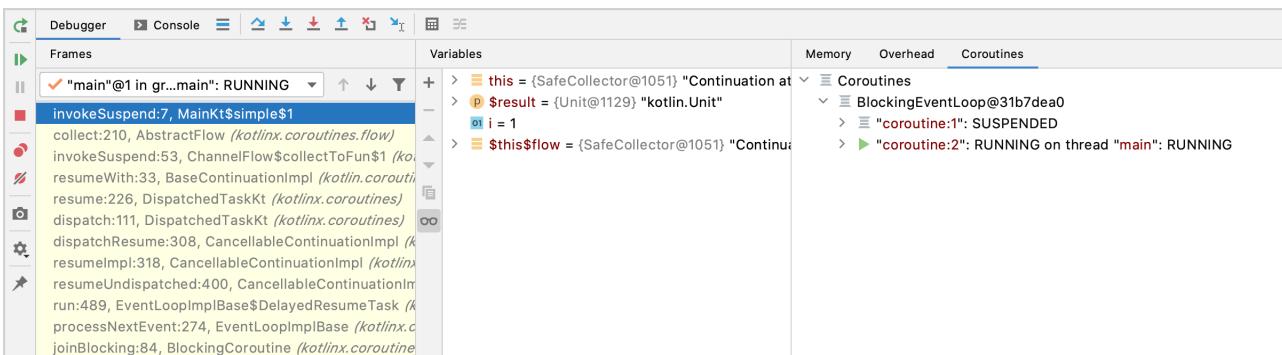
1. Open the main.kt file in src/main/kotlin.
2. Enhance the code to run the emitter and collector concurrently:
 - Add a call to the `buffer()` function to run the emitter and collector concurrently. `buffer()` stores emitted values and runs the flow collector in a separate coroutine.

```
fun main() = runBlocking<Unit> {
    simple()
        .buffer()
        .collect { value ->
            delay(300)
            println(value)
        }
}
```

3. Build the code by clicking Build Project.

Debug a Kotlin flow with two coroutines

1. Set a new breakpoint at `println(value)`.
2. Run the code in debug mode by clicking Debug next to the run configuration at the top of the screen.

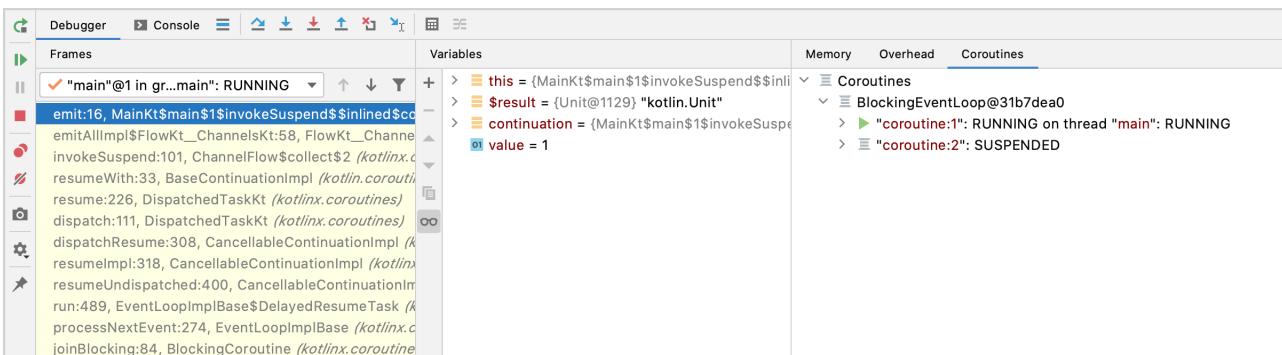


Build a console application

The Debug tool window appears.

In the Coroutines tab, you can see that there are two coroutines running concurrently. The flow collector and emitter run in separate coroutines because of the `buffer()` function. The `buffer()` function buffers emitted values from the flow. The emitter coroutine has the RUNNING status, and the collector coroutine has the SUSPENDED status.

- Resume the debugger session by clicking Resume program in the Debug tool window.



Debugging coroutines

Now the collector coroutine has the RUNNING status, while the emitter coroutine has the SUSPENDED status.

You can dig deeper into each coroutine to debug your code.

Serialization

Serialization is the process of converting data used by an application to a format that can be transferred over a network or stored in a database or a file. In turn, deserialization is the opposite process of reading data from an external source and converting it into a runtime object. Together they are an essential part of most applications that exchange data with third parties.

Some data serialization formats, such as [JSON](#) and [protocol buffers](#) are particularly common. Being language-neutral

and platform-neutral, they enable data exchange between systems written in any modern language.

In Kotlin, data serialization tools are available in a separate component, [kotlinx.serialization](#). It consists of two main parts: the Gradle plugin – org.jetbrains.kotlin.plugin.serialization and the runtime libraries.

Libraries

kotlinx.serialization provides sets of libraries for all supported platforms – JVM, JavaScript, Native – and for various serialization formats – JSON, CBOR, protocol buffers, and others. You can find the complete list of supported serialization formats [below](#).

All Kotlin serialization libraries belong to the org.jetbrains.kotlinx: group. Their names start with kotlinx-serialization- and have suffixes that reflect the serialization format. Examples:

- org.jetbrains.kotlinx:kotlinx-serialization-json provides JSON serialization for Kotlin projects.
- org.jetbrains.kotlinx:kotlinx-serialization-cbor provides CBOR serialization.

Platform-specific artifacts are handled automatically; you don't need to add them manually. Use the same dependencies in JVM, JS, Native, and multiplatform projects.

Note that the kotlinx.serialization libraries use their own versioning structure, which doesn't match Kotlin's versioning. Check out the releases on [GitHub](#) to find the latest versions.

Formats

kotlinx.serialization includes libraries for various serialization formats:

- [JSON: kotlinx-serialization-json](#)
- [Protocol buffers: kotlinx-serialization-protobuf](#)
- [CBOR: kotlinx-serialization-cbor](#)
- [Properties: kotlinx-serialization-properties](#)
- [HOCON: kotlinx-serialization-hocon](#) (only on JVM)

Note that all libraries except JSON serialization (kotlinx-serialization-core) are [Experimental](#), which means their API can be changed without notice.

There are also community-maintained libraries that support more serialization formats, such as [YAML](#) or [Apache Avro](#). For detailed information about available serialization formats, see the [kotlinx.serialization documentation](#).

Example: JSON serialization

Let's take a look at how to serialize Kotlin objects into JSON.

Before starting, you'll need to configure your build script so that you can use Kotlin serialization tools in your project:

1. Apply the Kotlin serialization Gradle plugin `org.jetbrains.kotlin.plugin.serialization` (or `kotlin("plugin.serialization")`) in the Kotlin Gradle DSL).

Groovy

```
plugins {  
    id 'org.jetbrains.kotlin.jvm' version '1.5.20'  
    id 'org.jetbrains.kotlin.plugin.serialization' version '1.5.20'  
}
```

Kotlin

```
plugins {  
    kotlin("jvm") version "1.5.20"  
    kotlin("plugin.serialization") version "1.5.20"  
}
```

2. Add the JSON serialization library dependency: `org.jetbrains.kotlinx:kotlinx-serialization-json:1.2.1`

Groovy

```
dependencies {  
    implementation 'org.jetbrains.kotlinx:kotlinx-serialization-json:1.2.1'  
}
```

Kotlin

```
dependencies {  
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.2.1")  
}
```

Now you're ready to use the serialization API in your code. The API is located in the the `kotlinx.serialization` package and its format-specific subpackages such as `kotlinx.serialization.json`.

First, make a class serializable by annotating it with `@Serializable`.

```
@Serializable  
data class Data(val a: Int, val b: String)
```

You can now serialize an instance of this class by calling `Json.encodeToString()`.

```
Json.encodeToString(Data(42, "str"))
```

As a result, you get a string containing the state of this object in the JSON format: {"a": 42, "b": "str"}

You can also serialize object collections, such as lists, in a single call.

```
val dataList = listOf(Data(42, "str"), Data(12, "test"))
val jsonList = Json.encodeToString(dataList)
```

To deserialize an object from JSON, use the `decodeFromString()` function:

```
val obj = Json.decodeFromString<Data>("""{"a":42, "b": "str"}""")
```

For more information about serialization in Kotlin, see the [Kotlin Serialization Guide](#).

Keywords and operators

Hard keywords

The following tokens are always interpreted as keywords and cannot be used as identifiers:

- `as`
 - is used for [type casts](#)
 - specifies an [alias for an import](#)
- `as?` is used for [safe type casts](#)
- `break` [terminates the execution of a loop](#)
- `class` declares a [class](#)
- `continue` [proceeds to the next step of the nearest enclosing loop](#)
- `do` begins a [do/while loop](#) (loop with postcondition)
- `else` defines the branch of an [if expression](#) which is executed when the condition is false
- `false` specifies the 'false' value of the [Boolean type](#)
- `for` begins a [for loop](#)
- `fun` declares a [function](#)

- if begins an if expression
- in
 - specifies the object being iterated in a for loop
 - is used as an infix operator to check that a value belongs to a range, a collection or another entity that defines the 'contains' method
 - is used in when expressions for the same purpose
 - marks a type parameter as contravariant
- !in
 - is used as an operator to check that a value does NOT belong to a range, a collection or another entity that defines the 'contains' method
 - is used in when expressions for the same purpose
- interface declares an interface
- is
 - checks that a value has a certain type
 - is used in when expressions for the same purpose
- !is
 - checks that a value does NOT have a certain type
 - is used in when expressions for the same purpose
- null is a constant representing an object reference that doesn't point to any object
- object declares a class and its instance at the same time
- package specifies the package for the current file
- return returns from the nearest enclosing function or anonymous function
- super
 - refers to the superclass implementation of a method or property
 - calls the superclass constructor from a secondary constructor
- this
 - refers to the current receiver
 - calls another constructor of the same class from a secondary constructor
- throw throws an exception
- true specifies the 'true' value of the Boolean type

- try begins an exception handling block
- typealias declares a type alias
- typeof reserved for future use
- val declares a read-only property or local variable
- var declares a mutable property or local variable
- when begins a when expression (executes one of the given branches)
- while begins a while loop (loop with precondition)

Soft keywords

The following tokens act as keywords in the context when they are applicable and can be used as identifiers in other contexts:

- by
 - delegates the implementation of an interface to another object
 - delegates the implementation of accessors for a property to another object
- catch begins a block that handles a specific exception type
- constructor declares a primary or secondary constructor
- delegate is used as an annotation use-site target
- dynamic references a dynamic type in Kotlin/JS code
- field is used as an annotation use-site target
- file is used as an annotation use-site target
- finally begins a block that is always executed when a try block exits
- get
 - declares the getter of a property
 - is used as an annotation use-site target
- import imports a declaration from another package into the current file
- init begins an initializer block
- param is used as an annotation use-site target
- property is used as an annotation use-site target

- receiver is used as an [annotation use-site target](#)
- set
 - declares the [setter of a property](#)
 - is used as an [annotation use-site target](#)
- setparam is used as an [annotation use-site target](#)
- value with the class keyword declares an [inline class](#)
- where specifies [constraints for a generic type parameter](#)

Modifier keywords

The following tokens act as keywords in modifier lists of declarations and can be used as identifiers in other contexts:

- actual denotes a platform-specific implementation in [multiplatform projects](#)
- abstract marks a class or member as [abstract](#)
- annotation declares an [annotation class](#)
- companion declares a [companion object](#)
- const marks a property as a [compile-time constant](#)
- crossinline forbids [non-local returns in a lambda passed to an inline function](#)
- data instructs the compiler to [generate canonical members for a class](#)
- enum declares an [enumeration](#)
- expect marks a declaration as [platform-specific](#), expecting an implementation in platform modules.
- external marks a declaration as implemented not in Kotlin (accessible through [JNI](#) or in [JavaScript](#))
- final forbids [overriding a member](#)
- infix allows calling a function in [infix notation](#)
- inline tells the compiler to [inline the function and the lambdas passed to it at the call site](#)
- inner allows referring to the outer class instance from a [nested class](#)
- internal marks a declaration as [visible in the current module](#)
- lateinit allows initializing a [non-null property outside of a constructor](#)
- noinline turns off [inlining of a lambda passed to an inline function](#)

- open allows subclassing a class or overriding a member
- operator marks a function as overloading an operator or implementing a convention
- out marks a type parameter as covariant
- override marks a member as an override of a superclass member
- private marks a declaration as visible in the current class or file
- protected marks a declaration as visible in the current class and its subclasses
- public marks a declaration as visible anywhere
- reified marks a type parameter of an inline function as accessible at runtime
- sealed declares a sealed class (a class with restricted subclassing)
- suspend marks a function or lambda as suspending (usable as a coroutine)
- tailrec marks a function as tail-recursive (allowing the compiler to replace recursion with iteration)
- vararg allows passing a variable number of arguments for a parameter

Special identifiers

The following identifiers are defined by the compiler in specific contexts and can be used as regular identifiers in other contexts:

- field is used inside a property accessor to refer to the backing field of the property
- it is used inside a lambda to refer to its parameter implicitly

Operators and special symbols

Kotlin supports the following operators and special symbols:

- +, -, *, /, %- mathematical operators
 - * is also used to pass an array to a vararg parameter
- =
 - assignment operator
 - is used to specify default values for parameters
- +=, -=, *=, /=, %=- augmented assignment operators
- ++, --- increment and decrement operators

- &&, ||, !- logical 'and', 'or', 'not' operators (for bitwise operations, use corresponding [infix functions](#))
- ==, !=- [equality operators](#) (translated to calls of equals() for non-primitive types)
- ===, !==- [referential equality operators](#)
- <, >, <=, >=- [comparison operators](#) (translated to calls of compareTo() for non-primitive types)
- [,]- [indexed access operator](#) (translated to calls of get and set)
- !! [asserts that an expression is non-null](#)
- ?. performs a [safe call](#) (calls a method or accesses a property if the receiver is non-null)
- ?: takes the right-hand value if the left-hand value is null (the [elvis operator](#))
- :: creates a [member reference](#) or a [class reference](#)
- .. creates a [range](#)
- : separates a name from a type in declarations
- ? marks a type as [nullable](#)
- ->
 - separates the parameters and body of a [lambda expression](#)
 - separates the parameters and return type declaration in a [function type](#)
 - separates the condition and body of a [when expression](#) branch
- @
 - introduces an [annotation](#)
 - introduces or references a [loop label](#)
 - introduces or references a [lambda label](#)
 - references a 'this' expression from an outer scope
 - references an [outer superclass](#)
- ; separates multiple statements on the same line
- \$ references a variable or expression in a [string template](#)
- -
 - substitutes an unused parameter in a [lambda expression](#)
 - substitutes an unused parameter in a [destructuring declaration](#)

Gradle

In order to build a Kotlin project with Gradle, you should [apply the Kotlin Gradle plugin to your project](#) and [configure dependencies](#).

Plugin and versions

Apply the Kotlin Gradle plugin by using [the Gradle plugins DSL](#).

The Kotlin Gradle plugin and kotlin-multiplatform plugin 1.5.20 require Gradle 6.1 or later.

Groovy

```
plugins {  
    id 'org.jetbrains.kotlin.<...>' version '1.5.20'  
}
```

Kotlin

```
plugins {  
    kotlin("<...>") version "1.5.20"  
}
```

The placeholder <...> should be replaced with one of the plugin names that can be found in further sections.

Targeting multiple platforms

Projects targeting [multiple platforms](#), called [multiplatform projects](#), require the kotlin-multiplatform plugin. [Learn more about the plugin](#).

The kotlin-multiplatform plugin works with Gradle 6.1 or later.

Groovy

```
plugins {  
    id 'org.jetbrains.kotlin.multiplatform' version '1.5.20'  
}
```

Kotlin

```
plugins {  
    kotlin("multiplatform") version "1.5.20"  
}
```

Targeting the JVM

To target the JVM, apply the Kotlin JVM plugin.

Groovy

```
plugins {  
    id "org.jetbrains.kotlin.jvm" version "1.5.20"  
}
```

Kotlin

```
plugins {  
    kotlin("jvm") version "1.5.20"  
}
```

The version should be literal in this block, and it cannot be applied from another build script.

Alternatively, you can use the older apply plugin approach:

```
apply plugin: 'kotlin'
```

It's not recommended that you apply Kotlin plugins with apply in Gradle Kotlin DSL – [see why](#).

Kotlin and Java sources

Kotlin sources can be stored with Java sources in the same folder, or placed to different folders. The default convention is using different folders:

```
project  
    - src  
        - main (root)  
            - kotlin  
            - java
```

The corresponding sourceSets property should be updated if not using the default convention:

Groovy

```
sourceSets {  
    main.kotlin.srcDirs += 'src/main/myKotlin'  
    main.java.srcDirs += 'src/main/myJava'  
}
```

Kotlin

```
sourceSets.main {  
    java.srcDirs("src/main/myJava", "src/main/myKotlin")  
}
```

Targeting JavaScript

When targeting only JavaScript, use the `kotlin-js` plugin. [Learn more](#)

Groovy

```
plugins {  
    id 'org.jetbrains.kotlin.js' version '1.5.20'  
}
```

Kotlin

```
plugins {  
    kotlin("js") version "1.5.20"  
}
```

Kotlin and Java sources for JavaScript

This plugin only works for Kotlin files so it is recommended that you keep Kotlin and Java files separately (in case the project contains Java files). If you don't store them separately, specify the source folder in the `sourceSets` block:

Groovy

```
kotlin {  
    sourceSets {  
        main.kotlin.srcDirs += 'src/main/myKotlin'  
    }  
}
```

Kotlin

```
kotlin {
```

```
sourceSets["main"].apply {
    kotlin.srcDir("src/main/myKotlin")
}
}
```

Targeting Android

It's recommended that you use Android Studio for creating Android applications. [Learn how to use Android Gradle plugin.](#)

Configuring dependencies

To add a dependency on a library, set the dependency of the required `type` (for example, `implementation`) in the `dependencies` block of the source sets DSL.

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'com.example:my-library:1.0'
            }
        }
    }
}
```

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("com.example:my-library:1.0")
            }
        }
    }
}
```

Alternatively, you can [set dependencies at the top level](#).

Dependency types

Choose the dependency type based on your requirements.

Type	Description	When to use
api	Used both during compilation and at runtime and is exported to library consumers.	If any type from a dependency is used in the public API of the current module, use an api dependency.
implementation	Used during compilation and at runtime for the current module, but is not exposed for compilation of other modules depending on the one with the `implementation` dependency.	Use for dependencies needed for the internal logic of a module. If a module is an endpoint application which is not published, use implementation dependencies instead of api dependencies.
compileOnly	Used for compilation of the current module and is not available at runtime nor during compilation of other modules.	Use for APIs which have a third-party implementation available at runtime.
runtimeOnly	Available at runtime but is not visible during compilation of any module.	

Dependency on the standard library

A dependency on a standard library (stdlib) in each source set is added automatically. The version of the standard library is the same as the version of the Kotlin Gradle plugin.

For platform-specific source sets, the corresponding platform-specific variant of the library is used, while a common standard library is added to the rest. The Kotlin Gradle plugin will select the appropriate JVM standard library depending on the `kotlinOptions.jvmTarget compiler` option of your Gradle build script.

If you declare a standard library dependency explicitly (for example, if you need a different version), the Kotlin Gradle plugin won't override it or add a second standard library.

If you do not need a standard library at all, you can add the opt-out flag to the `gradle.properties`:

```
kotlin.stdlib.default.dependency=false
```

Set dependencies on test libraries

The `kotlin.test` API is available to test Kotlin projects on all supported platforms. Add the dependency `kotlin-test` to the `commonTest` source set, and the Gradle plugin will infer the corresponding test dependencies for each test source set:

- `kotlin-test-common` and `kotlin-test-annotations-common` for common source sets
- `kotlin-test-junit` for JVM source sets

- kotlin-test-js for Kotlin/JS source sets

Kotlin/Native targets do not require additional test dependencies, and the kotlin.test API implementations are built-in.

Groovy

```
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // This brings all the platform dependencies
automatically
            }
        }
    }
}
```

Kotlin

```
kotlin {
    sourceSets {
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test")) // This brings all the platform dependencies
automatically
            }
        }
    }
}
```

You can use shorthand for a dependency on a Kotlin module, for example, kotlin("test") for "org.jetbrains.kotlin:kotlin-test".

You can use the kotlin-test dependency in any shared or platform-specific source set as well.

For Kotlin/JVM, Gradle uses JUnit 4 by default. Therefore, the kotlin("test") dependency resolves to the variant for JUnit 4, namely kotlin-test-junit.

You can choose JUnit 5 or TestNG by calling [useJUnitPlatform\(\)](#) or [useTestNG\(\)](#) in the test task of your build script. The following example is for an MPP project:

Groovy

```
kotlin {
    jvm {
        testRuns["test"].executionTask.configure {
            useJUnitPlatform()
        }
    }
}
```

```
        }
    }
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test")
            }
        }
    }
}
```

Kotlin

```
kotlin {
    jvm {
        testRuns["test"].executionTask.configure {
            useJUnitPlatform()
        }
    }
    sourceSets {
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test"))
            }
        }
    }
}
```

The following example is for a JVM project:

Groovy

```
dependencies {
    testImplementation 'org.jetbrains.kotlin:kotlin-test'
}

test {
    useTestNG()
}
```

Kotlin

```
dependencies {
    testImplementation(kotlin("test"))
}

tasks {
    test {
        useTestNG()
    }
}
```

[Learn how to test code using JUnit on the JVM.](#)

If you need to use a different JVM test framework, disable automatic testing framework selection by adding the line `kotlin.test.infer.jvm.variant=false` to the project's `gradle.properties`. Once you do this, add the framework as a Gradle dependency.

If you had used a variant of `kotlin("test")` in your build script explicitly and project build stopped working with a conflict on capability, see [this issue in the Compatibility Guide](#).

Set a dependency on a kotlinx library

If you use a kotlinx library and need a platform-specific dependency, you can use platform-specific variants of libraries with suffixes such as `-jvm` or `-js`, for example, `kotlinx-coroutines-core-jvm`. You can also use the library base artifact name instead – `kotlinx-coroutines-core`.

Groovy

```
kotlin {  
    sourceSets {  
        jvmMain {  
            dependencies {  
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.5.0'  
            }  
        }  
    }  
}
```

Kotlin

```
kotlin {  
    sourceSets {  
        val jvmMain by getting {  
            dependencies {  
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.5.0")  
            }  
        }  
    }  
}
```

If you use a multiplatform library and need to depend on the shared code, set the dependency only once in the shared source set. Use the library base artifact name, such as `kotlinx-coroutines-core` or `ktor-client-core`.

Groovy

```
kotlin {  
    sourceSets {  
        commonMain {  
            dependencies {  
                implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:  
            }  
        }  
    }  
}
```

```
        dependencies {
            implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.0'
        }
    }
}
```

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.0")
            }
        }
    }
}
```

Set dependencies at the top level

Alternatively, you can specify the dependencies at the top level with the configuration names following the pattern <sourceSetName><DependencyType>. This is helpful for some Gradle built-in dependencies, like gradleApi(), localGroovy(), or gradleTestKit(), which are not available in the source sets dependency DSL.

Groovy

```
dependencies {
    commonMainImplementation 'com.example:my-library:1.0'
}
```

Kotlin

```
dependencies {
    "commonMainImplementation"("com.example:my-library:1.0")
}
```

Annotation processing

Kotlin supports annotation processing via the Kotlin annotation processing tool [kapt](#).

Incremental compilation

The Kotlin Gradle plugin supports incremental compilation. Incremental compilation tracks changes of source files

between builds so only files affected by these changes would be compiled.

Incremental compilation is supported for Kotlin/JVM and Kotlin/JS projects and is enabled by default.

There are several ways to switch off incremental compilation:

- * `kotlin.incremental=false` for Kotlin/JVM
- * `kotlin.incremental.js=false` for Kotlin/JS projects
- As the command line parameter, use `-Pkotlin.incremental=false` or `-Pkotlin.incremental.js=false`.

Note that in this case the parameter should be added to each subsequent build, and any build with disabled incremental compilation invalidates incremental caches.

Note that the first build isn't incremental in any case.

Gradle build cache support

The Kotlin plugin supports [Gradle build cache](#) that stores the build outputs for reuse in future builds.

To disable the caching for all Kotlin tasks, set the system property flag `kotlin.caching.enabled` to `false` (run the build with the argument `-Dkotlin.caching.enabled=false`).

If you use [kapt](#), note that the kapt annotation processing tasks are not cached by default. However, you can [enable caching for them manually](#).

Gradle configuration cache support

The configuration cache is available in Gradle 6.5 and later as an experimental feature. You can check the [Gradle releases page](#) to see whether it has been promoted to stable.

The Kotlin plugin supports [Gradle configuration cache](#), which speeds up the build process by reusing the results of the configuration phase.

See the [Gradle documentation](#) to learn how to enable the configuration cache. Once you enable the configuration cache feature, the Kotlin Gradle plugin will start using it.

Compiler options

To specify additional compilation options, use the `kotlinOptions` property of a Kotlin compilation task.

When targeting the JVM, the tasks are called `compileKotlin` for production code and `compileTestKotlin` for test code. The tasks for custom source sets are called accordingly to the `compile<Name>Kotlin` pattern.

The names of the tasks in Android Projects contain the [build variant](#) names and follow the pattern

`compile<BuildVariant>Kotlin`, for example, `compileDebugKotlin`, `compileReleaseUnitTestKotlin`.

When targeting JavaScript, the tasks are called `compileKotlinJs` and `compileTestKotlinJs` respectively, and `compile<Name>KotlinJs` for custom source sets.

To configure a single task, use its name. Examples:

Groovy

```
compileKotlin {  
    kotlinOptions.suppressWarnings = true  
}  
  
// or  
  
compileKotlin {  
    kotlinOptions {  
        suppressWarnings = true  
    }  
}
```

Kotlin

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile  
// ...  
  
val compileKotlin: KotlinCompile by tasks  
  
compileKotlin.kotlinOptions.suppressWarnings = true
```

Note that with Gradle Kotlin DSL, you should get the task from the project's tasks first.

Use the types `Kotlin2JsCompile` and `KotlinCompileCommon` for the JS and Common targets, accordingly.

It is also possible to configure all Kotlin compilation tasks in the project:

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompile).configureEach {  
    kotlinOptions { /*...*/ }  
}
```

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompile>().configureEach {  
    kotlinOptions { /*...*/ }  
}
```

The complete list of options for the Gradle tasks is the following:

Attributes common for JVM, JS, and JS DCE

Name	Description	Possible values	Default value
allWarningsAsErrors	Report an error if there are any warnings	false	
suppressWarnings	Generate no warnings	false	
verbose	Enable verbose logging output	false	
freeCompilerArgs	A list of additional compiler arguments	[]	

Attributes common for JVM and JS

Name	Description	Possible values	Default value
apiVersion	Allow using declarations only from the specified version of bundled libraries	"1.3" (DEPRECATED), "1.4", "1.5", "1.6" (EXPERIMENTAL)	
languageVersion	Provide source compatibility with the specified version of Kotlin	"1.3" (DEPRECATED), "1.4", "1.5", "1.6" (EXPERIMENTAL)	

Attributes specific for JVM

Name	Description	Possible values	Default value
javaParameters	Generate metadata for Java 1.8 reflection on method parameters		false
jdkHome	Include a custom JDK from the specified location into the classpath instead of the default JAVA_HOME		
jvmTarget	Target version of the generated JVM bytecode	"1.6" (DEPRECATED), "1.8", "9", "10", "11", "12", "13", "14", "15", "16"	"1.8"
noJdk	Don't automatically include the Java runtime into the classpath		false
useOldBackend	Use the old JVM backend		false

Attributes specific for JS

Name	Description	Possible values	Default value
friendModulesDisabled	Disable internal declaration export		false
main	Define whether the main function should be called upon execution	"call", "noCall"	"call"
metaInfo	Generate .meta.js and .kjsm files with metadata. Use to create a library		true
moduleKind	The kind of JS module generated by the compiler	"umd", "commonjs", "amd", "plain"	"umd"
noStdlib	Don't automatically include the default Kotlin/JS stdlib into compilation dependencies		true
outputFile	Destination *.js file for the compilation result	"<buildDir>/js/packages/<project.name>/kotlin/<project.name>.js"	
sourceMap	Generate source map		true
sourceMapEmbedSources	Embed source files into source map	"never", "always", "inlining"	"never"

Name	Description	Possible values	Default value
sourceMapPrefix	Add the specified prefix to paths in the source map		
target	Generate JS files for specific ECMA version	"v5"	"v5"
typedArrays	Translate primitive arrays to JS typed arrays	true	

Generating documentation

To generate documentation for Kotlin projects, use [Dokka](#); please refer to the [Dokka README](#) for configuration instructions. Dokka supports mixed-language projects and can generate output in multiple formats, including standard JavaDoc.

OSGi

For OSGi support see the [Kotlin OSGi page](#).

Using Gradle Kotlin DSL

When using [Gradle Kotlin DSL](#), apply the Kotlin plugins using the plugins { ... } block. If you apply them with apply { plugin(...) } instead, you may encounter unresolved references to the extensions generated by Gradle Kotlin DSL. To resolve that, you can comment out the erroneous usages, run the Gradle task kotlinDslAccessorsSnapshot, then uncomment the usages back and rerun the build or reimport the project into the IDE.

Maven

Plugin and versions

The kotlin-maven-plugin compiles Kotlin sources and modules. Currently, only Maven v3 is supported.

Define the version of Kotlin you want to use via a `kotlin.version` property:

```
<properties>
    <kotlin.version>1.5.20</kotlin.version>
</properties>
```

Dependencies

Kotlin has an extensive standard library that can be used in your applications. To use the standard library in your project, add the following dependency in the pom file:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

If you're targeting JDK 7 or JDK 8, you can use extended versions of the Kotlin standard library. They contain additional extension functions for APIs added in new JDK versions. Instead of `kotlin-stdlib`, use `kotlin-stdlib-jdk7` or `kotlin-stdlib-jdk8`, depending on your JDK version.

For Kotlin versions older than 1.2, use `kotlin-stdlib-jre7` and `kotlin-stdlib-jre8`.

If your project uses [Kotlin reflection](#) or testing facilities, you need to add the corresponding dependencies as well. The artifact IDs are `kotlin-reflect` for the reflection library, and `kotlin-test` and `kotlin-test-junit` for the testing libraries.

Compile Kotlin-only source code

To compile source code, specify the source directories in the `<build>` tag:

```
<build>
    <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
    <testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirectory>
</build>
```

The Kotlin Maven Plugin needs to be referenced to compile the sources:

```
<build>
    <plugins>
```

```

<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>

    <executions>
        <execution>
            <id>compile</id>
            <goals>
                <goal>compile</goal>
            </goals>
        </execution>

        <execution>
            <id>test-compile</id>
            <goals>
                <goal>test-compile</goal>
            </goals>
        </execution>
    </executions>
</plugin>
</plugins>
</build>

```

Compile Kotlin and Java sources

To compile projects that include Kotlin and Java source code, invoke the Kotlin compiler before the Java compiler. In maven terms that means that kotlin-maven-plugin should be run before maven-compiler-plugin using the following method, making sure that the kotlin plugin comes before the maven-compiler-plugin in your pom.xml file:

```

<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <executions>
                <execution>
                    <id>compile</id>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                    <configuration>
                        <sourceDirs>
                            <sourceDir>${project.basedir}/src/main/kotlin</sourceDir>
                            <sourceDir>${project.basedir}/src/main/java</sourceDir>
                        </sourceDirs>
                    </configuration>
                </execution>
                <execution>
                    <id>test-compile</id>
                    <goals> <goal>test-compile</goal> </goals>
                    <configuration>
                        <sourceDirs>
                            <sourceDir>${project.basedir}/src/test/kotlin</sourceDir>

```

```

        <sourceDir>${project.basedir}/src/test/java</sourceDir>
    </sourceDirs>
    </configuration>
</execution>
</executions>
</plugin>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.5.1</version>
    <executions>
        <!-- Replacing default-compile as it is treated specially by maven -->
        <execution>
            <id>default-compile</id>
            <phase>none</phase>
        </execution>
        <!-- Replacing default-testCompile as it is treated specially by maven -->
        <execution>
            <id>default-testCompile</id>
            <phase>none</phase>
        </execution>
        <execution>
            <id>java-compile</id>
            <phase>compile</phase>
            <goals>
                <goal>compile</goal>
            </goals>
        </execution>
        <execution>
            <id>java-test-compile</id>
            <phase>test-compile</phase>
            <goals>
                <goal>testCompile</goal>
            </goals>
        </execution>
    </executions>
</plugin>
</plugins>
</build>

```

Incremental compilation

To make your builds faster, you can enable incremental compilation for Maven by defining the `kotlin.compiler.incremental` property:

```

<properties>
    <kotlin.compiler.incremental>true</kotlin.compiler.incremental>
</properties>

```

Alternatively, run your build with the `-Dkotlin.compiler.incremental=true` option.

Annotation processing

See the description of [Kotlin annotation processing tool \(kapt\)](#).

Jar file

To create a small Jar file containing just the code from your module, include the following under build->plugins in your Maven pom.xml file, where main.class is defined as a property and points to the main Kotlin or Java class:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.6</version>
  <configuration>
    <archive>
      <manifest>
        <addClasspath>true</addClasspath>
        <mainClass>${main.class}</mainClass>
      </manifest>
    </archive>
  </configuration>
</plugin>
```

Self-contained Jar file

To create a self-contained Jar file containing the code from your module along with dependencies, include the following under build->plugins in your Maven pom.xml file, where main.class is defined as a property and points to the main Kotlin or Java class:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.6</version>
  <executions>
    <execution>
      <id>make-assembly</id>
      <phase>package</phase>
      <goals> <goal>single</goal> </goals>
      <configuration>
        <archive>
          <manifest>
            <mainClass>${main.class}</mainClass>
          </manifest>
        </archive>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </execution>
  </executions>
```

```
</plugin>
```

This self-contained jar file can be passed directly to a JRE to run your application:

```
java -jar target/mymodule-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

Specifying compiler options

Additional options and arguments for the compiler can be specified as tags under the `<configuration>` element of the Maven plugin node:

```
<plugin>
  <groupId>org.jetbrains.kotlin</groupId>
  <artifactId>kotlin-maven-plugin</artifactId>
  <version>${kotlin.version}</version>
  <executions>...</executions>
  <configuration>
    <nowarn>true</nowarn> <!-- Disable warnings -->
    <args>
      <arg>-Xjsr305=strict</arg> <!-- Enable strict mode for JSR-305 annotations -->
      ...
    </args>
  </configuration>
</plugin>
```

Many of the options can also be configured through properties:

```
<project ...>
  <properties>
    <kotlin.compiler.languageVersion>1.0</kotlin.compiler.languageVersion>
  </properties>
</project>
```

The following attributes are supported:

Attributes common for JVM and JS

Name	Property name	Description	Possible values	Default value
nowarn		Generate no warnings	true, false	false
languageVersion	kotlin.compiler.languageVersion	Provide source compatibility with the specified version of Kotlin	"1.2" (DEPRECATED), "1.3", "1.4", "1.5" (EXPERIMENTAL)	

Name	Property name	Description	Possible values	Default value
apiVersion	kotlin.compiler.apiVersion	Allow using declarations only from the specified version of bundled libraries	"1.2" (DEPRECATED), "1.3", "1.4", "1.5" (EXPERIMENTAL)	
sourceDirs		The directories containing the source files to compile		The project source roots
compilerPlugins		Enabled compiler plugins	[]	
pluginOptions		Options for compiler plugins	[]	
args		Additional compiler arguments	[]	

Attributes specific for JVM

Name	Property name	Description	Possible values	Default value
jvmTarget	kotlin.compiler.jvmTarget	Target version of the generated JVM bytecode	"1.6" (DEPRECATED), "1.8", "9", "10", "11", "12", "13", "14", "15", "16"	"1.8"
jdkHome	kotlin.compiler.jdkHome	Include a custom JDK from the specified location into the classpath instead of the default JAVA_HOME		

Attributes specific for JS

Name	Property name	Description	Possible values	Default value
outputFile		Destination *.js file for the compilation result		
metalInfo		Generate .meta.js and .kjsm files with metadata. Use to create a library	true, false	true
sourceMap		Generate source map	true, false	false
sourceMapEmbedSources		Embed source files into source map	"never", "always", "inlining"	"inlining"

Name	Property name	Description	Possible values	Default value
sourceMapPrefix		Add the specified prefix to paths in the source map		
moduleKind		The kind of JS module generated by the compiler	"umd", "commonjs", "amd", "plain"	"umd"

Generating documentation

The standard JavaDoc generation plugin (`maven-javadoc-plugin`) does not support Kotlin code. To generate documentation for Kotlin projects, use [Dokka](#); please refer to the [Dokka README](#) for configuration instructions. Dokka supports mixed-language projects and can generate output in multiple formats, including standard JavaDoc.

OSGi

For OSGi support see the [Kotlin OSGi page](#).

Examples

An example Maven project can be [downloaded from the GitHub repository](#).

Ant

Getting the Ant tasks

Kotlin provides three tasks for Ant:

- `kotlinc`: Kotlin compiler targeting the JVM
- `kotlin2js`: Kotlin compiler targeting JavaScript
- `withKotlin`: Task to compile Kotlin files when using the standard `javac` Ant task

These tasks are defined in the `kotlin-ant.jar` library which is located in the `lib` folder in the [Kotlin Compiler](#) archive. Ant version 1.8.2+ is required.

Targeting JVM with Kotlin-only source

When the project consists of exclusively Kotlin source code, the easiest way to compile the project is to use the `kotlinc` task:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-
ant.jar"/>

    <target name="build">
        <kotlinc src="hello.kt" output="hello.jar"/>
    </target>
</project>
```

where \${kotlin.lib} points to the folder where the Kotlin standalone compiler was unzipped.

Targeting JVM with Kotlin-only source and multiple roots

If a project consists of multiple source roots, use `src` as elements to define paths:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-
ant.jar"/>

    <target name="build">
        <kotlinc output="hello.jar">
            <src path="root1"/>
            <src path="root2"/>
        </kotlinc>
    </target>
</project>
```

Targeting JVM with Kotlin and Java source

If a project consists of both Kotlin and Java source code, while it is possible to use `kotlinc`, to avoid repetition of task parameters, it is recommended to use `withKotlin` task:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-
ant.jar"/>

    <target name="build">
        <delete dir="classes" failonerror="false"/>
        <mkdir dir="classes"/>
        <javac destdir="classes" includeAntRuntime="false" srcdir="src">
            <withKotlin/>
        </javac>
        <jar destfile="hello.jar">
```

```

        <fileset dir="classes" />
    </jar>
</target>
</project>
```

You can also specify the name of the module being compiled as the moduleName attribute:

```
<withKotlin moduleName="myModule" />
```

Targeting JavaScript with single source folder

```

<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-
ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js"/>
    </target>
</project>
```

Targeting JavaScript with Prefix, PostFix and sourcemap options

```

<project name="Ant Task Test" default="build">
    <taskdef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-
ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js" outputPrefix="prefix" outputPostfix="postfix"
sourcemap="true"/>
    </target>
</project>
```

Targeting JavaScript with single source folder and metaInfo option

The metaInfo option is useful, if you want to distribute the result of translation as a Kotlin/JavaScript library. If metaInfo was set to true, then during compilation additional JS file with binary metadata will be created. This file should be distributed together with the result of translation:

```

<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-
ant.jar"/>

    <target name="build">
```

```

<!-- out.meta.js will be created, which contains binary metadata -->
<kotlin2js src="root1" output="out.js" metaInfo="true"/>
</target>
</project>

```

References

Complete list of elements and attributes are listed below:

Attributes common for kotlinc and kotlin2js

Name	Description	Required	Default Value
src	Kotlin source file or directory to compile	Yes	
nowarn	Suppresses all compilation warnings	No	false
noStdlib	Does not include the Kotlin standard library into the classpath	No	false
failOnError	Fails the build if errors are detected during the compilation	No	true

kotlinc attributes

Name	Description	Required	Default Value
output	Destination directory or .jar file name	Yes	
classpath	Compilation class path	No	
classpathref	Compilation class path reference	No	
includeRuntime	If output is a .jar file, whether Kotlin runtime library is included in the jar	No	true
moduleName	Name of the module being compiled	No	The name of the target (if specified) or the project

kotlin2js attributes

Name	Description	Required

Name	Description	Required
output	Destination file	Yes
libraries	Paths to Kotlin libraries	No
outputPrefix	Prefix to use for generated JavaScript files	No
outputSuffix	Suffix to use for generated JavaScript files	No
sourcemap	Whether sourcemap file should be generated	No
metaInfo	Whether metadata file with binary descriptors should be generated	No
main	Should compiler generated code call the main function	No

Passing raw compiler arguments

To pass custom raw compiler arguments, you can use `<compilerarg>` elements with either value or line attributes. This can be done within the `<kotlinc>`, `<kotlin2js>`, and `<withKotlin>` task elements, as follows:

```
<kotlinc src="${test.data}/hello.kt" output="${temp}/hello.jar">
  <compilerarg value="-Xno-inline"/>
  <compilerarg line="-Xno-call-assertions -Xno-param-assertions"/>
  <compilerarg value="-Xno-optimize"/>
</kotlinc>
```

The full list of arguments that can be used is shown when you run `kotlinc -help`.

Run code snippets – tutorial

There may be times when you need to quickly write and execute code outside of a project or application. This may be useful, for example, when learning Kotlin or evaluating expressions. Let's have a look at three handy ways we can use to run Kotlin code quickly:

- Scratches let us write and run code in a temporary file outside of our project in the IDE.
- Worksheets are like scratches, but they reside within projects.
- REPL (Read-Eval-Print-Loop) runs code in an interactive console.

Scratches and worksheets

The Kotlin plugin for IntelliJ IDEA supports [scratches](#) and [worksheets](#).

Scratches let you create code drafts in the same IDE window with your project and run them on the fly. Scratches are not tied to projects; you can access and run all your scratches from any IntelliJ IDEA window on your OS.

To create a Kotlin scratch, click File | New | Scratch file and select the Kotlin type.

In turn, worksheets are project files: they are stored in project directories and tied to the project modules. Worksheets are useful for writing code parts that don't actually make a software unit but should still be stored together in a project. For example, you can use worksheets for education or demo materials.

To create a Kotlin worksheet in a project directory, right-click the directory in the project tree and select New | Kotlin Worksheet.

In scratches and worksheets, you can write any valid Kotlin code. Syntax highlighting, auto-completion, and the other IntelliJ IDEA code editing features are all supported too. Note that there's no need for declaring the main function: all the code you write is executed as if it would be in the body of main.

Once you have finished writing your code in a scratch or a worksheet, click Run. The execution results will appear in the lines opposite your code.

```
1 val x = 1
2 val y = 2
3
4 var i = x + y
5 while (i++ < 8)
6     println(i)
7
8 println("Hello world!")
```

Line	Code	Output
1	val x: Int	
2	val y: Int	
3		
4	var i: Int	
5	4	
6	5	
7	6	
8	7	
9	8	
10		
11	Hello world!	

Run scratch

Interactive mode

IntelliJ IDEA can run the code from scratches and worksheets automatically. To get the execution results when you stop typing, switch on the Interactive mode.

```
scratch_2.kts x scratch_3.kts x
Use classpath of module <no module>  Interactive mode  Use REPL
1 val x = 1
2 val y = 2
3
4 val z = x + y
5 >> |
6
7
```

Scratch interactive mode

Using modules

We can use classes or functions from a Kotlin project, in our scratches and worksheets.

Worksheets automatically get access to classes and functions from the module where they reside.

To use classes or functions from a project in a scratch, import them into the scratch file as usual with the `import` statement. Then write your code and run it with the appropriate module selected in `Use classpath of module` list.

Both scratches and worksheets use the compiled versions of connected modules. So, if you modify a module's source files, the changes will propagate to scratches and worksheets when you rebuild the module. To rebuild the module automatically before each run of a scratch or a worksheet, select `Make before Run`.

```
Main.kt x scratch_1.kts x DeepThought.kt x
Use classpath of module myproject-main <no module>  Make module before Run  Interactive mode  Use REPL
1 import org.sample.my<br/>
2 <no module> <br/>
3 myproject-main
4 1 + 5
5 >> DeepThought().answerUltimateQuestion()
6 <br/>
```

Scratch select module

Running as REPL

To evaluate each particular expression in a scratch or a worksheet, run it with `Use REPL` selected. The code will be executed the same way as in [REPL](#): the code lines will run sequentially, providing results of each call. You can later refer to the results by the names `res*` shown in the corresponding lines.

The screenshot shows the IntelliJ IDEA interface with three tabs at the top: Main.kt, scratch_1.kts (selected), and DeepThought.kt. Below the tabs, there are several buttons: a green play button, a trash can, 'Use classpath of module' (checkbox), 'myproject-main' (dropdown), 'Make module before Run' (checkbox), 'Interactive mode' (checkbox), and 'Use REPL' (checkbox, which is checked and highlighted with a red box). The code editor contains the following Kotlin code:

```
1 import org.sample.mypackage.DeepThought
2
3 1 + 5
4 >> DeepThought().answerUltimateQuestion()
```

To the right of the code editor, the results are displayed in two columns:

	1	2	3	4
res1:	kotlin.Int = 6			
res2:			kotlin.Int = 42	

Scratch REPL

REPL

REPL (Read-Eval-Print-Loop) is a tool for running Kotlin code interactively. REPL lets you evaluate expressions and code chunks without creating projects or even functions if you don't need them.

To run REPL in IntelliJ IDEA, open Tools | Kotlin | Kotlin REPL.

To run REPL in the OS command line, open /bin/kotlinc-jvm from the directory of standalone Kotlin compiler.

The REPL command line interface will open. You can enter any valid Kotlin code and see the result. The results are printed as variables with auto-generated names like res*. You can later use such variables in the code you run in REPL.

The screenshot shows the 'Run' tool window in IntelliJ IDEA with the title 'Kotlin REPL (in module org.sample.myproject.main)'. The left sidebar has icons for back, forward, equals, and close. The main area shows the following REPL session:

```
1 + 3
res0: kotlin.Int = 4

res0 * 3
res1: kotlin.Int = 12

res0 + res1
res2: kotlin.Int = 16
```

A tooltip at the bottom left says '<Ctrl+Enter> to execute'. At the bottom of the window, there are tabs for 'Run' (selected), 'TODO', and 'Terminal'.

Run REPL

REPL supports multiline input as well. The result of the multiline input is the value of its last expression.

The screenshot shows the IntelliJ IDEA interface with the 'Run' tool window open. The title bar says 'Run: Kotlin REPL (in module org.sample.myproject)'. The main area displays a multi-line run session:

```
Welcome to Kotlin version 1.3.20-eap-25 (JRE 1.8.0_192-b12)
Type :help for help, :quit for quit

fun foo( i: Int) = i * 2

foo(2)
foo(3)
foo(4)
res1: kotlin.Int = 8
<Ctrl+Enter> to execute
```

The left sidebar has icons for Run, Stop, and Close. At the bottom of the window are tabs for 'Run' (selected), 'TODO', and 'Terminal'.

REPL multi-line run

Migrate to Kotlin code style

Kotlin coding conventions and IntelliJ IDEA formatter

[Kotlin coding conventions](#) affect several aspects of writing idiomatic Kotlin, and a set of formatting recommendations aimed at improving Kotlin code readability is among them.

Unfortunately, the code formatter built into IntelliJ IDEA had to work long before this document was released and now has a default setup that produces different formatting from what is now recommended.

It may seem a logical next step to remove this obscurity by switching the defaults in IntelliJ IDEA and make formatting consistent with the Kotlin coding conventions. But this would mean that all the existing Kotlin projects will have a new code style enabled the moment the Kotlin plugin is installed. Not really the expected result for plugin update, right?

That's why we have the following migration plan instead:

- Enable the official code style formatting by default starting from Kotlin 1.3 and only for new projects (old formatting can be enabled manually)
- Authors of existing projects may choose to migrate to the Kotlin coding conventions
- Authors of existing projects may choose to explicitly declare using the old code style in a project (this way the project won't be affected by switching to the defaults in the future)
- Switch to the default formatting and make it consistent with Kotlin coding conventions in Kotlin 1.4

Differences between "Kotlin coding conventions" and "IntelliJ IDEA default code style"

The most notable change is in the continuation indentation policy. There's a nice idea to use the double indent for showing that a multi-line expression hasn't ended on the previous line. This is a very simple and general rule, but several Kotlin constructions look a bit awkward when they are formatted this way. In Kotlin coding conventions, it's recommended to use a single indent in cases where the long continuation indent has been forced before.



Code formatting

In practice, quite a bit of code is affected, so this can be considered a major code style update.

Migration to a new code style discussion

A new code style adoption might be a very natural process if it starts with a new project, when there's no code formatted in the old way. That is why starting from version 1.3, the Kotlin IntelliJ Plugin creates new projects with formatting from the [Coding conventions](#) document which is enabled by default.

Changing formatting in an existing project is a far more demanding task, and should probably be started with discussing all the caveats with the team.

The main disadvantage of changing the code style in an existing project is that the blame/annotate VCS feature will point to irrelevant commits more often. While each VCS has some kind of way to deal with this problem ("Annotate Previous Revision" can be used in IntelliJ IDEA), it's important to decide if a new style is worth all the effort. The practice of separating reformatting commits from meaningful changes can help a lot with later investigations.

Also migrating can be harder for larger teams because committing a lot of files in several subsystems may produce merging conflicts in personal branches. And while each conflict resolution is usually trivial, it's still wise to know if there are large feature branches currently in work.

In general, for small projects, we recommend converting all the files at once.

For medium and large projects the decision may be tough. If you are not ready to update many files right away you may decide to migrate module by module, or continue with gradual migration for modified files only.

Migration to a new code style

Switching to the Kotlin Coding Conventions code style can be done in Settings | Editor | Code Style | Kotlin dialog.

Switch scheme to Project and activate Set from... | Predefined Style | Kotlin Style Guide.

In order to share those changes for all project developers .idea/codeStyle folder have to be committed to VCS.

If an external build system is used for configuring the project, and it's been decided not to share .idea/codeStyle folder, Kotlin coding conventions can be forced with an additional property:

In Gradle

Add kotlin.code.style=official property to the gradle.properties file at the project root and commit the file to VCS.

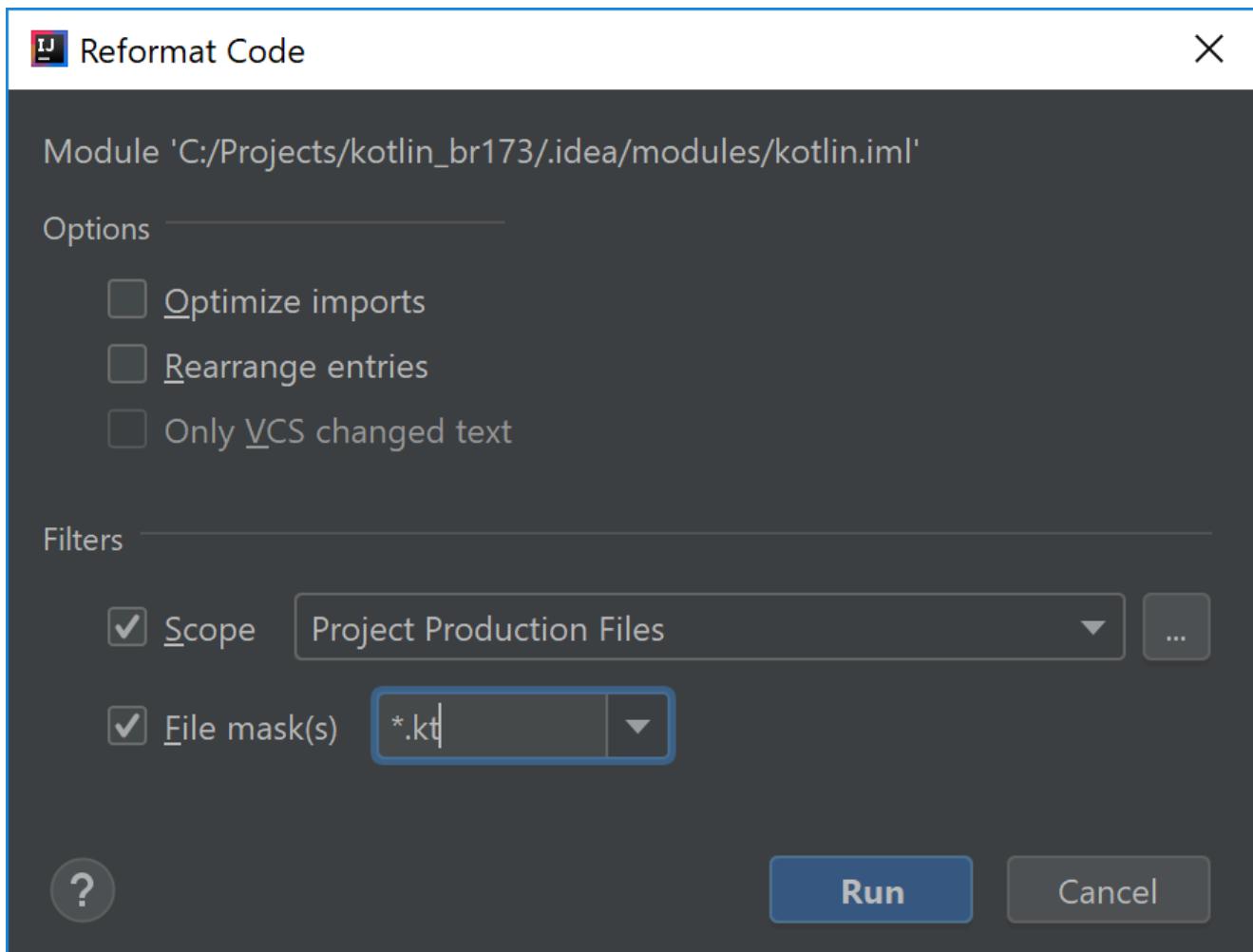
In Maven

Add kotlin.code.style official property to root pom.xml project file.

```
<properties> <kotlin.code.style>official</kotlin.code.style> </properties>
```

Having the kotlin.code.style option set may modify the code style scheme during a project import and may change the code style settings.

After updating your code style settings, activate Reformat Code in the project view on the desired scope.



Reformat code

For a gradual migration, it's possible to enable the File is not formatted according to project settings inspection. It will highlight the places that should be reformatted. After enabling the Apply only to modified files option, inspection will show formatting problems only in modified files. Such files are probably going to be committed soon anyway.

Store old code style in project

It's always possible to explicitly set the IntelliJ IDEA code style as the correct code style for the project. To do so, switch to the Project scheme in Settings | Editor | Code Style | Kotlin and select Kotlin obsolete IntelliJ IDEA codestyle in the Use defaults from: on the Load tab.

In order to share the changes across the project developers .idea/codeStyle folder, it has to be committed to VCS. Alternatively, `kotlin.code.style = obsolete` can be used for projects configured with Gradle or Maven.

Eclipse IDE

[Eclipse IDE](#) is a widely known IDE that offers various packages for development in different languages and for different platforms. You can use it for writing Kotlin code. On this page, you will learn how to get started with Kotlin in Eclipse IDE.

Set up the environment

First of all, you need the Eclipse IDE installed on your system. You can download its latest version from [download page](#). The Eclipse IDE for Java Developers bundle is recommended.

To add the Kotlin support to your Eclipse IDE, install the Kotlin Plugin for Eclipse. We recommend installing the Kotlin plugin from [Eclipse Marketplace](#). Open the Help | Eclipse Marketplace... menu and search for Kotlin Plugin for Eclipse:

Eclipse Marketplace

Eclipse Marketplace

Select solutions to install. Press Install Now to proceed with installation.
Press the "more info" link to learn more about a solution.

Search Recent Popular Favorites Installed 2019 in Focus

Find: Kotlin All Markets All Categories

Kotlin Plugin for Eclipse 0.8.14

The Kotlin Plugin for Eclipse helps you write, run, debug and test programs in Kotlin language. [more info](#)

by JetBrains, Apache 2.0
[kotlin jetbrains jvm java fileExtension_kt](#)

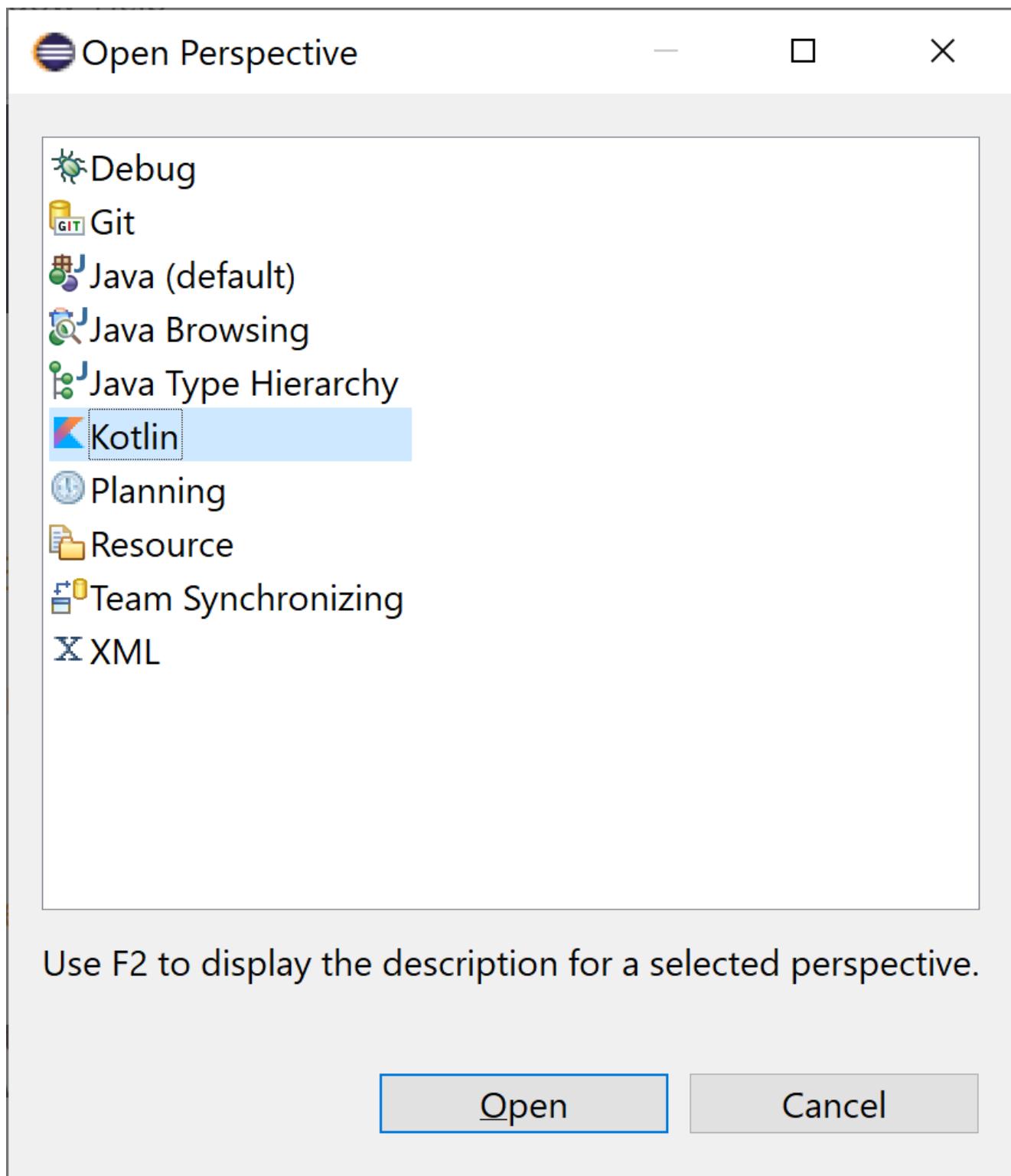
188 Installs: **83.7K** (2,907 last month)

Marketplaces

[? Back](#) [Install Now >](#) [Finish](#) [Cancel](#)

Eclipse Marketplace

Once the plugin is installed and Eclipse is restarted, make sure the plugin is installed correctly: open the Kotlin perspective in the menu Window | Open Perspective | Other...

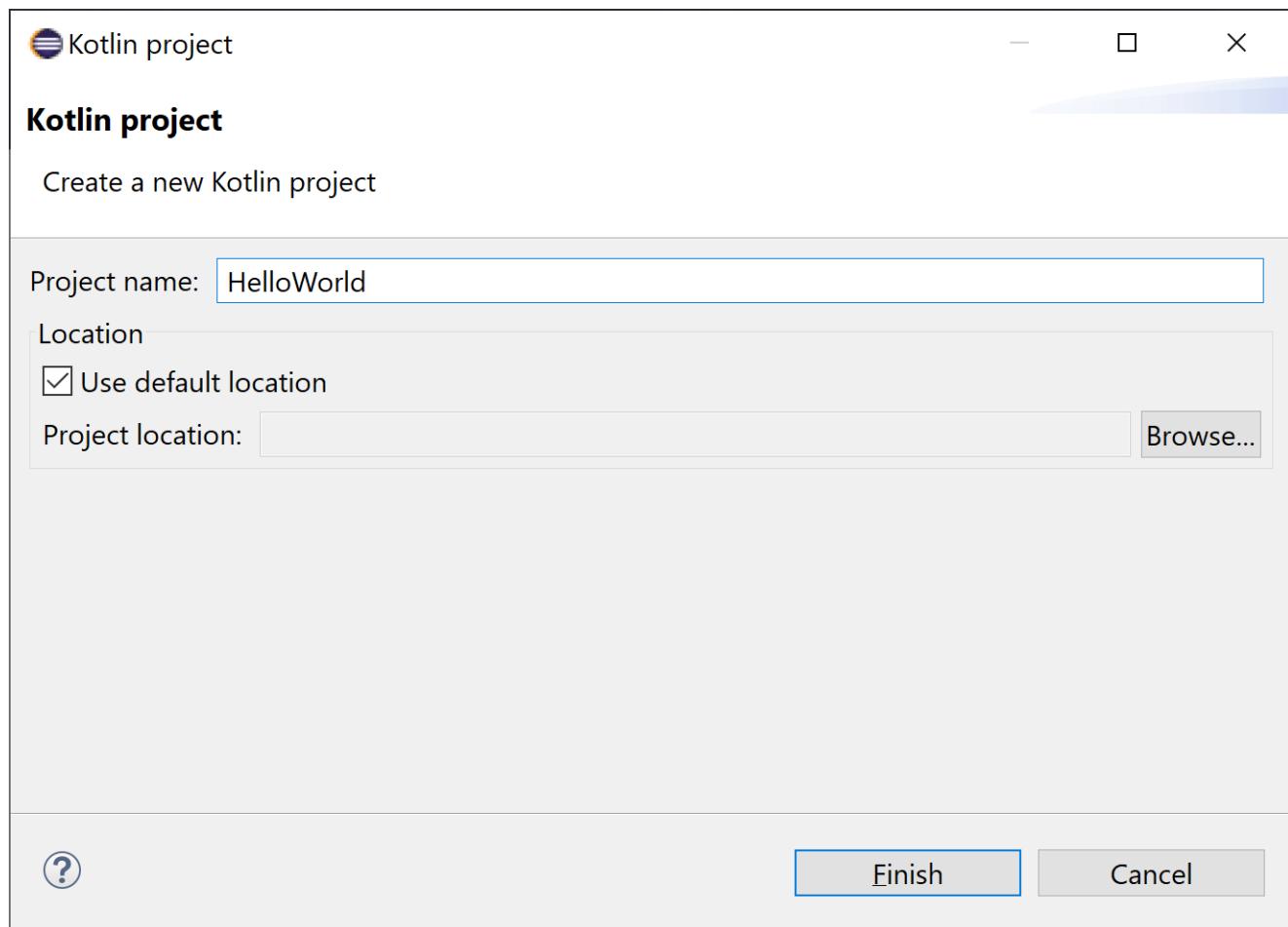


Kotlin Perspective

Create a new project

Now you are ready to create a new Kotlin project.

First, select File | New | Kotlin Project.

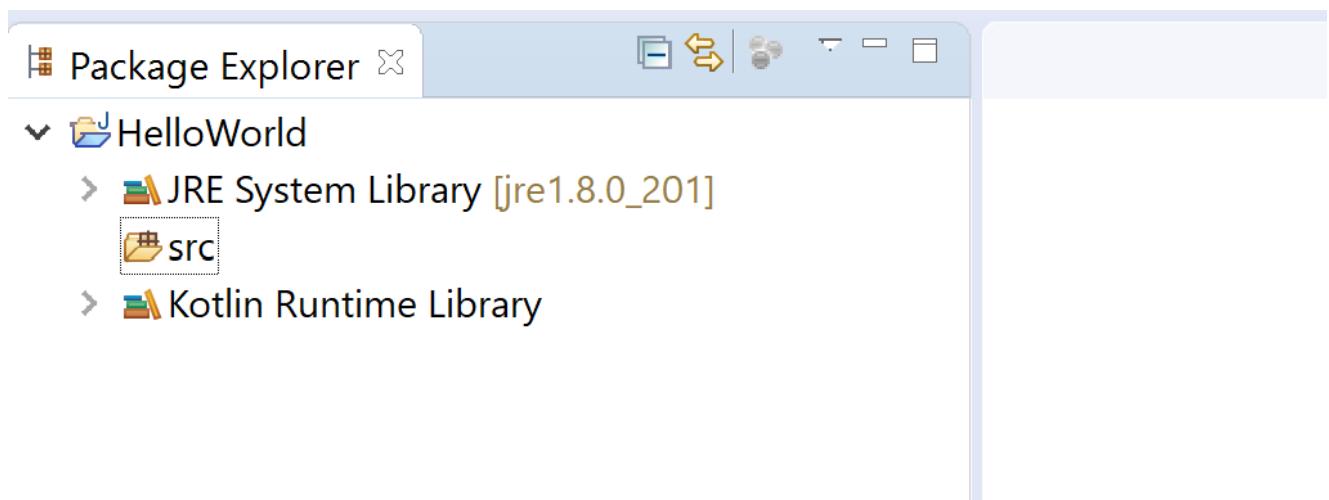


New Kotlin project

An empty Kotlin/JVM project will be created.

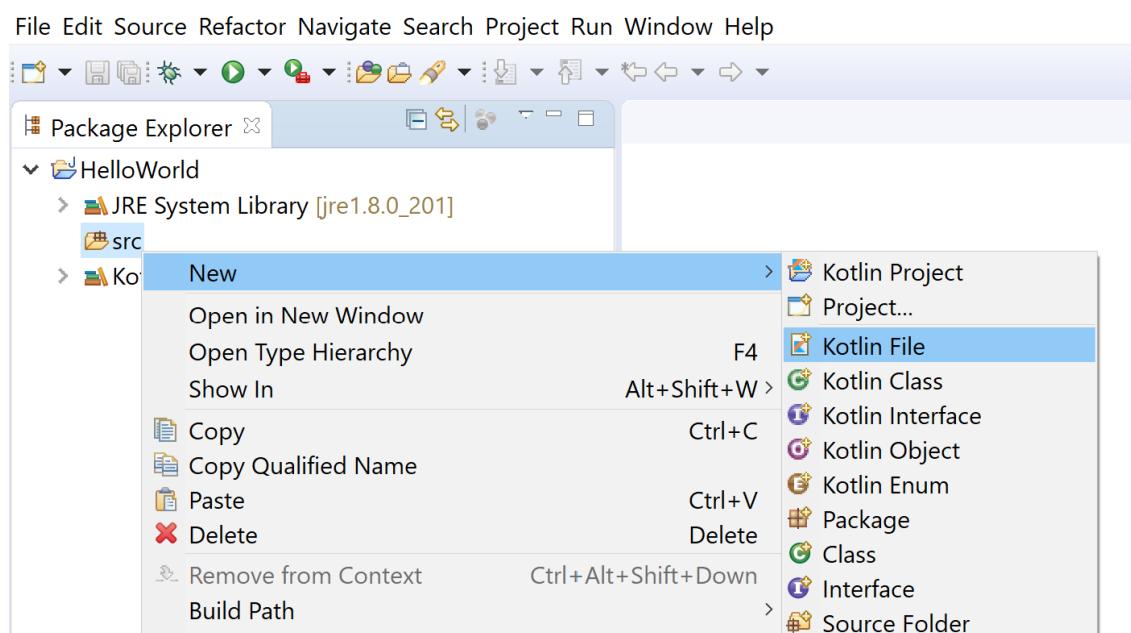
For Eclipse IDE, the project is also a Java project but configured with Kotlin nature, meaning it has the Kotlin Builder and reference to the Kotlin Runtime Library. Great thing about this solution is that you can add both Kotlin and Java code to the same project.

The project structure looks like this:



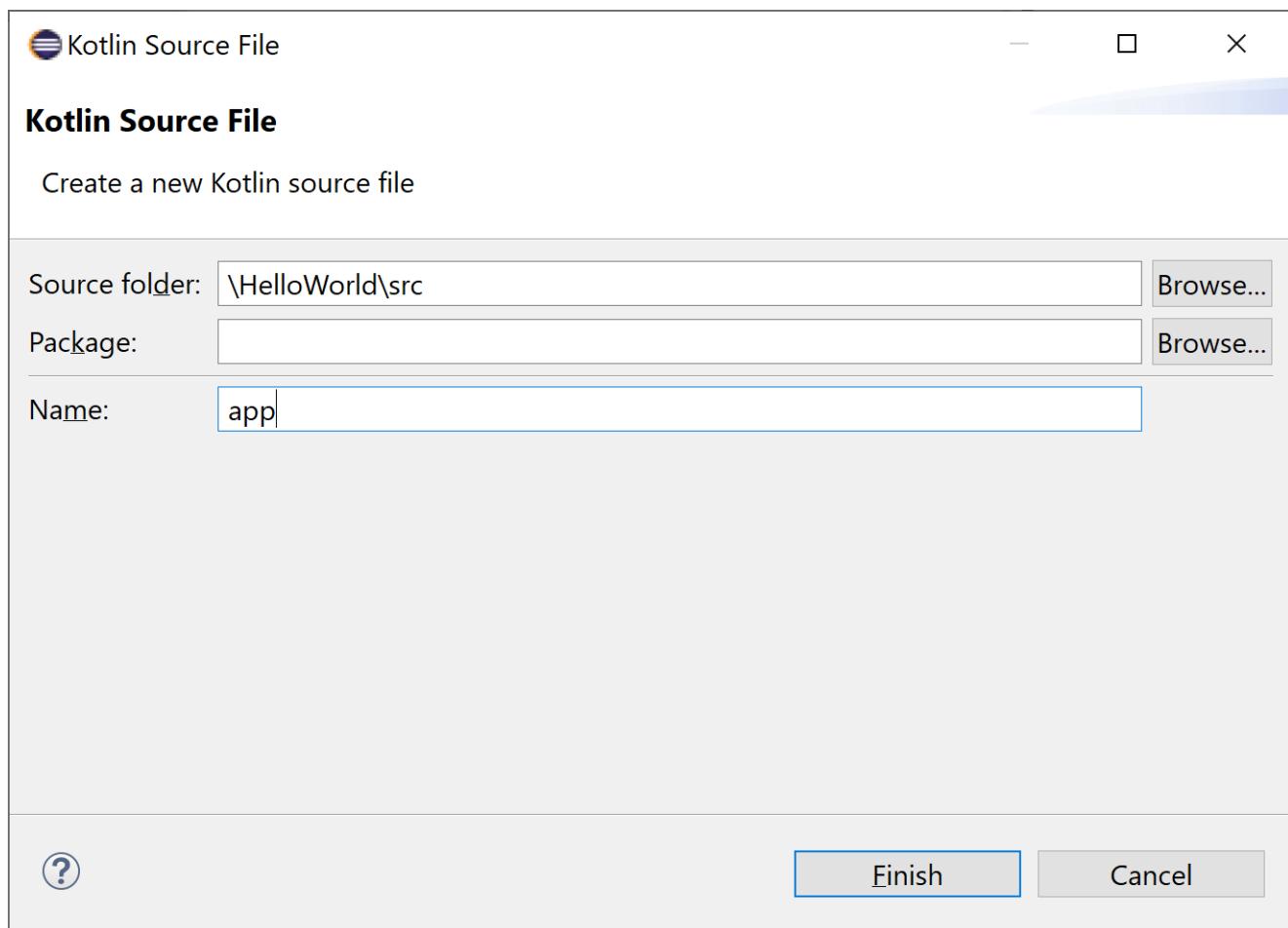
Empty Kotlin project

Now, create a new Kotlin file in the the source directory.



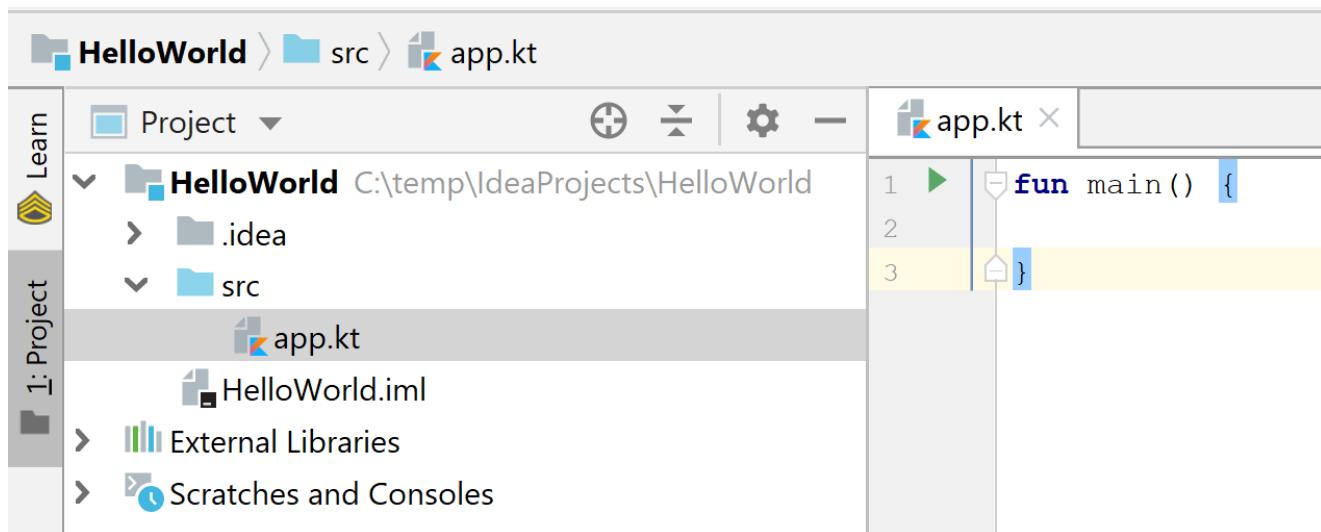
New file from context menu

You can enter the name without the .kt extension. Eclipse will add it automatically.

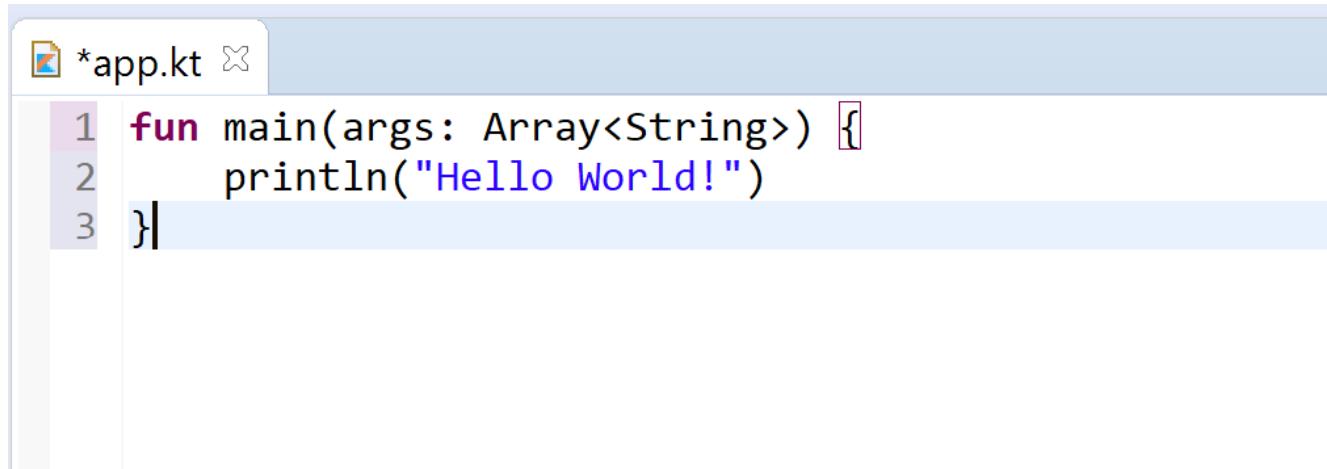


New Kotlin file wizard

Once you have a source file, add the main function - the entry point to a Kotlin application. You can simply type main and invoke code completion by hitting Ctrl + Space.



Finally, add a simple line of Kotlin code to print a message:

A screenshot of a code editor window titled "*app.kt". The code inside the file is:

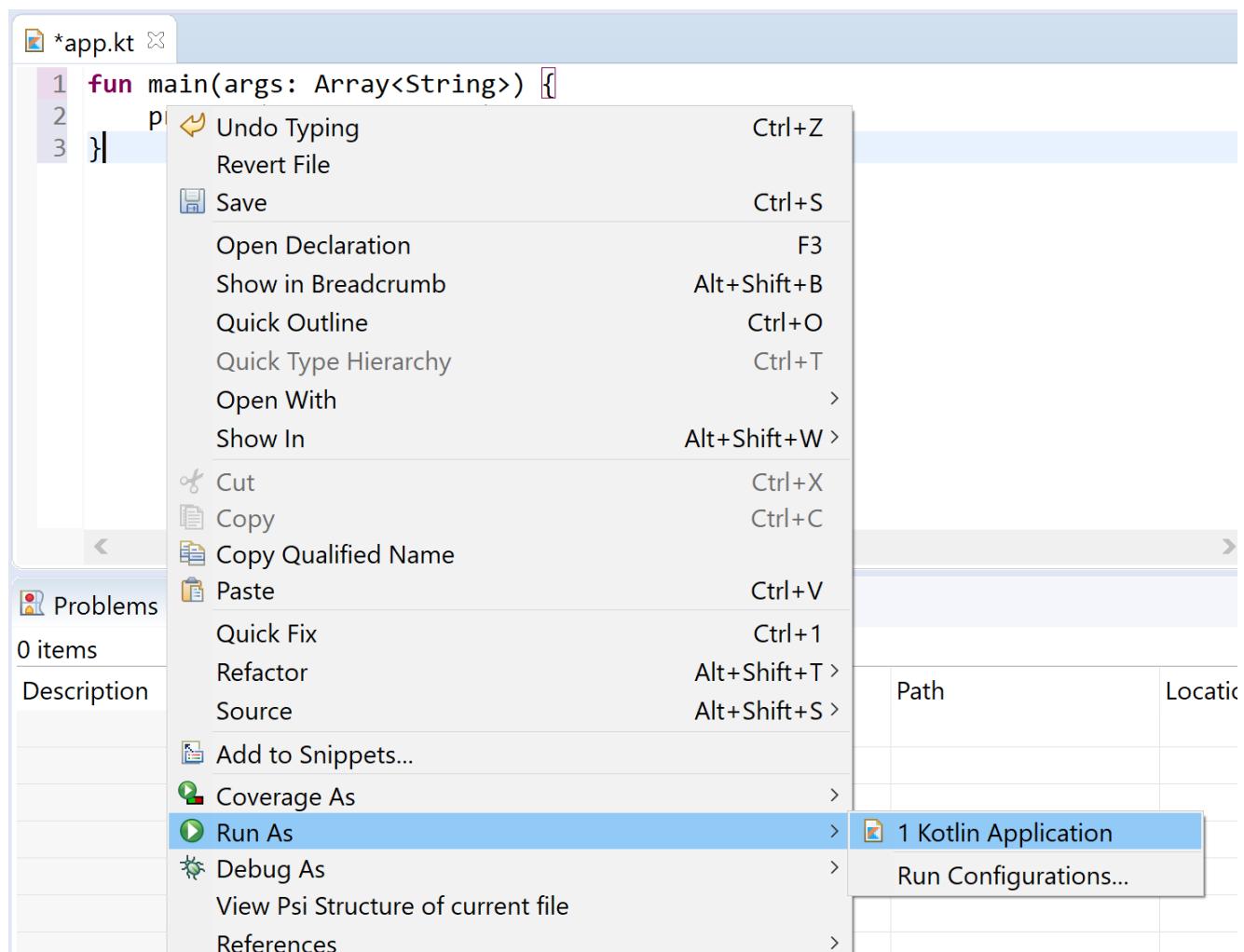
```
1 fun main(args: Array<String>) {
2     println("Hello World!")
3 }
```

The code is color-coded: "fun" and "main" are in purple, "args" is in blue, and the string "Hello World!" is in red.

Hello World example

Run the application

To run the application, right-click somewhere in the main file and select Run As | Kotlin Application.



Run Kotlin application

If everything went well, you'll see the result in the Console window.

```
hadihariri@Falcon:~/work/kotlin-native/httpclient » ./program.kexe
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
body {
    background-color: #f0f0f2;
    margin: 0;
    padding: 0;
    font-family: "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;
}

div {
    width: 600px;
    margin: 5em auto;
    padding: 50px;
    background-color: #fff;
    border-radius: 1em;
}
a:link, a:visited {
    color: #38488f;
    text-decoration: none;
}
@media (max-width: 700px) {
```

Program output view

Congratulations! You now have your Kotlin application running in Eclipse IDE.

Kotlin command-line compiler

Every release ships with a standalone version of the compiler. You can download the latest version (`kotlin-compiler-1.5.20.zip`) from [GitHub Releases](#).

Install the compiler

Manual install

Unzip the standalone compiler into a directory and optionally add the bin directory to the system path. The bin directory contains the scripts needed to compile and run Kotlin on Windows, OS X and Linux.

SDKMAN!

An easier way to install Kotlin on UNIX-based systems such as OS X, Linux, Cygwin, FreeBSD, and Solaris is [SDKMAN!](#). It also works in Bash and ZSH shells. [Learn how to install SDKMAN!](#)

To install the Kotlin compiler via SDKMAN!, run the following command in a terminal:

```
$ sdk install kotlin
```

Homebrew

Alternatively, on OS X you can install the compiler via [Homebrew](#).

```
$ brew update  
$ brew install kotlin
```

Snap package

If you use [Snap](#) on Ubuntu 16.04 or later, you can install the compiler from the command line:

```
$ sudo snap install --classic kotlin
```

Create and run an application

1. Create a simple application in Kotlin that displays "Hello, World!". In your favorite editor, create a new file called hello.kt with the following lines:

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

2. Compile the application using the Kotlin compiler:

```
$ kotlinc hello.kt -include-runtime -d hello.jar
```

The -d option indicates the output path for generated class files, which may be either a directory or a .jar file. The -include-runtime option makes the resulting .jar file self-contained and runnable by including the Kotlin runtime library in it.

To see all available options, run

```
$ kotlinc -help
```

3. Run the application.

```
$ java -jar hello.jar
```

Compile a library

If you're developing a library to be used by other Kotlin applications, you can build the .jar file without including the Kotlin runtime into it.

```
$ kotlinc hello.kt -d hello.jar
```

Since binaries compiled this way depend on the Kotlin runtime you should make sure the latter is present in the classpath whenever your compiled library is used.

You can also use the kotlin script to run binaries produced by the Kotlin compiler:

```
$ kotlin -classpath hello.jar HelloKt
```

HelloKt is the main class name that the Kotlin compiler generates for the file named hello.kt.

Run the REPL

You can run the compiler without parameters to have an interactive shell. In this shell, you can type any valid Kotlin code and see the results.

```
[Ocean] ~/tutorials/kotlin/command_line/kotlinc$ bin/kotlinc-jvm
Kotlin interactive shell
Type :help for help, :quit for quit
>>> 2+2
4
>>> println("Welcome to the Kotlin Shell")
Welcome to the Kotlin Shell
>>>
```

Shell

Run scripts

Kotlin can also be used as a scripting language. A script is a Kotlin source file (.kts) with top-level executable code.

```
import java.io.File

// Get the passed in path, i.e. "-d some/path" or use the current path.
val path = if (args.contains("-d")) args[1 + args.indexOf("-d")]
            else ""

val folders = File(path).listFiles { file -> file.isDirectory() }
folders?.forEach { folder -> println(folder) }
```

To run a script, we just pass the `-script` option to the compiler with the corresponding script file.

```
$ kotlinc -script list_folders.kts -- -d <path_to_folder_to_inspect>
```

Since 1.3.0, Kotlin has an experimental support for scripts customization, such as adding external properties, providing static or dynamic dependencies, and so on. Customizations are defined by so-called Script definitions- annotated kotlin classes with appropriate support code. The script filename extension is used to select appropriate definition.

Properly prepared script definitions are detected and applied automatically when the appropriate jars are included in the compilation classpath. Alternatively, you can specify definitions manually using `-script-templates` option to the compiler:

```
$ kotlinc -script-templates org.example.CustomScriptDefinition -script custom.script1.kts
```

For additional details, please consult the [KEEP-75](#).

Kotlin compiler options

Each release of Kotlin includes compilers for the supported targets: JVM, JavaScript, and native binaries for [supported platforms](#).

These compilers are used by the IDE when you click the Compile or Run button for your Kotlin project.

You can also run Kotlin compilers manually from the command line as described in the [Working with command-line compiler](#) tutorial.

Compiler options

Kotlin compilers have a number of options for tailoring the compiling process. Compiler options for different targets are listed on this page together with a description of each one.

There are several ways to set the compiler options and their values (compiler arguments):

- In IntelliJ IDEA, write in the compiler arguments in the Additional command-line parameters text box in Settings | Build, Execution, Deployment | Compilers | Kotlin Compiler
- If you're using Gradle, specify the compiler arguments in the `kotlinOptions` property of the Kotlin compilation task. For details, see [Gradle](#).
- If you're using Maven, specify the compiler arguments in the `<configuration>` element of the Maven plugin node. For details, see [Maven](#).
- If you run a command-line compiler, add the compiler arguments directly to the utility call or write them into an [argfile](#).

For example:

```
$ kotlinc hello.kt -include-runtime -d hello.jar
```

On Windows, when you pass compiler arguments that contain delimiter characters (whitespace, =, ;, ,), surround these arguments with double quotes ("").

```
$ kotlinc.bat hello.kt -include-runtime -d "My Folder\hello.jar"
```

Common options

The following options are common for all Kotlin compilers.

-version

Display the compiler version.

-nowarn

Suppress the compiler from displaying warnings during compilation.

-Werror

Turn any warnings into a compilation error.

-verbose

Enable verbose logging output which includes details of the compilation process.

-script

Evaluate a Kotlin script file. When called with this option, the compiler executes the first Kotlin script (*.kts) file among the given arguments.

-help (-h)

Display usage information and exit. Only standard options are shown. To show advanced options, use -X.

-X

Display information about the advanced options and exit. These options are currently unstable: their names and behavior may be changed without notice.

-kotlin-home path

Specify a custom path to the Kotlin compiler used for the discovery of runtime libraries.

-P plugin:pluginId:optionName=value

Pass an option to a Kotlin compiler plugin. Available plugins and their options are listed in the Tools > Compiler plugins section of the documentation.

-language-version version

Provide source compatibility with the specified version of Kotlin.

-api-version version

Allow using declarations only from the specified version of Kotlin bundled libraries.

-progressive

Enable the progressive mode for the compiler.

In the progressive mode, deprecations and bug fixes for unstable code take effect immediately, instead of going through a graceful migration cycle. Code written in the progressive mode is backwards compatible; however, code written in a non-progressive mode may cause compilation errors in the progressive mode.

@argfile

Read the compiler options from the given file. Such a file can contain compiler options with values and paths to the source files. Options and paths should be separated by whitespaces. For example:

-include-runtime -d hello.jar hello.kt

To pass values that contain whitespaces, surround them with single ('') or double ("") quotes. If a value contains quotation marks in it, escape them with a backslash (\).

-include-runtime -d 'My folder'

You can also pass multiple argument files, for example, to separate compiler options from source files.

```
$ kotlinc @compiler.options @classes
```

If the files reside in locations different from the current directory, use relative paths.

```
$ kotlinc @options/compiler.options hello.kt
```

Kotlin/JVM compiler options

The Kotlin compiler for JVM compiles Kotlin source files into Java class files. The command-line tools for Kotlin to JVM compilation are `kotlin` and `kotlinc-jvm`. You can also use them for executing Kotlin script files.

In addition to the [common options](#), Kotlin/JVM compiler has the options listed below.

-classpath path (-cp path)

Search for class files in the specified paths. Separate elements of the classpath with system path separators (; on Windows, : on macOS/Linux). The classpath can contain file and directory paths, ZIP, or JAR files.

-d path

Place the generated class files into the specified location. The location can be a directory, a ZIP, or a JAR file.

-include-runtime

Include the Kotlin runtime into the resulting JAR file. Makes the resulting archive runnable on any Java-enabled environment.

-jdk-home path

Use a custom JDK home directory to include into the classpath if it differs from the default JAVA_HOME.

-jvm-target version

Specify the target version of the generated JVM bytecode. Possible values are 1.6 (DEPRECATED), 1.8, 9, 10, 11, 12, 13, 14, 15 and 16. The default value is 1.8.

-java-parameters

Generate metadata for Java 1.8 reflection on method parameters.

-module-name name (JVM)

Set a custom name for the generated `.kotlin_module` file.

-no-jdk

Don't automatically include the Java runtime into the classpath.

-no-reflect

Don't automatically include the Kotlin reflection (`kotlin-reflect.jar`) into the classpath.

-no-stdlib (JVM)

Don't automatically include the Kotlin/JVM stdlib (kotlin-stdlib.jar) and Kotlin reflection (kotlin-reflect.jar) into the classpath.

-script-templates classnames[,]

Script definition template classes. Use fully qualified class names and separate them with commas (,).

Kotlin/JS compiler options

The Kotlin compiler for JS compiles Kotlin source files into JavaScript code. The command-line tool for Kotlin to JS compilation is kotlinc-js.

In addition to the [common options](#), Kotlin/JS compiler has the options listed below.

-libraries path

Paths to Kotlin libraries with .meta.js and .kjsm files, separated by the system path separator.

-main {call|noCall}

Define whether the main function should be called upon execution.

-meta-info

Generate .meta.js and .kjsm files with metadata. Use this option when creating a JS library.

-module-kind {umd|commonjs|amd|plain}

The kind of JS module generated by the compiler:

- umd- a [Universal Module Definition](#) module
- commonjs- a [CommonJS](#) module
- amd- an [Asynchronous Module Definition](#) module
- plain- a plain JS module

To learn more about the different kinds of JS module and the distinctions between them, see [this article](#).

-no-stdlib (JS)

Don't automatically include the default Kotlin/JS stdlib into the compilation dependencies.

-output filepath

Set the destination file for the compilation result. The value must be a path to a .js file including its name.

-output-postfix filepath

Add the content of the specified file to the end of the output file.

-output-prefix filepath

Add the content of the specified file to the beginning of the output file.

-source-map

Generate the source map.

-source-map-base-dirs path

Use the specified paths as base directories. Base directories are used for calculating relative paths in the source map.

-source-map-embed-sources {always|never|inlining}

Embed source files into the source map.

-source-map-prefix

Add the specified prefix to paths in the source map.

Kotlin/Native compiler options

Kotlin/Native compiler compiles Kotlin source files into native binaries for the [supported platforms](#). The command-line tool for Kotlin/Native compilation is kotlinc-native.

In addition to the [common options](#), Kotlin/Native compiler has the options listed below.

-enable-assertions (-ea)

Enable runtime assertions in the generated code.

-g

Enable emitting debug information.

-generate-test-runner (-tr)

Produce an application for running unit tests from the project.

-generate-worker-test-runner (-trw)

Produce an application for running unit tests in a worker thread.

-generate-no-exit-test-runner (-trn)

Produce an application for running unit tests without an explicit process exit.

-include-binary path (-ib path)

Pack external binary within the generated klib file.

-library path (-l path)

Link with the library. To learn about using libraries in Kotlin/native projects, see [Kotlin/Native libraries](#).

-library-version version (-lv version)

Set the library version.

-list-targets

List the available hardware targets.

-manifest path

Provide a manifest addend file.

-module-name name (Native)

Specify a name for the compilation module. This option can also be used to specify a name prefix for the declarations exported to Objective-C: [How do I specify a custom Objective-C prefix/name for my Kotlin framework?](#)

-native-library path (-nl path)

Include the native bitcode library.

-no-default-libs

Disable linking user code with the default platform libraries distributed with the compiler.

-nomain

Assume the main entry point to be provided by external libraries.

-nopack

Don't pack the library into a klib file.

-linker-option

Pass an argument to the linker during binary building. This can be used for linking against some native library.

-linker-options args

Pass multiple arguments to the linker during binary building. Separate arguments with whitespaces.

-nostdlib

Don't link with stdlib.

-opt

Enable compilation optimizations.

-output name (-o name)

Set the name for the output file.

-entry name (-e name)

Specify the qualified entry point name.

-produce output (-p output)

Specify output file kind:

- program
- static
- dynamic
- framework
- library
- bitcode

-repo path (-r path)

Library search path. For more information, see [Library search sequence](#).

-target target

Set hardware target. To see the list of available targets, use the [-list-targets](#) option.

All-open compiler plugin

Kotlin has classes and their members final by default, which makes it inconvenient to use frameworks and libraries such as Spring AOP that require classes to be open. The all-open compiler plugin adapts Kotlin to the requirements of those frameworks and makes classes annotated with a specific annotation and their members open without the explicit open keyword.

For instance, when you use Spring, you don't need all the classes to be open, but only classes annotated with specific annotations like @Configuration or @Service. All-open allows to specify such annotations.

We provide all-open plugin support both for Gradle and Maven with the complete IDE integration.

For Spring, you can use the kotlin-spring compiler plugin ([see below](#)).

Gradle

Add the plugin artifact to the build script dependencies and apply the plugin:

```
buildscript {  
    dependencies {  
        classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"  
    }  
}  
  
apply plugin: "kotlin-allopen"
```

As an alternative, you can enable it using the plugins block:

```
plugins {  
    id "org.jetbrains.kotlin.plugin.allopen" version "1.5.20"  
}
```

Then specify the list of annotations that will make classes open:

```
allOpen {  
    annotation("com.my.Annotation")
```

```
// annotations("com.another.Annotation", "com.third.Annotation")
}
```

If the class (or any of its superclasses) is annotated with com.my.Annotation, the class itself and all its members will become open.

It also works with meta-annotations:

```
@com.my.Annotation
annotation class MyFrameworkAnnotation

@MyFrameworkAnnotation
class MyClass // will be all-open
```

MyFrameworkAnnotation is annotated with the all-open meta-annotation com.my.Annotation, so it becomes an all-open annotation as well.

Maven

Here's how to use all-open with Maven:

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- Or "spring" for the Spring support -->
            <plugin>all-open</plugin>
        </compilerPlugins>

        <pluginOptions>
            <!-- Each annotation is placed on its own line -->
            <option>all-open:annotation=com.my.Annotation</option>
            <option>all-open:annotation=com.their.AnotherAnnotation</option>
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-allopen</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

Please refer to the [Gradle](#) section for the detailed information about how all-open annotations work.

Spring support

If you use Spring, you can enable the kotlin-spring compiler plugin instead of specifying Spring annotations manually. The kotlin-spring is a wrapper on top of all-open, and it behaves exactly the same way.

As with all-open, add the plugin to the build script dependencies:

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
    }
}

apply plugin: "kotlin-spring" // instead of "kotlin-allopen"
```

Or using the Gradle plugins DSL:

```
plugins {
    id "org.jetbrains.kotlin.plugin.spring" version "1.5.20"
}
```

In Maven, the spring plugin is provided by the kotlin-maven-allopen plugin dependency, so to enable it:

```
<compilerPlugins>
    <plugin>spring</plugin>
</compilerPlugins>

<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-maven-allopen</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

The plugin specifies the following annotations:

- [@Component](#)
- [@Async](#)
- [@Transactional](#)
- [@Cacheable](#)
- [@SpringBootTest](#)

Thanks to meta-annotations support, classes annotated with [@Configuration](#), [@Controller](#), [@RestController](#), [@Service](#) or [@Repository](#) are automatically opened since these annotations are meta-annotated with [@Component](#).

Of course, you can use both kotlin-allopen and kotlin-spring in the same project.

Note that if you use the project template generated by the [start.spring.io](#) service, the kotlin-spring plugin will be enabled by default.

Command-line compiler

All-open compiler plugin JAR is available in the binary distribution of the Kotlin compiler. You can attach the plugin by providing the path to its JAR file using the Xplugin kotlinc option:

```
-Xplugin=$KOTLIN_HOME/lib/allopen-compiler-plugin.jar
```

You can specify all-open annotations directly, using the annotation plugin option, or enable the "preset". The only preset available now for all-open is spring.

```
# The plugin option format is: "-P plugin:<plugin id>:<key>=<value>".
# Options can be repeated.

-P plugin:org.jetbrains.kotlin.allopen:annotation=com.my.Annotation
-P plugin:org.jetbrains.kotlin.allopen:preset=spring
```

No-arg compiler plugin

The no-arg compiler plugin generates an additional zero-argument constructor for classes with a specific annotation.

The generated constructor is synthetic so it can't be directly called from Java or Kotlin, but it can be called using reflection.

This allows the Java Persistence API (JPA) to instantiate a class although it doesn't have the zero-parameter constructor from Kotlin or Java point of view (see the description of kotlin-jpa plugin [below](#)).

Gradle

Add the plugin and specify the list of annotations that must lead to generating a no-arg constructor for the annotated classes.

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"
    }
}

apply plugin: "kotlin-noarg"
```

Or using the Gradle plugins DSL:

```
plugins {
    id "org.jetbrains.kotlin.plugin.noarg" version "1.5.20"
}
```

Then specify the list of no-arg annotations:

```
noArg {
    annotation("com.my.Annotation")
}
```

Enable `invokeInitializers` option if you want the plugin to run the initialization logic from the synthetic constructor. By default, it is disabled.

```
noArg {
    invokeInitializers = true
}
```

Maven

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- Or "jpa" for JPA support -->
            <plugin>no-arg</plugin>
        </compilerPlugins>

        <pluginOptions>
            <option>no-arg:annotation=com.my.Annotation</option>
            <!-- Call instance initializers in the synthetic constructor -->
            <!-- <option>no-arg:invokeInitializers=true</option> -->
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-noarg</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

JPA support

As with the kotlin-spring plugin wrapped on top of all-open, kotlin-jpa is wrapped on top of no-arg. The plugin specifies `@Entity`, `@Embeddable`, and `@MappedSuperclass` no-arg annotations automatically.

That's how you add the plugin in Gradle:

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"
    }
}

apply plugin: "kotlin-jpa"
```

Or using the Gradle plugins DSL:

```
plugins {
    id "org.jetbrains.kotlin.plugin.jpa" version "1.5.20"
}
```

In Maven, enable the jpa plugin:

```
<compilerPlugins>
    <plugin>jpa</plugin>
</compilerPlugins>
```

Command-line compiler

Add the plugin JAR file to the compiler plugin classpath and specify annotations or presets:

```
-Xplugin=$KOTLIN_HOME/lib/noarg-compiler-plugin.jar
-P plugin:org.jetbrains.kotlin.noarg:annotation=com.my.Annotation
-P plugin:org.jetbrains.kotlin.noarg:preset=jpa
```

SAM-with-receiver compiler plugin

The sam-with-receiver compiler plugin makes the first parameter of the annotated Java "single abstract method" (SAM) interface method a receiver in Kotlin. This conversion only works when the SAM interface is passed as a Kotlin lambda, both for SAM adapters and SAM constructors (see the [SAM conversions documentation](#) for more details).

Here is an example:

```
public @interface SamWithReceiver {}  
  
@SamWithReceiver  
public interface TaskRunner {  
    void run(Task task);  
}
```

```
fun test(context: TaskContext) {  
    val runner = TaskRunner {  
        // Here 'this' is an instance of 'Task'  
  
        println("$name is started")  
        context.executeTask(this)  
        println("$name is finished")  
    }  
}
```

Gradle

The usage is the same to [all-open](#) and [no-arg](#), except the fact that sam-with-receiver does not have any built-in presets, and you need to specify your own list of special-treated annotations.

```
buildscript {  
    dependencies {  
        classpath "org.jetbrains.kotlin:kotlin-sam-with-receiver:$kotlin_version"  
    }  
}  
  
apply plugin: "kotlin-sam-with-receiver"
```

Then specify the list of SAM-with-receiver annotations:

```
samWithReceiver {  
    annotation("com.my.SamWithReceiver")  
}
```

Maven

```
<plugin>  
    <artifactId>kotlin-maven-plugin</artifactId>  
    <groupId>org.jetbrains.kotlin</groupId>  
    <version>${kotlin.version}</version>  
  
    <configuration>  
        <compilerPlugins>  
            <plugin>sam-with-receiver</plugin>  
        </compilerPlugins>
```

```

<pluginOptions>
    <option>
        sam-with-receiver:annotation=com.my.SamWithReceiver
    </option>
</pluginOptions>
</configuration>

<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-maven-sam-with-receiver</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
</plugin>

```

Command-line compiler

Add the plugin JAR file to the compiler plugin classpath and specify the list of sam-with-receiver annotations:

```

-Xplugin=$KOTLIN_HOME/lib/sam-with-receiver-compiler-plugin.jar
-P plugin:org.jetbrains.kotlin.samWithReceiver:annotation=com.my.SamWithReceiver

```

Using kapt

Annotation processors (see [JSR 269](#)) are supported in Kotlin with the kapt compiler plugin.

In a nutshell, you can use libraries such as [Dagger](#) or [Data Binding](#) in your Kotlin projects.

Please read below about how to apply the kapt plugin to your Gradle/Maven build.

Using in Gradle

Apply the kotlin-kapt Gradle plugin:

Groovy

```

plugins {
    id "org.jetbrains.kotlin.kapt" version "1.5.20"
}

```

Kotlin

```
plugins {
    kotlin("kapt") version "1.5.20"
}
```

Alternatively, you can use the apply plugin syntax:

```
apply plugin: 'kotlin-kapt'
```

Then add the respective dependencies using the kapt configuration in your dependencies block:

Groovy

```
dependencies {
    kapt 'groupId:artifactId:version'
}
```

Kotlin

```
dependencies {
    kapt("groupId:artifactId:version")
}
```

If you previously used the [Android support](#) for annotation processors, replace usages of the annotationProcessor configuration with kapt. If your project contains Java classes, kapt will also take care of them.

If you use annotation processors for your androidTest or test sources, the respective kapt configurations are named kaptAndroidTest and kaptTest. Note that kaptAndroidTest and kaptTest extends kapt, so you can just provide the kapt dependency and it will be available both for production sources and tests.

Annotation processor arguments

Use arguments {} block to pass arguments to annotation processors:

```
kapt {
    arguments {
        arg("key", "value")
    }
}
```

Gradle build cache support

The kapt annotation processing tasks are [cached in Gradle](#) by default. However, annotation processors run arbitrary code that may not necessarily transform the task inputs into the outputs, might access and modify the files that are not tracked by Gradle etc. If the annotation processors used in the build cannot be properly cached, it is possible to disable caching for kapt entirely by adding the following lines to the build script, in order to avoid false-positive cache hits for the kapt tasks:

```
kapt {  
    useBuildCache = false  
}
```

Improving the speed of builds that use kapt

Running kapt tasks in parallel

To improve the speed of builds that use kapt, you can enable the [Gradle worker API](#) for kapt tasks. Using the worker API lets Gradle run independent annotation processing tasks from a single project in parallel, which in some cases significantly decreases the execution time. However, running kapt with Gradle worker API enabled can result in increased memory consumption due to parallel execution.

To use the Gradle worker API for parallel execution of kapt tasks, add this line to your gradle.properties file:

```
kapt.use.worker.api=true
```

Caching for annotation processors' classloaders

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](#).

Caching for annotation processors' classloaders helps kapt perform faster if you run many Gradle tasks consecutively.

To enable this feature, use the following properties in your gradle.properties file:

```
# 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
```

If you run into any problems with caching for annotation processors, disable caching for them:

```
# specify annotation processors' full names to disable caching for them  
kapt.classloaders.cache.disableForProcessors=[annotation processors full names]
```

Compile avoidance for kapt

To improve the times of incremental builds with kapt, it can use the Gradle [compile avoidance](#). With compile avoidance enabled, Gradle can skip annotation processing when rebuilding a project. Particularly, annotation processing is skipped when:

- The project's source files are unchanged.
- The changes in dependencies are [ABI](#) compatible. For example, the only changes are in method bodies.

However, compile avoidance can't be used for annotation processors discovered in the compile classpath since any changes in them require running the annotation processing tasks.

To run kapt with compile avoidance:

- Add the annotation processor dependencies to the kapt* configurations manually as described [above](#).
- Turn off the discovery of annotation processors in the compile classpath by adding this line to your gradle.properties file:

```
kapt.include.compile.classpath=false
```

Incremental annotation processing

kapt supports incremental annotation processing that is enabled by default. Currently, annotation processing can be incremental only if all annotation processors being used are incremental.

To disable incremental annotation processing, add this line to your gradle.properties file:

```
kapt.incremental.apt=false
```

Note that incremental annotation processing requires [incremental compilation](#) to be enabled as well.

Java compiler options

kapt uses Java compiler to run annotation processors.

Here is how you can pass arbitrary options to javac:

```
kapt {  
    javacOptions {  
        // Increase the max count of errors from annotation processors.  
        // Default is 100.  
        option("-Xmaxerrs", 500)  
    }  
}
```

Non-existent type correction

Some annotation processors (such as AutoFactory) rely on precise types in declaration signatures. By default, kapt replaces every unknown type (including types for the generated classes) to NonExistentClass, but you can change this behavior. Add the option to the build.gradle file to enable error type inferring in stubs:

```
kapt {  
    correctErrorTypes = true  
}
```

Using in Maven

Add an execution of the kapt goal from kotlin-maven-plugin before compile:

```
<execution>  
    <id>kapt</id>  
    <goals>  
        <goal>kapt</goal>  
    </goals>  
    <configuration>  
        <sourceDirs>  
            <sourceDir>src/main/kotlin</sourceDir>  
            <sourceDir>src/main/java</sourceDir>  
        </sourceDirs>  
        <annotationProcessorPaths>  
            <!-- Specify your annotation processors here. -->  
            <annotationProcessorPath>  
                <groupId>com.google.dagger</groupId>  
                <artifactId>dagger-compiler</artifactId>  
                <version>2.9</version>  
            </annotationProcessorPath>  
        </annotationProcessorPaths>  
    </configuration>  
</execution>
```

You can find a complete sample project showing the use of Kotlin, Maven and Dagger in the [Kotlin examples repository](#).

Please note that kapt is still not supported for IntelliJ IDEA's own build system. Launch the build from the “Maven Projects” toolbar whenever you want to re-run the annotation processing.

Using in CLI

kapt compiler plugin is available in the binary distribution of the Kotlin compiler.

You can attach the plugin by providing the path to its JAR file using the Xplugin kotlinc option:

```
-Xplugin=$KOTLIN_HOME/lib/kotlin-annotation-processing.jar
```

Here is a list of the available options:

- sources (required): An output path for the generated files.
- classes (required): An output path for the generated class files and resources.
- stubs (required): An output path for the stub files. In other words, some temporary directory.
- incrementalData: An output path for the binary stubs.
- apclasspath (repeatable): A path to the annotation processor JAR. Pass as many apclasspath options as many JARs you have.
- apoptions: A base64-encoded list of the annotation processor options. See [AP/javac options encoding](#) for more information.
- javacArguments: A base64-encoded list of the options passed to javac. See [AP/javac options encoding](#) for more information.
- processors: A comma-specified list of annotation processor qualified class names. If specified, kapt does not try to find annotation processors in apclasspath.
- verbose: Enable verbose output.
- aptMode (required)
 - stubs – only generate stubs needed for annotation processing;
 - apt – only run annotation processing;
 - stubsAndApt – generate stubs and run annotation processing.
- correctErrorTypes: See [below](#). Disabled by default.

The plugin option format is: -P plugin:<plugin id>:<key>=<value>. Options can be repeated.

An example:

```
-P plugin:org.jetbrains.kotlin.kapt3:sources=build/kapt/sources
-P plugin:org.jetbrains.kotlin.kapt3:classes=build/kapt/classes
-P plugin:org.jetbrains.kotlin.kapt3:stubs=build/kapt/stubs

-P plugin:org.jetbrains.kotlin.kapt3:apclasspath=lib/ap.jar
-P plugin:org.jetbrains.kotlin.kapt3:apclasspath=lib/anotherAp.jar

-P plugin:org.jetbrains.kotlin.kapt3:correctErrorTypes=true
```

Generating Kotlin sources

kapt can generate Kotlin sources. Just write the generated Kotlin source files to the directory specified by `processingEnv.options["kapt.kotlin.generated"]`, and these files will be compiled together with the main sources.

You can find the complete sample in the [kotlin-examples](#) GitHub repository.

Note that kapt does not support multiple rounds for the generated Kotlin files.

AP/Javac options encoding

`apoptions` and `javacArguments` CLI options accept an encoded map of options.

Here is how you can encode options by yourself:

```
fun encodeList(options: Map<String, String>): String {
    val os = ByteArrayOutputStream()
    val oos = ObjectOutputStream(os)

    oos.writeInt(options.size)
    for ((key, value) in options.entries) {
        oos.writeUTF(key)
        oos.writeUTF(value)
    }

    oos.flush()
    return Base64.getEncoder().encodeToString(os.toByteArray())
}
```

Keeping Java compiler's annotation processors

By default, kapt runs all annotation processors and disables annotation processing by javac. However, you may need some of javac's annotation processors working (for example, [Lombok](#)).

In the Gradle build file, use the option `keepJavaAnnotationProcessors`:

```
kapt {
    keepJavaAnnotationProcessors = true
}
```

If you use Maven, you need to specify concrete plugin settings. See this [example of settings for the Lombok compiler plugin](#).

Lombok compiler plugin

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](#).

The Kotlin Lombok compiler plugin allows the generation and use of Java's Lombok declarations by Kotlin code in the same mixed Java/Kotlin module. If you call such declarations from another module, then you don't need to use this plugin for the compilation of that module.

The Lombok compiler plugin cannot replace [Lombok](#), but it helps Lombok work in mixed Java/Kotlin modules. Thus, you still need to configure Lombok as usual when using this plugin. Learn more about [how to make the plugin seeing Lombok's config](#).

Supported annotations

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](#).

Kotlin compiler ignores Lombok annotations if you use them in Kotlin code.

Gradle

Apply the `kotlin-plugin-lombok` Gradle plugin in the `build.gradle(.kts)` file:

Groovy

```
plugins {  
    id 'org.jetbrains.kotlin.plugin.lombok' version '1.5.20'  
    id 'io.freefair.lombok' version '5.3.0'  
}
```

Kotlin

```
plugins {
    id ("org.jetbrains.kotlin.plugin.lombok") version "1.5.20"
    id ("io.freefair.lombok") version "5.3.0"
}
```

See this [test project with examples of the Lombok compiler plugin in use](#).

Using the Lombok configuration file

If you use a [Lombok configuration file](#) lombok.config, provide a path to it to the plugin. The path should be relative to the module's directory. Add the following code to your build.gradle(.kts) file:

Groovy

```
kotlinLombok {
    lombokConfigurationFile file("lombok.config")
}
```

Kotlin

```
kotlinLombok {
    lombokConfigurationFile(file("lombok.config"))
}
```

See this [test project with examples of the Lombok compiler plugin and lombok.config in use](#).

Maven

To use the Lombok compiler plugin, add the plugin lombok to the compilerPlugins section and the dependency kotlin-maven-lombok to the dependencies section. If you use a [Lombok configuration file](#) lombok.config, provide a path to it to the plugin in the pluginOptions. Add the following lines to the pom.xml file:

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <configuration>
        <compilerPlugins>
            <plugin>lombok</plugin>
        </compilerPlugins>
        <pluginOptions>
```

```

        <option>lombok:config=${project.basedir}/lombok.config</option>
    </pluginOptions>
</configuration>
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-maven-lombok</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.20</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
</plugin>

```

See this [test project example of the Lombok compiler plugin and lombok.config in use](#).

Using with kapt

By default, the [kapt](#) compiler plugin runs all annotation processors and disables annotation processing by javac. To run [Lombok](#) along with kapt, set up kapt to keep javac's annotation processors working.

If you use Gradle, add the option to the build.gradle(.kts) file:

```

kapt {
    keepJavaAnnotationProcessors = true
}

```

In Maven, use the following settings to launch Lombok with Java's compiler:

```

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.5.1</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <annotationProcessorPaths>
            <annotationProcessorPath>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>${lombok.version}</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
</plugin>

```

The Lombok compiler plugin works correctly with [kapt](#) if annotation processors don't depend on the code generated

by Lombok.

Look through the test project examples of kapt and the Lombok compiler plugin in use:

- Using [Gradle](#).
- Using [Maven](#)

Kotlin and continuous integration with TeamCity

On this page, you'll learn how to set up [TeamCity](#) to build your Kotlin project. For more information and basics of TeamCity please check the [Documentation page](#) which contains information about installation, basic configuration, etc.

Kotlin works with different build tools, so if you're using a standard tool such as Ant, Maven or Gradle, the process for setting up a Kotlin project is no different to any other language or library that integrates with these tools. Where there are some minor requirements and differences is when using the internal build system of IntelliJ IDEA, which is also supported on TeamCity.

Gradle, Maven, and Ant

If using Ant, Maven or Gradle, the setup process is straightforward. All that is needed is to define the Build Step. For example, if using Gradle, simply define the required parameters such as the Step Name and Gradle tasks that need executing for the Runner Type.

Build Step

Runner type: 
Runner for Gradle projects

Step name:
Optional, specify to distinguish this build step from other steps.

Execute step: 
Specify the step execution policy.

Gradle Parameters

Gradle tasks:
Enter task names separated by spaces, leave blank to use the 'default' task.
Example: ':myproject:clean :myproject:build' or 'clean build'.

Gradle build file:
Path to build file 

Gradle Build Step

Since all the dependencies required for Kotlin are defined in the Gradle file, nothing else needs to be configured specifically for Kotlin to run correctly.

If using Ant or Maven, the same configuration applies. The only difference being that the Runner Type would be Ant or Maven respectively.

IntelliJ IDEA Build System

If using IntelliJ IDEA build system with TeamCity, make sure that the version of Kotlin being used by IntelliJ IDEA is the same as the one that TeamCity runs. You may need to download the specific version of the Kotlin plugin and install it on TeamCity.

Fortunately, there is a meta-runner already available that takes care of most of the manual work. If not familiar with the concept of TeamCity meta-runners, check the [documentation](#). They are very easy and powerful way to introduce custom Runners without the need to write plugins.

Download and install the meta-runner

The meta-runner for Kotlin is available on [GitHub](#). Download that meta-runner and import it from the TeamCity user interface

The screenshot shows the 'Project Settings' sidebar on the left with various tabs like General Settings, VCS Roots, Report Tabs, Parameters, Builds Schedule, Issue Trackers, Maven Settings, SSH Keys, Shared Resources, and Meta-Runners (which is currently selected). The main content area is titled 'Meta-Runners' and contains a message: 'Meta-Runner is a generalized build step that can be used across different configuration'. Below this is a button labeled '+ Upload Meta-Runner' with a hand cursor icon pointing at it. A message below says 'There are no meta-runners defined in the current project.' Further down, it says 'Meta-runners inherited from <Root project>:' followed by a table listing two entries:

Runner ID	Name
ClassesVersionCheckerMR	Java Classes Version Checker Runs Java Classes Version checker tool
kotlinc	Setup kotlinc Downloads and registers Kotlin compiler into Inte...

Meta-runner

Setup Kotlin compiler fetching step

Basically this step is limited to defining the Step Name and the version of Kotlin you need. Tags can be used.

New Build Step

Runner type:

Setup kotlinc 

Downloads and registers Kotlin compiler into IntelliJ IDEA build runner

Step name:

Optional, specify to distinguish this build step from other steps.

Execute step: 

If all previous steps finished successfully 

Specify the step execution policy.

Kotlin Version

M10 

Select Kotlin build tag, e.g. 'M9 or bootstrap'

Setup Kotlin Compiler

The runner will set the value for the property system.path.macro.KOTLIN.BUNDLED to the correct one based on the path settings from the IntelliJ IDEA project. However, this value needs to be defined in TeamCity (and can be set to any value). Therefore, you need to define it as a system variable.

Setup Kotlin compilation step

The final step is to define the actual compilation of the project, which uses the standard IntelliJ IDEA Runner Type.

New Build Step

Runner type:

IntelliJ IDEA Project 

Runner for IntelliJ IDEA projects

Step name:

Optional, specify to distinguish this build step from other steps.

Project Settings

Path to the project: 

Should reference path to project file (.ipr) or project directory for directory-based (.idea) the checkout directory. Leave empty if .idea directory is right under the checkout direct

Check/Reparse Project

IntelliJ IDEA Runner

With that, our project should now build and produce the corresponding artifacts.

Other CI servers

If using a continuous integration tool different to TeamCity, as long as it supports any of the build tools, or calling

command line tools, compiling Kotlin and automating things as part of a CI process should be possible.

Document Kotlin code: KDoc and Dokka

The language used to document Kotlin code (the equivalent of Java's JavaDoc) is called KDoc. In its essence, KDoc combines JavaDoc's syntax for block tags (extended to support Kotlin's specific constructs) and Markdown for inline markup.

Generate the documentation

Kotlin's documentation generation tool is called [Dokka](#). See the [Dokka README](#) for usage instructions.

Dokka has plugins for Gradle, Maven, and Ant, so you can integrate documentation generation into your build process.

KDoc syntax

Just like with JavaDoc, KDoc comments start with `/**` and end with `*/`. Every line of the comment may begin with an asterisk, which is not considered part of the contents of the comment.

By convention, the first paragraph of the documentation text (the block of text until the first blank line) is the summary description of the element, and the following text is the detailed description.

Every block tag begins on a new line and starts with the `@` character.

Here's an example of a class documented using KDoc:

```
/**
 * A group of *members*.
 *
 * This class has no useful logic; it's just a documentation example.
 *
 * @param T the type of a member in this group.
 * @property name the name of this group.
 * @constructor Creates an empty group.
 */
class Group<T>(val name: String) {
    /**
     * Adds a [member] to this group.
     * @return the new size of the group.
     */
    fun add(member: T): Int { ... }
}
```

Block tags

KDoc currently supports the following block tags:

@param name

Documents a value parameter of a function or a type parameter of a class, property or function. To better separate the parameter name from the description, if you prefer, you can enclose the name of the parameter in brackets. The following two syntaxes are therefore equivalent:

`@param name description.` `@param[name] description.`

@return

Documents the return value of a function.

@constructor

Documents the primary constructor of a class.

@receiver

Documents the receiver of an extension function.

@property name

Documents the property of a class which has the specified name. This tag can be used for documenting properties declared in the primary constructor, where putting a doc comment directly before the property definition would be awkward.

@throws class, @exception class

Documents an exception which can be thrown by a method. Since Kotlin does not have checked exceptions, there is also no expectation that all possible exceptions are documented, but you can still use this tag when it provides useful information for users of the class.

@sample identifier

Embeds the body of the function with the specified qualified name into the documentation for the current element, in order to show an example of how the element could be used.

@see identifier

Adds a link to the specified class or method to the See also block of the documentation.

@author

Specifies the author of the element being documented.

@since

Specifies the version of the software in which the element being documented was introduced.

@suppress

Excludes the element from the generated documentation. Can be used for elements which are not part of the official API of a module but still have to be visible externally.

KDoc does not support the @deprecated tag. Instead, please use the @Deprecated annotation.

Inline markup

For inline markup, KDoc uses the regular [Markdown](#) syntax, extended to support a shorthand syntax for linking to other elements in the code.

Links to elements

To link to another element (class, method, property, or parameter), simply put its name in square brackets:

Use the method [foo] for this purpose.

If you want to specify a custom label for the link, use the Markdown reference-style syntax:

Use [this method][foo] for this purpose.

You can also use qualified names in the links. Note that, unlike JavaDoc, qualified names always use the dot character to separate the components, even before a method name:

Use [kotlin.reflect.KClass.properties] to enumerate the properties of the class.

Names in links are resolved using the same rules as if the name was used inside the element being documented. In particular, this means that if you have imported a name into the current file, you don't need to fully qualify it when you use it in a KDoc comment.

Note that KDoc does not have any syntax for resolving overloaded members in links. Since the Kotlin documentation generation tool puts the documentation for all overloads of a function on the same page, identifying a specific overloaded function is not required for the link to work.

Module and package documentation

Documentation for a module as a whole, as well as packages in that module, is provided as a separate Markdown file, and the paths to that file is passed to Dokka using the -include command line parameter or the corresponding parameters in Ant, Maven and Gradle plugins.

Inside the file, the documentation for the module as a whole and for individual packages is introduced by the corresponding first-level headings. The text of the heading must be Module <module name> for the module, and Package <package qualified name> for a package.

Here's an example content of the file:

```
# Module kotlin-demo

The module shows the Dokka syntax usage.

# Package org.jetbrains.kotlin.demo

Contains assorted useful stuff.

## Level 2 heading

Text after this heading is also part of documentation for `org.jetbrains.kotlin.demo`

# Package org.jetbrains.kotlin.demo2

Useful stuff in another package.
```

Kotlin and OSGi

To enable Kotlin [OSGi](#) support in your Kotlin project, include `kotlin-osgi-bundle` instead of the regular Kotlin libraries. It is recommended to remove `kotlin-runtime`, `kotlin-stdlib` and `kotlin-reflect` dependencies as `kotlin-osgi-bundle` already contains all of them. You also should pay attention in case when external Kotlin libraries are included. Most regular Kotlin dependencies are not OSGi-ready, so you shouldn't use them and should remove them from your project.

Maven

To include the Kotlin OSGi bundle to a Maven project:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-osgi-bundle</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

To exclude the standard library from external libraries (notice that "star exclusion" works in Maven 3 only):

```
<dependency>
    <groupId>some.group.id</groupId>
    <artifactId>some.library</artifactId>
    <version>some.library.version</version>
```

```
<exclusions>
    <exclusion>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>*</artifactId>
    </exclusion>
</exclusions>
</dependency>
```

Gradle

To include kotlin-osgi-bundle to a Gradle project:

```
compile "org.jetbrains.kotlin:kotlin-osgi-bundle:$kotlinVersion"
```

To exclude default Kotlin libraries that comes as transitive dependencies you can use the following approach:

```
dependencies {
    compile (
        [group: 'some.group.id', name: 'some.library', version: 'someversion'],
        ....) {
        exclude group: 'org.jetbrains.kotlin'
    }
}
```

FAQ

Why not just add required manifest options to all Kotlin libraries

Even though it is the most preferred way to provide OSGi support, unfortunately it couldn't be done for now due to so called "package split" issue that couldn't be easily eliminated and such a big change is not planned for now. There is Require-Bundle feature but it is not the best option too and not recommended to use. So it was decided to make a separate artifact for OSGi.

Learning materials overview

You can use the following materials and resources for learning Kotlin:

- [Basic syntax](#) – get a quick overview of the Kotlin syntax.
- [Idioms](#) – learn how to write idiomatic Kotlin code for popular cases.
- [Java to Kotlin migration guide: Strings](#) – learn how to perform typical tasks with strings in Java and Kotlin.

- [Kotlin Koans](#) – complete exercises to learn the Kotlin syntax. Each exercise is created as a failing unit test and your job is to make it pass.
- [Kotlin by example](#) – review a set of small and simple annotated examples for the Kotlin syntax.
- [Kotlin books](#) – find books we've reviewed and recommend for learning Kotlin.
- [Kotlin hands-on tutorials](#) – complete long-form tutorials to fully grasp a technology. These tutorials guide you through a self-contained project related to a specific topic.
- [Kotlin for Java Developers](#) - course for developers with experience in Java. It shows the similarities between the two languages and focuses on what's going to be different.

Kotlin Koans

Kotlin Koans are a series of exercises to get you familiar with the Kotlin Syntax. Each exercise is created as a failing unit test and your job is to make it pass. You can work with Kotlin Koans in one of following two ways:

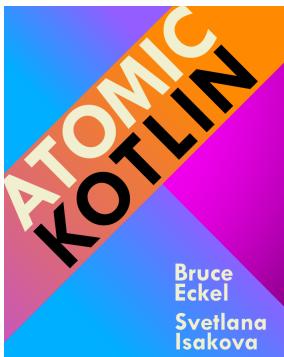
1. You can play with [Koans online](#).
2. You can perform the tasks right inside IntelliJ IDEA or Android Studio by [installing the EduTools plugin](#) and [choosing Kotlin Koans course](#).

Whatever way you choose to solve koans, you can see the solution for each task: in the online version click on "Show answer", inside EduTools plugin choose "Peek solution". We recommend checking the solution after you implement the task to compare your solution with the proposed one. Make sure you don't cheat!

Kotlin books

More and more authors write books for learning Kotlin in different languages. We are very thankful to all of them and appreciate all their efforts in helping us increase a number of professional Kotlin developers.

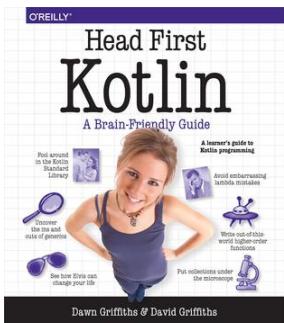
Here are just a few books we've reviewed and recommend you for learning Kotlin. You can find more books on [our community website](#).



[Atomic Kotlin](#) is for both beginning and experienced programmers!

Atomic Kotlin

From Bruce Eckel, author of the multi-award-winning Thinking in C++ and Thinking in Java, and Svetlana Isakova, Kotlin Developer Advocate at JetBrains, comes a book that breaks the language concepts into small, easy-to-digest “atoms”, along with a free course consisting of exercises supported by hints and solutions directly inside IntelliJ IDEA!

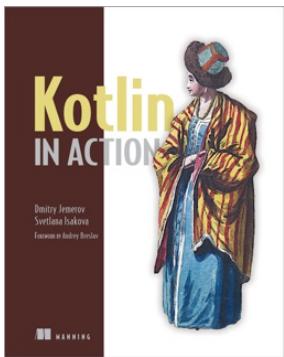


Head First Kotlin

[Head First Kotlin](#) is a complete introduction to coding in Kotlin. This hands-on book helps you learn the Kotlin language with a unique method that goes beyond syntax and how-to manuals and teaches you how to think like a great Kotlin developer.

You'll learn everything from language fundamentals to collections, generics, lambdas, and higher-order functions. Along the way, you'll get to play with both object-oriented and functional programming.

If you want to really understand Kotlin, this is the book for you.

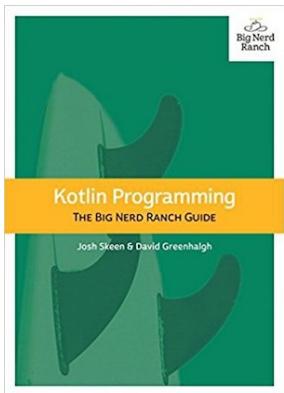


Kotlin in Action

[Kotlin in Action](#) teaches you to use the Kotlin language for production-quality applications. Written for experienced Java developers, this example-rich book goes further than most language books, covering interesting topics like building DSLs with natural language syntax.

The book is written by Dmitry Jemerov and Svetlana Isakova, developers on the Kotlin team.

Chapter 6, covering the Kotlin type system, and chapter 11, covering DSLs, are available as a free preview on the [publisher web site](#).



Kotlin Programming: The Big Nerd Ranch Guide

[Kotlin Programming: The Big Nerd Ranch Guide](#)

In this book you will learn to work effectively with the Kotlin language through carefully considered examples designed to teach you Kotlin's elegant style and features.

Starting from first principles, you will work your way to advanced usage of Kotlin, empowering you to create programs that are more reliable with less code.



Programming Kotlin

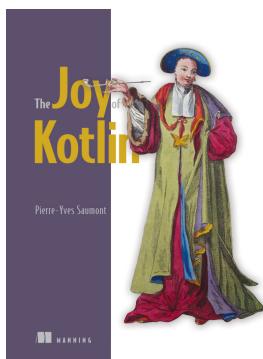
Programming Kotlin is written by Venkat Subramaniam.

Programmers don't just use Kotlin, they love it. Even Google has adopted it as a first-class language for Android development.

With Kotlin, you can intermix imperative, functional, and object-oriented styles of programming and benefit from the approach that's most suitable for the problem at hand.

Learn to use the many features of this highly concise, fluent, elegant, and expressive statically typed language with easy-to-understand examples.

Learn to write maintainable, high-performing JVM and Android applications, create DSLs, program asynchronously, and much more.



The Joy of Kotlin

[The Joy of Kotlin](#) teaches you the right way to code in Kotlin.

In this insight-rich book, you'll master the Kotlin language while exploring coding techniques that will make you a better developer no matter what language you use. Kotlin natively supports a functional style of programming, so seasoned author Pierre-Yves Saumont begins by reviewing the FP principles of immutability, referential transparency, and the separation between functions and effects.

Then, you'll move deeper into using Kotlin in the real world, as you learn to handle errors and data properly, encapsulate shared state mutations, and work with laziness.

This book will change the way you code — and give you back some of the joy you had when you first started.

Learning Kotlin with EduTools plugin

With the [EduTools plugin](#), available both in [Android Studio](#) and [IntelliJ IDEA](#), you can learn Kotlin through code practicing tasks.

Take a look at the [Learner Start Guide](#), which will get you started with the Kotlin Koans course inside IntelliJ IDEA. Solve interactive coding challenges and get instant feedback right inside the IDE.

If you want to use the EduTools plugin for teaching, read [Teaching Kotlin with EduTools plugin](#).

Teaching Kotlin with EduTools plugin

With the [EduTools plugin](#), available both in [Android Studio](#) and [IntelliJ IDEA](#), you can teach Kotlin through code practicing tasks. Take a look at the [Educator Start Guide](#) to learn how to create a simple Kotlin course that includes a set of programming tasks and integrated tests.

If you want to use the EduTools plugin to learn Kotlin, read [Learning Kotlin with EduTools plugin](#).

FAQ

What is Kotlin?

Kotlin is an open-source statically typed programming language that targets the JVM, Android, JavaScript and Native. It's developed by [JetBrains](#). The project started in 2010 and was open source from very early on. The first official 1.0 release was in February 2016.

What is the current version of Kotlin?

The currently released version is 1.5.20, published on June 24, 2021.

Is Kotlin free?

Yes. Kotlin is free, has been free and will remain free. It is developed under the Apache 2.0 license and the source code is available [on GitHub](#).

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

Kotlin has both object-oriented and functional constructs. You can use it in both OO and FP styles, or mix elements of the two. With first-class support for features such as higher-order functions, function types and lambdas, Kotlin is a great choice if you're doing or exploring functional programming.

What advantages does Kotlin give me over the Java programming language?

Kotlin is more concise. Rough estimates indicate approximately a 40% cut in the number of lines of code. It's also more type-safe, e.g. support for non-nullables makes applications less prone to NPE's. Other features including smart casting, higher-order functions, extension functions and lambdas with receivers provide the ability to write expressive code as well as facilitating creation of DSL.

Is Kotlin compatible with the Java programming language?

Yes. Kotlin is 100% interoperable with the Java programming language and major emphasis has been placed on making sure that your existing codebase can interact properly with Kotlin. You can easily call Kotlin code from Java and Java code from Kotlin. This makes adoption much easier and lower-risk. There's also an automated Java-to-Kotlin converter built into the IDE that simplifies migration of existing code.

What can I use Kotlin for?

Kotlin can be used for any kind of development, be it server-side, client-side web and Android. With Kotlin/Native currently in the works, support for other platforms such as embedded systems, macOS and iOS is coming. People are using Kotlin for mobile and server-side applications, client-side with JavaScript or JavaFX, and data science, just to name a few possibilities.

Can I use Kotlin for Android development?

Yes. Kotlin is supported as a first-class language on Android. There are hundreds of applications already using Kotlin for Android, such as Basecamp, Pinterest and more. For more information check out [the resource on Android development](#).

Can I use Kotlin for server-side development?

Yes. Kotlin is 100% compatible with the JVM and as such you can use any existing frameworks such as Spring Boot, vert.x or JSF. In addition there are specific frameworks written in Kotlin such as [Ktor](#). For more information check out [the resource on server-side development](#).

Can I use Kotlin for web development?

Yes. In addition to using for backend web, you can also use Kotlin/JS for client-side web. Kotlin can use definitions from [DefinitelyTyped](#) to get static typing for common JavaScript libraries, and it is compatible with existing module systems such as AMD and CommonJS. For more information check out [the resource on client-side development](#).

Can I use Kotlin for desktop development?

Yes. You can use any Java UI framework such as JavaFx, Swing or other. In addition there are Kotlin specific frameworks such as [TornadoFX](#).

Can I use Kotlin for native development?

Yes. Kotlin/Native is available as a part of Kotlin project. It compiles Kotlin to native code that can run without a VM. It is still in beta, but you can already try it on popular desktop and mobile platforms and even some IoT devices. For more information, check out the [Kotlin/Native documentation](#).

What IDEs support Kotlin?

Kotlin is supported by all major Java IDEs including [IntelliJ IDEA](#), [Android Studio](#), and [Eclipse](#). In addition, a [command line compiler](#) is available and provides straightforward support for compiling and running applications.

What build tools support Kotlin?

On the JVM side, the main build tools include [Gradle](#), [Maven](#), [Ant](#), and [Kobalt](#). There are also some build tools available that target client-side JavaScript.

What does Kotlin compile down to?

When targeting the JVM, Kotlin produces Java compatible bytecode.

When targeting JavaScript, Kotlin transpiles to ES5.1 and generates code which is compatible with module systems including AMD and CommonJS.

When targeting native, Kotlin will produce platform-specific code (via LLVM).

Which versions of JVM does Kotlin target?

Kotlin lets you choose the version of JVM for execution. By default, the Kotlin/JVM compiler produces Java 8 compatible bytecode. If you want to make use of optimizations available in newer versions of Java, you can explicitly specify the target Java version from 9 to 16. Note that in this case the resulting bytecode might not run on lower versions.

Is Kotlin hard?

Kotlin is inspired by existing languages such as Java, C#, JavaScript, Scala and Groovy. We've tried to ensure that Kotlin is easy to learn, so that people can easily jump on board, reading and writing Kotlin in a matter of days. Learning idiomatic Kotlin and using some more of its advanced features can take a little longer, but overall it is not a complicated language.

What companies are using Kotlin?

There are too many companies using Kotlin to list, but some more visible companies that have publicly declared usage of Kotlin, be this via blog posts, GitHub repositories or talks include [Square](#), [Pinterest](#), [Basecamp](#), and [Corda](#).

Who develops Kotlin?

Kotlin is primarily developed by a team of engineers at JetBrains (current team size is 100+). The lead language designer is [Roman Elizarov](#). In addition to the core team, there are also over 250 external contributors on GitHub.

Where can I learn more about Kotlin?

The best place to start is [this website](#). From there you can download the compiler, [try it online](#) as well as get access to resources.

Are there any books on Kotlin?

There are a number of books available for Kotlin. Some of them we have reviewed and can recommend to start with. They are listed on the [Books](#) page. For more books, see the community-maintained list at [kotlin.link](#).

Are there any online courses available for Kotlin?

There are a few courses available for Kotlin, including a [Pluralsight Kotlin Course](#) by Kevin Jones, an [O'Reilly Course](#) by Hadi Hariri and an [Udemy Kotlin Course](#) by Peter Sommerhoff.

There are also many recordings of [Kotlin talks](#) available on YouTube and Vimeo.

Does Kotlin have a community?

Yes. Kotlin has a very vibrant community. Kotlin developers hang out on the [Kotlin forums](#), [StackOverflow](#) and more actively on the [Kotlin Slack](#) (with close to 30000 members as of April 2020).

Are there Kotlin events?

Yes. There are many User Groups and Meetups now focused exclusively around Kotlin. You can find [a list on the web site](#). In addition, there are community-organized [Kotlin Nights](#) events around the world.

Is there a Kotlin conference?

Yes. The official annual [KotlinConf](#) is hosted by JetBrains. It took place in San-Francisco in [2017](#), Amsterdam in [2018](#), and Copenhagen in [2019](#). Kotlin is also being covered in different conferences worldwide. You can find a list of [upcoming talks on the web site](#).

Is Kotlin on social media?

Yes. The most active Kotlin account is [on Twitter](#).

Any other online Kotlin resources?

The web site has a bunch of [online resources](#), including [Kotlin Digests](#) by community members, a [newsletter](#), a

[podcast](#) and more.

Where can I get an HD Kotlin logo?

Logos can be downloaded [here](#). When using the logos, please follow simple rules in the guidelines.pdf inside the archive and [Kotlin brand usage guidelines](#).

Participate in the Kotlin Early Access Preview

You can participate in the Kotlin Early Access Preview (EAP) to try out the latest Kotlin features before they are released.

We ship a few Milestone (M) builds before every feature (1.x) and incremental (1.x.y) release.

We'll be very thankful if you find and report bugs to our issue tracker [YouTrack](#). It is very likely that we'll be able to fix them before the final release, which means you won't need to wait until the next Kotlin release for your issues to be addressed.

By participating in the Early Access Preview and reporting bugs, you contribute to Kotlin and help us make it better for everyone in [the growing Kotlin community](#). We appreciate your help a lot!

If you have any questions and want to participate in discussions, you are welcome to join the #eap channel in [Kotlin Slack](#). In this channel, you can also get notifications about new EAP builds.

[Install the Kotlin EAP Plugin for IDEA or Android Studio](#)

By participating in the EAP, you expressly acknowledge that the EAP version may not be reliable, may not work as intended, and may contain errors.

Please note that we don't provide any guarantees of compatibility between EAP and final versions of the same release.

If you have already installed the EAP version and want to work on projects that were created previously, check [our instructions on how to configure your build to support this version](#).

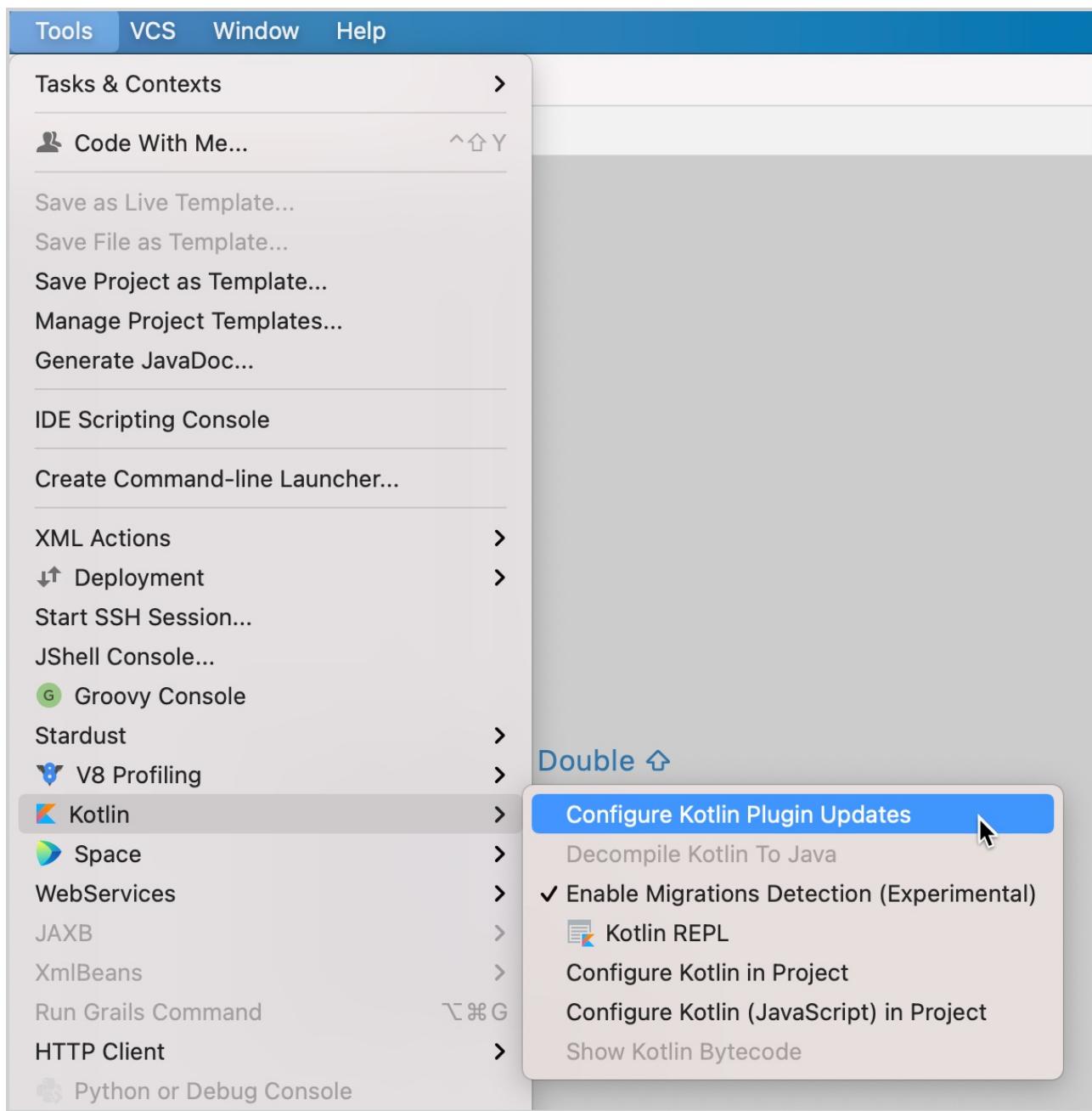
Build details

No preview versions are currently available.

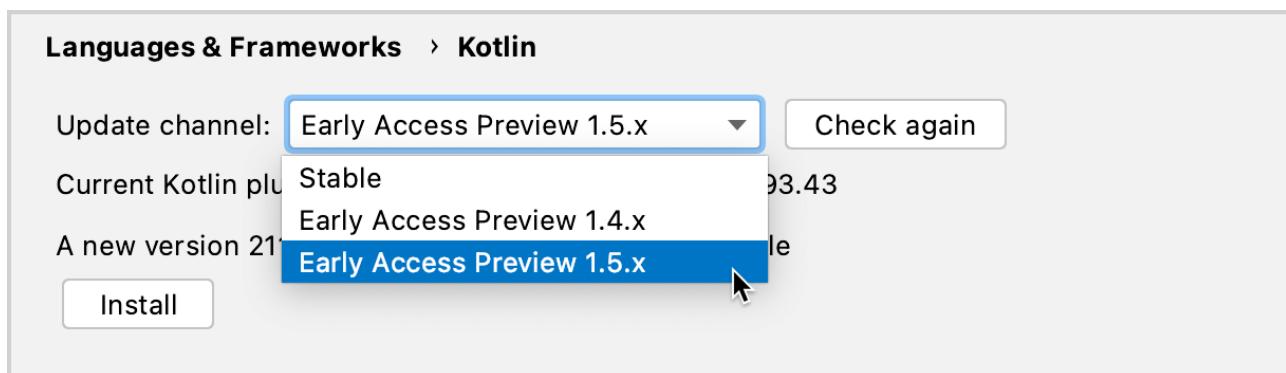
Install the EAP Plugin for IntelliJ IDEA or Android Studio

You can follow these instructions to install [the preview version of the Kotlin Plugin for IntelliJ IDEA or Android Studio](#).

1. Select Tools | Kotlin | Configure Kotlin Plugin Updates.

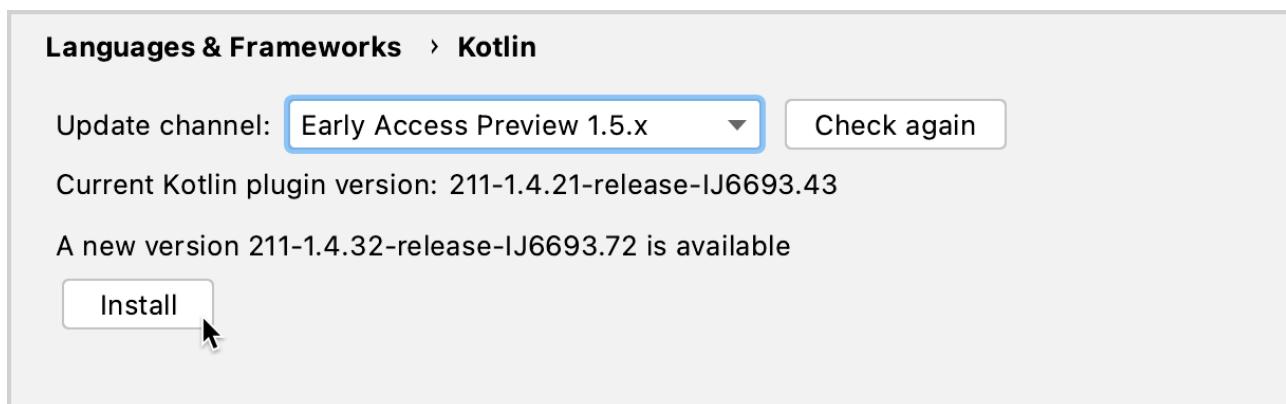


2. In the Update channel list, select the Early Access Preview X channel, where X is the latest version of Kotlin.



Select the EAP update channel

3. Click Check again. The latest EAP build version appears.



Install the EAP build

4. Click Install.

If you want to work on existing projects that were created before installing the EAP version, you need to [configure your build for EAP](#).

Configure your build for EAP

If you create new projects using the EAP version of Kotlin, you don't need to perform any additional steps. The [Kotlin Plugin](#) will do everything for you!

You only need to configure your build manually for existing projects — projects that were created before installing the EAP version.

To configure your build to use the EAP version of Kotlin, you need to:

- Specify the EAP version of Kotlin. [Available EAP versions are listed here](#).
- Change the versions of dependencies to EAP ones. The EAP version of Kotlin may not work with the libraries of the previously released version.

The following procedures describe how to configure your build in Gradle and Maven:

- [Configure in Gradle](#)
- [Configure in Maven](#)

Configure in Gradle

This section describes how you can:

- [Adjust the Kotlin version](#)
- [Adjust versions in dependencies](#)

Adjust the Kotlin version

In the plugins block within build.gradle(.kts), change the KOTLIN-EAP-VERSION to the actual EAP version, such as 1.5.20-RC. [Available EAP versions are listed here](#).

Alternatively, you can specify the EAP version in the pluginManagement block in settings.gradle(.kts) – see [Gradle documentation](#) for details.

Here is an example for the Multiplatform project.

Groovy

```
plugins {  
    id 'java'  
    id 'org.jetbrains.kotlin.multiplatform' version 'KOTLIN-EAP-VERSION'  
}  
  
repositories {  
    mavenCentral()  
}
```

Kotlin

```
plugins {  
    java  
    kotlin("multiplatform") version "KOTLIN-EAP-VERSION"  
}  
  
repositories {  
    mavenCentral()  
}
```

Adjust versions in dependencies

If you use kotlinx libraries in your project, your versions of the libraries may not be compatible with the EAP version of Kotlin.

To resolve this issue, you need to specify the version of a compatible library in dependencies. For a list of compatible libraries, see [EAP build details](#).

In most cases we create libraries only for the first EAP version of a specific release and these libraries work with the subsequent EAP versions for this release.

If there are incompatible changes in next EAP versions, we release a new version of the library.

Here is an example.

For the kotlinx.coroutines library, add the version number – 1.5.0-RC – that is compatible with 1.5.20-RC.

Groovy

```
dependencies {  
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.0-RC"  
}
```

Kotlin

```
dependencies {  
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.0-RC")  
}
```

Configure in Maven

In the sample Maven project definition, replace KOTLIN-EAP-VERSION with the actual version, such as 1.5.20-RC.

[Available EAP versions are listed here](#).

```
<project ...>  
  <properties>  
    <kotlin.version>KOTLIN-EAP-VERSION</kotlin.version>  
  </properties>  
  
  <repositories>  
    <repository>  
      <id>mavenCentral</id>  
      <url>https://repo1.maven.org/maven2/</url>  
    </repository>  
  </repositories>
```

```

<pluginRepositories>
    <pluginRepository>
        <id>mavenCentral</id>
        <url>https://repo1.maven.org/maven2/</url>
    </pluginRepository>
</pluginRepositories>

<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            ...
        </plugin>
    </plugins>
</build>
</project>

```

Contribution

Kotlin is an open-source project under the [Apache 2.0 License](#). The source code, tooling, documentation, and even this web site are maintained on [GitHub](#). While Kotlin is mostly developed by JetBrains, there are hundreds of external contributors to the Kotlin project and we are always on the lookout for more people to help us.

Participate in Early Access Preview

You can help us improve Kotlin by [participating in Kotlin Early Access Preview \(EAP\)](#) and providing us with your valuable feedback.

For every release, Kotlin ships a few preview builds where you can try out the latest features before they go to production. You can report any bugs you find to our issue tracker [YouTrack](#) and we will try to fix them before a final release. This way, you can get bug fixes earlier than the standard Kotlin release cycle.

Contribute to the compiler, standard library, and tooling

If you want to contribute to the Kotlin compiler, standard library, or tooling, go to [GitHub](#), check out the latest Kotlin version, and follow [the instructions on how to contribute](#).

You can help us by completing [open tasks](#). Please keep an open line of communication with us because we may have questions and comments on your changes. Otherwise, we won't be able to incorporate your contributions.

Contribute to other libraries and create your own

Besides the standard library that provides core capabilities, Kotlin has a number of additional (kotlinx) libraries that extend its functionality. Each kotlinx library is developed in a separate repository, has its own versioning and release cycle.

If you want to contribute to a kotlinx library, follow the contribution process described for each library, such as [kotlinx.coroutines](#), [kotlinx.serialization](#), and [ktor](#).

If you have a library that could be useful to other Kotlin developers, let us know via feedback@kotlinlang.org.

Contribute to the documentation

If you've found an issue in the Kotlin documentation, feel free to check out [the documentation source code on GitHub](#) and send us a pull request. Follow [these guidelines on style and formatting](#).

Please keep an open line of communication with us because we may have questions and comments on your changes. Otherwise, we won't be able to incorporate your contributions.

Create tutorials or videos

If you've created tutorials or videos for Kotlin, please share them with us via feedback@kotlinlang.org.

Translate documentation to other languages

You are welcome to translate the Kotlin documentation into your own language and publish the translation on your website. However, we won't be able to host your translation in the main repository and publish it on kotlinlang.org.

This site is the official documentation for the language, and we ensure that all the information here is correct and up to date. Unfortunately, we won't be able to review documentation in other languages.

Hold events and presentations

If you've given or just plan to give presentations or hold events on Kotlin, please fill out [the form](#). We'll feature them on [the event list](#).

Kotlin Evolution

Principles of Pragmatic Evolution

Language design is cast in stone,
but this stone is reasonably soft,
and with some effort we can reshape it later.

Kotlin Design Team

Kotlin is designed to be a pragmatic tool for programmers. When it comes to language evolution, its pragmatic nature is captured by the following principles:

- Keep the language modern over the years.
- Stay in the constant feedback loop with the users.
- Make updating to new versions comfortable for the users.

As this is key to understanding how Kotlin is moving forward, let's expand on these principles.

Keeping the Language Modern. We acknowledge that systems accumulate legacy over time. What had once been cutting-edge technology can be hopelessly outdated today. We have to evolve the language to keep it relevant to the needs of the users and up-to-date with their expectations. This includes not only adding new features, but also phasing out old ones that are no longer recommended for production use and have altogether become legacy.

Comfortable Updates. Incompatible changes, such as removing things from a language, may lead to painful migration from one version to the next if carried out without proper care. We will always announce such changes well in advance, mark things as deprecated and provide automated migration tools before the change happens. By the time the language is changed we want most of the code in the world to be already updated and thus have no issues migrating to the new version.

Feedback Loop. Going through deprecation cycles requires significant effort, so we want to minimize the number of incompatible changes we'll be making in the future. Apart from using our best judgement, we believe that trying things out in real life is the best way to validate a design. Before casting things in stone we want them battle-tested. This is why we use every opportunity to make early versions of our designs available in production versions of the language, but in one of the pre-stable statuses: [Experimental](#), [Alpha](#), or [Beta](#). Such features are not stable, they can be changed at any time, and the users that opt into using them do so explicitly to indicate that they are ready to deal with the future migration issues. These users provide invaluable feedback that we gather to iterate on the design and make it rock-solid.

Incompatible changes

If, upon updating from one version to another, some code that used to work doesn't work any more, it is an incompatible change in the language (sometimes referred to as "breaking change"). There can be debates as to what "doesn't work any more" means precisely in some cases, but it definitely includes the following:

- Code that compiled and ran fine is now rejected with an error (at compile or link time). This includes removing language constructs and adding new restrictions.
- Code that executed normally is now throwing an exception.

The less obvious cases that belong to the "grey area" include handling corner cases differently, throwing an exception of a different type than before, changing behavior observable only through reflection, changes in undocumented/undefined behavior, renaming binary artifacts, etc. Sometimes such changes are very important and affect migration experience dramatically, sometimes they are insignificant.

Some examples of what definitely isn't an incompatible change include

- Adding new warnings.
- Enabling new language constructs or relaxing limitations for existing ones.
- Changing private/internal APIs and other implementation details.

The principles of Keeping the Language Modern and Comfortable Updates suggest that incompatible changes are sometimes necessary, but they should be introduced carefully. Our goal is to make the users aware of upcoming changes well in advance to let them migrate their code comfortably.

Ideally, every incompatible change should be announced through a compile-time warning reported in the problematic code (usually referred to as a deprecation warning) and accompanied with automated migration aids. So, the ideal migration workflow goes as follows:

- Update to version A (where the change is announced)
 - See warnings about the upcoming change
 - Migrate the code with the help of the tooling
- Update to version B (where the change happens)
 - See no issues at all

In practice some changes can't be accurately detected at compile time, so no warnings can be reported, but at least the users will be notified through Release notes of version A that a change is coming in version B.

Dealing with compiler bugs

Compilers are complicated software and despite the best effort of their developers they have bugs. The bugs that cause the compiler itself to fail or report spurious errors or generate obviously failing code, though annoying and often embarrassing, are easy to fix, because the fixes do not constitute incompatible changes. Other bugs may cause the compiler to generate incorrect code that does not fail: e.g. by missing some errors in the source or simply generating

wrong instructions. Fixes of such bugs are technically incompatible changes (some code used to compile fine, but now it won't any more), but we are inclined to fixing them as soon as possible to prevent the bad code patterns from spreading across user code. In our opinion, this serves the principle of Comfortable Updates, because fewer users have a chance of encountering the issue. Of course, this applies only to bugs that are found soon after appearing in a released version.

Decision making

[JetBrains](#), the original creator of Kotlin, is driving its progress with the help of the community and in accord with the [Kotlin Foundation](#).

All changes to the Kotlin Programming Language are overseen by the [Lead Language Designer](#) (currently Roman Elizarov). The Lead Designer has the final say in all matters related to language evolution. Additionally, incompatible changes to fully stable components have to be approved to by the [Language Committee](#) designated under the [Kotlin Foundation](#) (currently comprised of Jeffrey van Gogh, William R. Cook and Roman Elizarov).

The Language Committee makes final decisions on what incompatible changes will be made and what exact measures should be taken to make user updates comfortable. In doing so, it relies on a set of guidelines available [here](#).

Feature releases and incremental releases

Stable releases with versions 1.2, 1.3, etc. are usually considered to be feature releases bringing major changes in the language. Normally, we publish incremental releases, numbered 1.2.20, 1.2.30, etc, in between feature releases.

Incremental releases bring updates in the tooling (often including features), performance improvements and bug fixes. We try to keep such versions compatible with each other, so changes to the compiler are mostly optimizations and warning additions/removals. Pre-stable features may, of course, be added, removed or changed at any time.

Feature releases often add new features and may remove or change previously deprecated ones. Feature graduation from pre-stable to stable also happens in feature releases.

EAP builds

Before releasing stable versions, we usually publish a number of preview builds dubbed EAP (for "Early Access Preview") that let us iterate faster and gather feedback from the community. EAPs of feature releases usually produce binaries that will be later rejected by the stable compiler to make sure that possible bugs in the binary format survive no longer than the preview period. Final Release Candidates normally do not bear this limitation.

Pre-stable features

According to the Feedback Loop principle described above, we iterate on our designs in the open and release versions of the language where some features have one of the pre-stable statuses and are supposed to change. Such features can be added, changed or removed at any point and without warning. We do our best to ensure that pre-stable features can't be used accidentally by an unsuspecting user. Such features usually require some sort of an explicit opt-

in either in the code or in the project configuration.

Pre-stable features usually graduate to the stable status after some iterations.

Status of different components

To check the stability status of different components of Kotlin (Kotlin/JVM, JS, Native, various libraries, etc), please consult [this link](#).

Libraries

A language is nothing without its ecosystem, so we pay extra attention to enabling smooth library evolution.

Ideally, a new version of a library can be used as a "drop-in replacement" for an older version. This means that upgrading a binary dependency should not break anything, even if the application is not recompiled (this is possible under dynamic linking).

On the one hand, to achieve this, the compiler has to provide certain ABI stability guarantees under the constraints of separate compilation. This is why every change in the language is examined from the point of view of binary compatibility.

On the other hand, a lot depends on the library authors being careful about which changes are safe to make. Thus it's very important that library authors understand how source changes affect compatibility and follow certain best practices to keep both APIs and ABIs of their libraries stable. Here are some assumptions that we make when considering language changes from the library evolution standpoint:

- Library code should always specify return types of public/protected functions and properties explicitly thus never relying on type inference for public API. Subtle changes in type inference may cause return types to change inadvertently, leading to binary compatibility issues.
- Overloaded functions and properties provided by the same library should do essentially the same thing. Changes in type inference may result in more precise static types to be known at call sites causing changes in overload resolution.

Library authors can use the `@Deprecated` and `@RequiresOptIn` annotations to control the evolution of their API surface. Note that `@Deprecated(level=HIDDEN)` can be used to preserve binary compatibility even for declarations removed from the API.

Also, by convention, packages named "internal" are not considered public API. All API residing in packages named "experimental" is considered pre-stable and can change at any moment.

We evolve the Kotlin Standard Library (`kotlin-stdlib`) for stable platforms according to the principles stated above. Changes to the contracts for its API undergo the same procedures as changes in the language itself.

Compiler keys

Command line keys accepted by the compiler are also a kind of public API, and they are subject to the same considerations. Supported flags (those that don't have the "-X" or "-XX" prefix) can be added only in feature releases and should be properly deprecated before removing them. The "-X" and "-XX" flags are experimental and can be added and removed at any time.

Compatibility tools

As legacy features get removed and bugs fixed, the source language changes, and old code that has not been properly migrated may not compile any more. The normal deprecation cycle allows a comfortable period of time for migration, and even when it's over and the change ships in a stable version, there's still a way to compile unmigrated code.

Compatibility flags

We provide the -language-version and -api-version flags that make a new version emulate the behavior of an old one, for compatibility purposes. Normally, at least one previous version is supported. This effectively leaves a time span of two full feature release cycles for migration (which usually amounts to about two years). Using an older kotlin-stdlib or kotlin-reflect with a newer compiler without specifying compatibility flags is not recommended, and the compiler will report a [warning](#) when this happens.

Actively maintained code bases can benefit from getting bug fixes ASAP, without waiting for a full deprecation cycle to complete. Currently such project can enable the -progressive flag and get such fixes enabled even in incremental releases.

All flags are available on the command line as well as [Gradle](#) and [Maven](#).

Evolving the binary format

Unlike sources that can be fixed by hand in the worst case, binaries are a lot harder to migrate, and this makes backwards compatibility very important in the case of binaries. Incompatible changes to binaries can make updates very uncomfortable and thus should be introduced with even more care than those in the source language syntax.

For fully stable versions of the compiler the default binary compatibility protocol is the following:

- All binaries are backwards compatible, i.e. a newer compiler can read older binaries (e.g. 1.3 understands 1.0 through 1.2),
- Older compilers reject binaries that rely on new features (e.g. a 1.0 compiler rejects binaries that use coroutines).
- Preferably (but we can't guarantee it), the binary format is mostly forwards compatible with the next feature release, but not later ones (in the cases when new features are not used, e.g. 1.3 can understand most binaries from 1.4, but not 1.5).

This protocol is designed for comfortable updates as no project can be blocked from updating its dependencies even if it's using a slightly outdated compiler.

Please note that not all target platforms have reached this level of stability (but Kotlin/JVM has).

Stability of Kotlin components

The Kotlin language and toolset are divided into many components such as the compilers for the JVM, JS and Native targets, the Standard Library, various accompanying tools and so on. Many of these components were officially released as Stable which means that they are evolved in the backward-compatible way following the [principles of Comfortable Updates and Keeping the Language Modern](#). Among such stable components are, for example, the Kotlin compiler for the JVM, the Standard Library, and Coroutines.

Following the Feedback Loop principle we release many things early for the community to try out, so a number of components are not yet released as Stable. Some of them are very early stage, some are more mature. We mark them as Experimental, Alpha or Beta depending on how quickly each component is evolving and how much risk the users are taking when adopting it.

Stability levels explained

Here's a quick guide to these stability levels and their meaning:

Experimental means "try it only in toy projects":

- We are just trying out an idea and want some users to play with it and give feedback. If it doesn't work out, we may drop it any minute.

Alpha means "use at your own risk, expect migration issues":

- We decided to productize this idea, but it hasn't reached the final shape yet.

Beta means "you can use it, we'll do our best to minimize migration issues for you":

- It's almost done, user feedback is especially important now.
- Still, it's not 100% finished, so changes are possible (including ones based on your own feedback).
- Watch for deprecation warnings in advance for the best update experience.

We collectively refer to Experimental, Alpha and Beta as pre-stable levels.

Stable means "use it even in most conservative scenarios":

- It's done. We will be evolving it according to our strict [backward compatibility rules](#).

Please note that stability levels do not say anything about how soon a component will be released as Stable. Similarly, they do not indicate how much a component will be changed before release. They only say how fast a component is changing and how much risk of update issues users are running.

Stability of subcomponents

A stable component may have an experimental subcomponent, for example:

- a stable compiler may have an experimental feature;
- a stable API may include experimental classes or functions;
- a stable command-line tool may have experimental options.

We make sure to document precisely which subcomponents are not stable. We also do our best to warn users where possible and ask to opt in explicitly to avoid accidental usages of features that have not been released as stable.

Current stability of Kotlin components

Component	Status	Status since version	Comment
Kotlin/JVM	Stable	1.0	
kotlin-stdlib (JVM)	Stable	1.0	
Coroutines	Stable	1.3	
kotlin-reflect (JVM)	Beta	1.0	
Kotlin/JS (Classic back-end)	Stable	1.3	
Kotlin/JVM (IR-based)	Alpha	1.4	
Kotlin/JS (IR-based)	Alpha	1.4	
Kotlin/Native Runtime	Beta	1.3	
KLib binaries	Alpha	1.4	
KDoc syntax	Stable	1.0	
dokka	Alpha	0.1	
Kotlin Scripts (*.kts)	Beta	1.2	
Kotlin Scripting APIs and custom hosts	Alpha	1.2	
Compiler Plugin API	Experimental	1.0	

Component	Status	Status since version	Comment
Serialization Compiler Plugin	Stable	1.4	
Serialization Core Library	Stable	1.0.0	Versioned separately from the language
Multiplatform Projects	Alpha	1.3	
expect/actual language feature	Beta	1.2	
Inline classes	Stable	1.5	
Unsigned arithmetics	Beta	1.3	
Contracts in stdlib	Stable	1.3	
User-defined contracts	Experimental	1.3	
All other experimental components, by default	Experimental	N/A	

[The pre-1.4 version of this page is available here.](#)

Stability of Kotlin components (pre 1.4)

There can be different modes of stability depending of how quickly a component is evolving:

- Moving fast (MF): no compatibility should be expected between even incremental releases, any functionality can be added, removed or changed without warning.
- Additions in Incremental Releases (AIR): things can be added in an incremental release, removals and changes of behavior should be avoided and announced in a previous incremental release if necessary.
- Stable Incremental Releases (SIR): incremental releases are fully compatible, only optimizations and bug fixes happen. Any changes can be made in a feature release.
- Fully Stable (FS): incremental releases are fully compatible, only optimizations and bug fixes happen. Feature releases are backwards compatible.

Source and binary compatibility may have different modes for the same component, e.g. the source language can reach full stability before the binary format stabilizes, or vice versa.

The provisions of the Kotlin evolution policy fully apply only to components that have reached Full Stability (FS). From

that point on incompatible changes have to be approved by the Language Committee.

Component	Status Entered at version	Mode for Sources	Mode for Binaries
Kotlin/JVM	1.0	FS	FS
kotlin-stdlib (JVM)	1.0	FS	FS
KDoc syntax	1.0	FS	N/A
Coroutines	1.3	FS	FS
kotlin-reflect (JVM)	1.0	SIR	SIR
Kotlin/JS	1.1	AIR	MF
Kotlin/Native	1.3	AIR	MF
Kotlin Scripts (*.kts)	1.2	AIR	MF
dokka	0.1	MF	N/A
Kotlin Scripting APIs	1.2	MF	MF
Compiler Plugin API	1.0	MF	MF
Serialization	1.3	MF	MF
Multiplatform Projects	1.2	MF	MF
Inline classes	1.3	MF	MF
Unsigned arithmetics	1.3	MF	MF
All other experimental features, by default	N/A	MF	MF

Compatibility guide for Kotlin 1.5

Keeping the Language Modern and Comfortable Updates are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or

compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.4 to Kotlin 1.5.

Basic terms

In this document we introduce several kinds of compatibility:

- source: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- binary: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- behavioral: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language and stdlib

Forbid spread operator in signature-polymorphic calls

Issue: [KT-35226](#)

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw the use of spread operator (*) on signature-polymorphic calls

Deprecation cycle:

- < 1.5: introduce warning for the problematic operator at call-site
- >= 1.5: raise this warning to an error, -XXLanguage:-ProhibitSpreadOnSignaturePolymorphicCall can be used to temporarily revert to pre-1.5 behavior

Forbid non-abstract classes containing abstract members invisible from that classes (internal/package-private)

Issue: [KT-27825](#)

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw non-abstract classes containing abstract members invisible from that classes (internal/package-private)

Deprecation cycle:

- < 1.5: introduce warning for the problematic classes
- >= 1.5: raise this warning to an error, `-XXLanguage:-ProhibitInvisibleAbstractMethodsInSuperclasses` can be used to temporarily revert to pre-1.5 behavior

Forbid using array based on non-reified type parameters as reified type arguments on JVM

Issue: [KT-31227](#)

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw using array based on non-reified type parameters as reified type arguments on JVM

Deprecation cycle:

- < 1.5: introduce warning for the problematic calls
- >= 1.5: raise this warning to an error, `-XXLanguage:-ProhibitNonReifiedArraysAsReifiedTypeArguments` can be used to temporarily revert to pre-1.5 behavior

Forbid secondary enum class constructors which do not delegate to the primary constructor

Issue: [KT-35870](#)

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw secondary enum class constructors which do not delegate to the primary constructor

Deprecation cycle:

- < 1.5: introduce warning for the problematic constructors
- >= 1.5: raise this warning to an error, -XXLanguage:-RequiredPrimaryConstructorDelegationCallInEnums can be used to temporarily revert to pre-1.5 behavior

Forbid exposing anonymous types from private inline functions

Issue: [KT-33917](#)

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw exposing anonymous types from private inline functions

Deprecation cycle:

- < 1.5: introduce warning for the problematic constructors
- >= 1.5: raise this warning to an error, -XXLanguage:-ApproximateAnonymousReturnTypesInPrivateInlineFunctions can be used to temporarily revert to pre-1.5 behavior

Forbid passing non-spread arrays after arguments with SAM-conversion

Issue: [KT-35224](#)

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw passing non-spread arrays after arguments with SAM-conversion

Deprecation cycle:

- 1.3.70: introduce warning for the problematic calls
- >= 1.5: raise this warning to an error, -XXLanguage:-ProhibitVarargAsArrayAfterSamArgument can be used to temporarily revert to pre-1.5 behavior

Support special semantics for underscore-named catch block parameters

Issue: [KT-31567](#)

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw references to the underscore symbol (_) that is used to omit parameter name of an exception in the catch block

Deprecation cycle:

- 1.4.20: introduce warning for the problematic references
- >= 1.5: raise this warning to an error, > -XXLanguage:-ForbidReferencingToUnderscoreNamedParameterOfCatchBlock can be used to temporarily revert to pre-1.5 behavior

Change implementation strategy of SAM conversion from anonymous class-based to invokedynamic

Issue: [KT-44912](#)

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since Kotlin 1.5, implementation strategy of SAM (single abstract method) conversion will be changed from generating an anonymous class to using the invokedynamic JVM instruction

Deprecation cycle:

- 1.5: change implementation strategy of SAM conversion, `-Xsam-conversions=class` can be used to revert implementation scheme to the one that used before

Generate nullability assertion for delegated properties with a generic call in the delegate expression

Issue: [KT-44304](#)

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since Kotlin 1.5, the Kotlin compiler will emit nullability assertions for delegated properties with a generic call in the delegate expression

Deprecation cycle:

- 1.5: emit nullability assertion for delegated properties (see details in the issue), `-Xuse-old-backend` or `-language-version 1.4` can be used to temporarily revert to pre-1.5 behavior

Turn warnings into errors for calls with type parameters annotated by `@OnlyInputTypes`

Issue: [KT-45861](#)

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.5 will outlaw calls like contains, indexOf, and assertEquals with senseless arguments to improve type safety

Deprecation cycle:

- 1.4.0: introduce warning for the problematic constructors
- >= 1.5: raise this warning to an error, -XXLanguage:-StrictOnlyInputTypesChecks can be used to temporarily revert to pre-1.5 behavior

Use the correct order of arguments execution in calls with named vararg

Issue: [KT-17691](#)

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin 1.5 will change the order of arguments execution in calls with named vararg

Deprecation cycle:

- < 1.5: introduce warning for the problematic constructors
- >= 1.5: raise this warning to an error, -XXLanguage:-UseCorrectExecutionOrderForVarargArguments can be used to temporarily revert to pre-1.5 behavior

Use default value of the parameter in operator functional calls

Issue: [KT-42064](#)

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin 1.5 will use default value of the parameter in operator calls

Deprecation cycle:

- < 1.5: old behavior (see details in the issue)
- >= 1.5: behavior changed, -XXLanguage:-JvmlrEnabledByDefault can be used to temporarily revert to pre-1.5 behavior

Produce empty reversed progressions in for loops if regular progression is also empty

Issue: [KT-42533](#)

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin 1.5 will produce empty reversed progressions in for loops if regular progression is also empty

Deprecation cycle:

- < 1.5: old behavior (see details in the issue)
- >= 1.5: behavior changed, -XXLanguage:-JvmlrEnabledByDefault can be used to temporarily revert to pre-1.5 behavior

Straighten Char-to-code and Char-to-digit conversions out

Issue: [KT-23451](#)

Component: kotlin-stdlib

Incompatible change type: source

Short summary: Since Kotlin 1.5, conversions of Char to number types will be deprecated

Deprecation cycle:

- 1.5: deprecate Char.toInt()/toShort()/toLong()/toByte()/toDouble()/toFloat() and the reverse functions like Long.toChar(), and propose replacement

Inconsistent case-insensitive comparison of characters in kotlin.text functions

Issue: [KT-45496](#)

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: Since Kotlin 1.5, Char.equals will be improved in case-insensitive case by first comparing whether the uppercase variants of characters are equal, then whether the lowercase variants of those uppercase variants (as opposed to the characters themselves) are equal

Deprecation cycle:

- < 1.5: old behavior (see details in the issue)
- 1.5: change behavior for Char.equals function

Remove default locale-sensitive case conversion API

Issue: [KT-43023](#)

Component: kotlin-stdlib

Incompatible change type: source

Short summary: Since Kotlin 1.5, default locale-sensitive case conversion functions like `String.toUpperCase()` will be deprecated

Deprecation cycle:

- 1.5: deprecate case conversions functions with the default locale (see details in the issue), and propose replacement

Gradually change the return type of collection min and max functions to non-nullables

Issue: [KT-38854](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: return type of collection min and max functions will be changed to non-nullables in 1.6

Deprecation cycle:

- 1.4: introduce ...`OrNull` functions as synonyms and deprecate the affected API (see details in the issue)
- 1.5.0: raise the deprecation level of the affected API to error
- >=1.6: reintroduce the affected API but with non-nullables return type

Raise the deprecation level of conversions of floating-point types to Short and Byte

Issue: [KT-30360](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: conversions of floating-point types to Short and Byte deprecated in Kotlin 1.4 with WARNING level will cause errors since Kotlin 1.5.0.

Deprecation cycle:

- 1.4: deprecate Double.toShort()/toByte() and Float.toShort()/toByte() and propose replacement
- 1.5.0: raise the deprecation level to error

Tools

Do not mix several JVM variants of kotlin-test in a single project

Issue: [KT-40225](#)

Component: Gradle

Incompatible change type: behavioral

Short summary: several mutually exclusive kotlin-test variants for different testing frameworks could have been in a project if one of them is brought by a transitive dependency. From 1.5.0, Gradle won't allow having mutually exclusive kotlin-test variants for different testing frameworks.

Deprecation cycle:

- < 1.5: having several mutually exclusive kotlin-test variants for different testing frameworks is allowed
- >= 1.5: behavior changed,

Gradle throws an exception like "Cannot select module with conflict on capability...". Possible solutions:

- use the same kotlin-test variant and the corresponding testing framework as the transitive dependency brings.
- find another variant of the dependency that doesn't bring the kotlin-test variant transitively, so you can use the testing framework you would like to use.
- find another variant of the dependency that brings another kotlin-test variant transitively, which uses the same testing framework you would like to use.
- exclude the testing framework that is brought transitively. The following example is for excluding JUnit 4:

```
configurations {
    testImplementation.get().exclude("org.jetbrains.kotlin", "kotlin-test-junit")
}
```

After excluding the testing framework, test your application. If it stopped working, rollback excluding changes, use the same testing framework as the library does, and exclude your testing framework.

Compatibility guide for Kotlin 1.4

[Keeping the Language Modern](#) and [Comfortable Updates](#) are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.3 to Kotlin 1.4.

Basic terms

In this document we introduce several kinds of compatibility:

- source: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- binary: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- behavioral: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language and stdlib

Unexpected behavior with in infix operator and ConcurrentHashMap

Issue: [KT-18053](#)

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.4 will outlaw auto operator contains coming from the implementors of java.util.Map written in Java

Deprecation cycle:

- < 1.4: introduce warning for problematic operators at call-site
- >= 1.4: raise this warning to an error, -XXLanguage:-ProhibitConcurrentHashMapContains can be used to temporarily revert to pre-1.4 behavior

Prohibit access to protected members inside public inline members

Issue: [KT-21178](#)

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.4 will prohibit access to protected members from public inline members.

Deprecation cycle:

- < 1.4: introduce warning at call-site for problematic cases
- 1.4: raise this warning to an error, -XXLanguage:-ProhibitProtectedCallFromInline can be used to temporarily revert to pre-1.4 behavior

Contracts on calls with implicit receivers

Issue: [KT-28672](#)

Component: Core Language

Incompatible change type: behavioral

Short summary: smart casts from contracts will be available on calls with implicit receivers in 1.4

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-ContractsOnCallsWithImplicitReceiver can be used to temporarily revert to pre-1.4 behavior

Inconsistent behavior of floating-point number comparisons

Issues: [KT-22723](#)

Component: Core language

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, Kotlin compiler will use IEEE 754 standard to compare floating-point numbers

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-ProperIEEE754Comparisons can be used to temporarily revert to pre-1.4 behavior

No smart cast on the last expression in a generic lambda

Issue: [KT-15020](#)

Component: Core Language

Incompatible change type: behavioral

Short summary: smart casts for last expressions in lambdas will be correctly applied since 1.4

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Do not depend on the order of lambda arguments to coerce result to Unit

Issue: [KT-36045](#)

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, lambda arguments will be resolved independently without implicit coercion to Unit

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Wrong common supertype between raw and integer literal type leads to unsound code

Issue: [KT-35681](#)

Components: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, common supertype between raw Comparable type and integer literal type will be more specific

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type safety problem because several equal type variables are instantiated with a different types

Issue: [KT-35679](#)

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, Kotlin compiler will prohibit instantiating equal type variables with different types

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type safety problem because of incorrect subtyping for intersection types

Issues: [KT-22474](#)

Component: Core language

Incompatible change type: source

Short summary: in Kotlin 1.4, subtyping for intersection types will be refined to work more correctly

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

No type mismatch with an empty when expression inside lambda

Issue: [KT-17995](#)

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, there will be a type mismatch for empty when expression if it's used as the last expression in a lambda

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Return type Any inferred for lambda with early return with integer literal in one of possible return values

Issue: [KT-20226](#)

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, integer type returning from a lambda will be more specific for cases when there is early return

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Proper capturing of star projections with recursive types

Issue: [KT-33012](#)

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, more candidates will become applicable because capturing for recursive types will work more correctly

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Common supertype calculation with non-proper type and flexible one leads to incorrect results

Issue: [KT-37054](#)

Component: Core language

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, common supertype between flexible types will be more specific protecting from runtime errors

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type safety problem because of lack of captured conversion against nullable type argument

Issue: [KT-35487](#)

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, subtyping between captured and nullable types will be more correct protecting from runtime errors

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Preserve intersection type for covariant types after unchecked cast

Issue: [KT-37280](#)

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, unchecked casts of covariant types produce the intersection type for smart casts, not the type of the unchecked cast.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type variable leaks from builder inference because of using this expression

Issue: [KT-32126](#)

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, using this inside builder functions like sequence {} is prohibited if there are no other proper constraints

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Wrong overload resolution for contravariant types with nullable type arguments

Issue: [KT-31670](#)

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, if two overloads of a function that takes contravariant type arguments differ only by the nullability of the type (such as In<T> and In<T?>), the nullable type is considered more specific.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Builder inference with non-nested recursive constraints

Issue: [KT-34975](#)

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, builder functions such as sequence {} with type that depends on a recursive constraint inside the passed lambda cause a compiler error.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Eager type variable fixation leads to a contradictory constraint system

Issue: [KT-25175](#)

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, the type inference in certain cases works less eagerly allowing to find the constraint system that is not contradictory.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-NewInference can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Prohibit tailrec modifier on open functions

Issue: [KT-18541](#)

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, functions can't have open and tailrec modifiers at the same time.

Deprecation cycle:

- < 1.4: report a warning on functions that have open and tailrec modifiers together (error in the progressive mode).
- >= 1.4: raise this warning to an error.

The INSTANCE field of a companion object more visible than the companion object class itself

Issue: [KT-11567](#)

Component: Kotlin/JVM

Incompatible change type: source

Short summary: since Kotlin 1.4, if a companion object is private, then its field INSTANCE will be also private

Deprecation cycle:

- < 1.4: the compiler generates object INSTANCE with a deprecated flag
- >= 1.4: companion object INSTANCE field has proper visibility

Outer finally block inserted before return is not excluded from the catch interval of the inner try block without finally

Issue: [KT-31923](#)

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, the catch interval will be computed properly for nested try/catch blocks

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-ProperFinally can be used to temporarily revert to pre-1.4 behavior

Use the boxed version of an inline class in return type position for covariant and generic-specialized overrides

Issues: [KT-30419](#)

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, functions using covariant and generic-specialized overrides will return boxed values of inline classes

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Do not declare checked exceptions in JVM bytecode when using delegation to Kotlin interfaces

Issue: [KT-35834](#)

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will not generate checked exceptions during interface delegation to Kotlin interfaces

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, -XXLanguage:-DoNotGenerateThrowsForDelegatedKotlinMembers can be used to temporarily revert to pre-1.4 behavior

Changed behavior of signature-polymorphic calls to methods with a single vararg parameter to avoid wrapping the argument into another array

Issue: [KT-35469](#)

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will not wrap the argument into another array on a signature-polymorphic call

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Incorrect generic signature in annotations when KClass is used as a generic parameter

Issue: [KT-35207](#)

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will fix incorrect type mapping in annotations when KClass is used as a generic parameter

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Forbid spread operator in signature-polymorphic calls

Issue: [KT-35226](#)

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will prohibit the use of spread operator (*) on signature-polymorphic calls

Deprecation cycle:

- < 1.4: report a warning on the use of a spread operator in signature-polymorphic calls
- >= 1.5: raise this warning to an error, -XXLanguage:-ProhibitSpreadOnSignaturePolymorphicCall can be used to temporarily revert to pre-1.4 behavior

Change initialization order of default values for tail-recursive optimized functions

Issue: [KT-31540](#)

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since Kotlin 1.4, the initialization order for tail-recursive functions will be the same as for regular functions

Deprecation cycle:

- < 1.4: report a warning at declaration-site for problematic functions
- >= 1.4: behavior changed, -XXLanguage:-ProperComputationOrderOfTailrecDefaultParameters can be used to temporarily revert to pre-1.4 behavior

Do not generate ConstantValue attribute for non-const vals

Issue: [KT-16615](#)

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Since Kotlin 1.4, the compiler will not generate the ConstantValue attribute for non- const val s

Deprecation cycle:

- < 1.4: report a warning through an IntelliJ IDEA inspection
- >= 1.4: behavior changed, -XXLanguage:-NoConstantValueAttributeForNonConstVals can be used to temporarily revert to pre-1.4 behavior

Generated overloads for @JvmOverloads on open methods should be final

Issue: [KT-33240](#)

Components: Kotlin/JVM

Incompatible change type: source

Short summary: overloads for functions with `@JvmOverloads` will be generated as final

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-GenerateJvmOverloadsAsFinal` can be used to temporarily revert to pre-1.4 behavior

Lambdas returning `kotlin.Result` now return boxed value instead of unboxed

Issue: [KT-39198](#)

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, lambdas returning values of `kotlin.Result` type will return boxed value instead of unboxed

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Unify exceptions from null checks

Issue: [KT-22275](#)

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Starting from Kotlin 1.4, all runtime null checks will throw a `java.lang.NullPointerException`

Deprecation cycle:

- < 1.4: runtime null checks throw different exceptions, such as `KotlinNullPointerException`, `IllegalStateException`, `IllegalArgumentException`, and `TypeCastException`
- >= 1.4: all runtime null checks throw a `java.lang.NullPointerException`. `-Xno-unified-null-checks` can be used to temporarily revert to pre-1.4 behavior

Comparing floating-point values in array/list operations contains, indexOf, lastIndexOf: IEEE 754 or total order

Issue: [KT-28753](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: behavioral

Short summary: the List implementation returned from `Double/FloatArray.asList()` will implement `contains`, `indexOf`, and `lastIndexOf`, so that they use total order equality

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Gradually change the return type of collection min and max functions to non-nullable

Issue: [KT-38854](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: return type of collection min and max functions will be changed to non-nullables in 1.6

Deprecation cycle:

- 1.4: introduce ...OrNull functions as synonyms and deprecate the affected API (see details in the issue)
- 1.5.x: raise the deprecation level of the affected API to error
- >=1.6: reintroduce the affected API but with non-nullables return type

Deprecate appendIn in favor of appendLine

Issue: [KT-38754](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: `StringBuilder.appendIn()` will be deprecated in favor of `StringBuilder.appendLine()`

Deprecation cycle:

- 1.4: introduce `appendLine` function as a replacement for `appendIn` and deprecate `appendIn`
- >=1.5: raise the deprecation level to error

Deprecate conversions of floating-point types to Short and Byte

Issue: [KT-30360](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: source

Short summary: since Kotlin 1.4, conversions of floating-point types to Short and Byte will be deprecated

Deprecation cycle:

- 1.4: deprecate Double.toShort()/toByte() and Float.toShort()/toByte() and propose replacement
- >=1.5: raise the deprecation level to error

Fail fast in Regex.findAll on an invalid startIndex

Issue: [KT-28356](#)

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: since Kotlin 1.4, findAll will be improved to check that startIndex is in the range of the valid position indices of the input char sequence at the moment of entering findAll, and throw IndexOutOfBoundsException if it's not

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Remove deprecated kotlin.coroutines.experimental

Issue: [KT-36083](#)

Component: kotlin-stdlib

Incompatible change type: source

Short summary: since Kotlin 1.4, the deprecated `kotlin.coroutines.experimental` API is removed from stdlib

Deprecation cycle:

- < 1.4: `kotlin.coroutines.experimental` is deprecated with the ERROR level
- >= 1.4: `kotlin.coroutines.experimental` is removed from stdlib. On the JVM, a separate compatibility artifact is provided (see details in the issue).

Remove deprecated mod operator

Issue: [KT-26654](#)

Component: kotlin-stdlib

Incompatible change type: source

Short summary: since Kotlin 1.4, mod operator on numeric types is removed from stdlib

Deprecation cycle:

- < 1.4: mod is deprecated with the ERROR level
- >= 1.4: mod is removed from stdlib

Hide `Throwable.addSuppressed` member and prefer extension instead

Issue: [KT-38777](#)

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: `Throwable.addSuppressed()` extension function is now preferred over the `Throwable.addSuppressed()` member function

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

capitalize should convert digraphs to title case

Issue: [KT-38817](#)

Component: kotlin-stdlib

Incompatible change type: behavioral

Short summary: `String.capitalize()` function now capitalizes digraphs from the [Serbo-Croatian Gaj's Latin alphabet](#) in the title case (Dž instead of DŽ)

Deprecation cycle:

- < 1.4: digraphs are capitalized in the upper case (DŽ)
- >= 1.4: digraphs are capitalized in the title case (Dž)

Tools

Compiler arguments with delimiter characters must be passed in double quotes on Windows

Issue: [KT-30211](#)

Component: CLI

Incompatible change type: behavioral

Short summary: on Windows, kotlinc.bat arguments that contain delimiter characters (whitespace, =, ;, ,) now require double quotes ("")

Deprecation cycle:

- < 1.4: all compiler arguments are passed without quotes
- >= 1.4: compiler arguments that contain delimiter characters (whitespace, =, ;, ,) require double quotes ("")

KAPT: Names of synthetic \$annotations() methods for properties have changed

Issue: [KT-36926](#)

Component: KAPT

Incompatible change type: behavioral

Short summary: names of synthetic \$annotations() methods generated by KAPT for properties have changed in 1.4

Deprecation cycle:

- < 1.4: names of synthetic \$annotations() methods for properties follow the template
`<propertyName>@annotations()`
- >= 1.4: names of synthetic \$annotations() methods for properties include the get prefix:
`get<PropertyName>@annotations()`

Compatibility guide for Kotlin 1.3

[Keeping the Language Modern and Comfortable Updates](#) are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.2 to Kotlin 1.3.

Basic terms

In this document we introduce several kinds of compatibility:

- Source: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- Binary: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- Behavioral: a change is said to be behavioral-incompatible if one and the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (e.g. from Java) is out of the scope of this document.

Incompatible changes

Evaluation order of constructor arguments regarding call

Issue: [KT-19532](#)

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: evaluation order with respect to class initialization is changed in 1.3

Deprecation cycle:

- <1.3: old behavior (see details in the Issue)
- >= 1.3: behavior changed, -Xnormalize-constructor-calls=disable can be used to temporarily revert to pre-1.3 behavior. Support for this flag is going to be removed in the next major release.

Missing getter-targeted annotations on annotation constructor parameters

Issue: [KT-25287](#)

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: getter-target annotations on annotations constructor parameters will be properly written to classfiles in 1.3

Deprecation cycle:

- <1.3: getter-target annotations on annotation constructor parameters are not applied
- >=1.3: getter-target annotations on annotation constructor parameters are properly applied and written to the generated code

Missing errors in class constructor's @get: annotations

Issue: [KT-19628](#)

Component: Core language

Incompatible change type: Source

Short summary: errors in getter-target annotations will be reported properly in 1.3

Deprecation cycle:

- <1.2: compilation errors in getter-target annotations were not reported, causing incorrect code to be compiled fine.
- 1.2.x: errors reported only by tooling, the compiler still compiles such code without any warnings
- >=1.3: errors reported by the compiler too, causing erroneous code to be rejected

Nullability assertions on access to Java types annotated with @NotNull

Issue: [KT-20830](#)

Component: Kotlin/JVM

Incompatible change type: Behavioral

Short summary: nullability assertions for Java-types annotated with not-null annotations will be generated more aggressively, causing code which passes null here to fail faster.

Deprecation cycle:

- <1.3: the compiler could miss such assertions when type inference was involved, allowing potential null propagation during compilation against binaries (see Issue for details).
- >=1.3: the compiler generates missed assertions. This can cause code which was (erroneously) passing null here fail faster.
-XXLanguage:-StrictJavaNullabilityAssertions can be used to temporarily return to the pre-1.3 behavior. Support for this flag will be removed in the next major release.

Unsound smartcasts on enum members

Issue: [KT-20772](#)

Component: Core language

Incompatible change type: Source

Short summary: a smartcast on a member of one enum entry will be correctly applied to only this enum entry

Deprecation cycle:

- <1.3: a smartcast on a member of one enum entry could lead to an unsound smartcast on the same member of other enum entries.
- >=1.3: smartcast will be properly applied only to the member of one enum entry.
-XXLanguage:-SoundSmartcastForEnumEntries will temporarily return old behavior. Support for this flag will be removed in the next major release.

val backing field reassignment in getter

Issue: [KT-16681](#)

Components: Core language

Incompatible change type: Source

Short summary: reassignment of the backing field of val -property in its getter is now prohibited

Deprecation cycle:

- <1.2: Kotlin compiler allowed to modify backing field of val in its getter. Not only it violates Kotlin semantic, but also generates ill-behaved JVM bytecode which reassigns final field.
- 1.2.X: deprecation warning is reported on code which reassigned backing field of val
- >=1.3: deprecation warnings are elevated to errors

Array capturing before the for-loop where it is iterated

Issue: [KT-21354](#)

Component: Kotlin/JVM

Incompatible change type: Source

Short summary: if an expression in for-loop range is a local variable updated in a loop body, this change affects loop execution. This is inconsistent with iterating over other containers, such as ranges, character sequences, and collections.

Deprecation cycle:

- <1.2: described code patterns are compiled fine, but updates to local variable affect loop execution
- 1.2.X: deprecation warning reported if a range expression in a for-loop is an array-typed local variable which is assigned in a loop body
- 1.3: change behavior in such cases to be consistent with other containers

Nested classifiers in enum entries

Issue: [KT-16310](#)

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, nested classifiers (classes, object, interfaces, annotation classes, enum classes) in enum entries are prohibited

Deprecation cycle:

- <1.2: nested classifiers in enum entries are compiled fine, but may fail with exception at runtime
- 1.2.X: deprecation warnings reported on the nested classifiers
- >=1.3: deprecation warnings elevated to errors

Data class overriding copy

Issue: [KT-19618](#)

Components: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, data classes are prohibited to override copy()

Deprecation cycle:

- <1.2: data classes overriding copy() are compiled fine but may fail at runtime/expose strange behavior
- 1.2.X: deprecation warnings reported on data classes overriding copy()
- >=1.3: deprecation warnings elevated to errors

Inner classes inheriting Throwable that capture generic parameters from the outer class

Issue: [KT-17981](#)

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, inner classes are not allowed to inherit Throwable

Deprecation cycle:

- <1.2: inner classes inheriting Throwable are compiled fine. If such inner classes happen to capture generic parameters, it could lead to strange code patterns which fail at runtime.
- 1.2.X: deprecation warnings reported on inner classes inheriting Throwable
- >=1.3: deprecation warnings elevated to errors

Visibility rules regarding complex class hierarchies with companion objects

Issues: [KT-21515](#), [KT-25333](#)

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, rules of visibility by short names are stricter for complex class hierarchies involving companion objects and nested classifiers.

Deprecation cycle:

- <1.2: old visibility rules (see Issue for details)
- 1.2.X: deprecation warnings reported on short names which are not going to be accessible anymore. Tooling suggests automated migration by adding full name.
- >=1.3: deprecation warnings elevated to errors. Offending code should add full qualifiers or explicit imports

Non-constant vararg annotation parameters

Issue: [KT-23153](#)

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, setting non-constant values as vararg annotation parameters is prohibited

Deprecation cycle:

- <1.2: the compiler allows to pass non-constant value for vararg annotation parameter, but actually drops that value during bytecode generation, leading to non-obvious behavior
- 1.2.X: deprecation warnings reported on such code patterns
- >=1.3: deprecation warnings elevated to errors

Local annotation classes

Issue: [KT-23277](#)

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 local annotation classes are not supported

Deprecation cycle:

- <1.2: the compiler compiled local annotation classes fine
- 1.2.X: deprecation warnings reported on local annotation classes
- >=1.3: deprecation warnings elevated to errors

Smartcasts on local delegated properties

Issue: [KT-22517](#)

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 smartcasts on local delegated properties are not allowed

Deprecation cycle:

- <1.2: the compiler allowed to smartcast local delegated property, which could lead to unsound smartcast in case of ill-behaved delegates
- 1.2.X: smartcasts on local delegated properties are reported as deprecated (the compiler issues warnings)
- >=1.3: deprecation warnings elevated to errors

mod operator convention

Issues: [KT-24197](#)

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 declaration of mod operator is prohibited, as well as calls which resolve to such declarations

Deprecation cycle:

- 1.1.X, 1.2.X: report warnings on declarations of operator mod, as well as on calls which resolve to it
- 1.3.X: elevate warnings to error, but still allow to resolve to operator mod declarations
- 1.4.X: do not resolve calls to operator mod anymore

Passing single element to vararg in named form

Issues: [KT-20588](#), [KT-20589](#). See also [KT-20171](#)

Component: Core language

Incompatible change type: Source

Short summary: in Kotlin 1.3, assigning single element to vararg is deprecated and should be replaced with consecutive spread and array construction.

Deprecation cycle:

- <1.2: assigning one value element to vararg in named form compiles fine and is treated as assigning single element to array, causing non-obvious behavior when assigning array to vararg
- 1.2.X: deprecation warnings are reported on such assignments, users are suggested to switch to consecutive spread and array construction.
- 1.3.X: warnings are elevated to errors
- >= 1.4: change semantic of assigning single element to vararg, making assignment of array equivalent to the assignment of a spread of an array

Retention of annotations with target EXPRESSION

Issue: [KT-13762](#)

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, only SOURCE retention is allowed for annotations with target EXPRESSION

Deprecation cycle:

- <1.2: annotations with target EXPRESSION and retention other than SOURCE are allowed, but silently ignored at use-sites
- 1.2.X: deprecation warnings are reported on declarations of such annotations
- >=1.3: warnings are elevated to errors

Annotations with target PARAMETER shouldn't be applicable to parameter's type

Issue: [KT-9580](#)

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, error about wrong annotation target will be properly reported when annotation with target PARAMETER is applied to parameter's type

Deprecation cycle:

- <1.2: aforementioned code patterns are compiled fine; annotations are silently ignored and not present in the bytecode
- 1.2.X: deprecation warnings are reported on such usages
- >=1.3: warnings are elevated to errors

Array.copyOfRange throws an exception when indices are out of bounds instead of enlarging the returned array

Issue: [KT-19489](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, ensure that the toIndex argument of Array.copyOfRange, which represents the exclusive end of the range being copied, is not greater than the array size and throw IllegalArgumentException if it is.

Deprecation cycle:

- <1.3: in case toIndex in the invocation of Array.copyOfRange is greater than the array size, the missing elements in range fill be filled with null s, violating soundness of the Kotlin type system.
- >=1.3: check that toIndex is in the array bounds, and throw exception if it isn't

Progressions of ints and longs with a step of Int.MIN_VALUE and Long.MIN_VALUE are outlawed and won't be allowed to be instantiated

Issue: [KT-17176](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, prohibit step value for integer progressions being the minimum negative value of its integer type (Long or Int), so that calling `IntProgression.fromClosedRange(0, 1, step = Int.MIN_VALUE)` will throw `IllegalArgumentException`

Deprecation cycle:

- <1.3: it was possible to create an `IntProgression` with `Int.MIN_VALUE` step, which yields two values [0, -2147483648], which is non-obvious behavior
- >=1.3: throw `IllegalArgumentException` if the step is the minimum negative value of its integer type

Check for index overflow in operations on very long sequences

Issue: [KT-16097](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, make sure index, count and similar methods do not overflow for long sequences. See the Issue for the full list of affected methods.

Deprecation cycle:

- <1.3: calling such methods on very long sequences could produce negative results due to integer overflow
- >=1.3: detect overflow in such methods and throw exception immediately

Unify split by an empty match regex result across the platforms

Issue: [KT-21049](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, unify behavior of split method by empty match regex across all platforms

Deprecation cycle:

- <1.3: behavior of described calls is different when comparing JS, JRE 6, JRE 7 versus JRE 8+
- >=1.3: unify behavior across the platforms

Discontinued deprecated artifacts in the compiler distribution

Issue: [KT-23799](#)

Component: other

Incompatible change type: Binary

Short summary: Kotlin 1.3 discontinues the following deprecated binary artifacts:

- kotlin-runtime: use kotlin-stdlib instead
- kotlin-stdlib-jre7/8: use kotlin-stdlib-jdk7/8 instead
- kotlin-stdlib in the compiler distribution: use kotlin-stdlib-js instead

Deprecation cycle:

- 1.2.X: the artifacts were marked as deprecated, the compiler reported warning on usage of those artifacts
- >=1.3: the artifacts are discontinued

Annotations in stdlib

Issue: [KT-21784](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: Binary

Short summary: Kotlin 1.3 removes annotations from the package org.jetbrains.annotations from stdlib and moves them to the separate artifacts shipped with the compiler: annotations-13.0.jar and mutability-annotations-compat.jar

Deprecation cycle:

- <1.3: annotations were shipped with the stdlib artifact
- >=1.3: annotations ship in separate artifacts

Compatibility modes

When a big team is migrating onto a new version, it may appear in a "inconsistent state" at some point, when some developers have already updated, and others haven't. To prevent the former from writing and committing code that others may not be able to compile, we provide the following command line switches (also available in the IDE and [Gradle/ Maven](#)):

- -language-version X.Y- compatibility mode for Kotlin language version X.Y, reports errors for all language features that came out later.
- -api-version X.Y- compatibility mode for Kotlin API version X.Y, reports errors for all code using newer APIs from the Kotlin Standard Library (including the code generated by the compiler).

Below, we use OV for "Older Version", and NV for "Newer Version".

Binary compatibility warnings

If you use the NV Kotlin compiler and have the OV standard library or the OV reflection library in the classpath, it can be a sign that the project is misconfigured. To prevent unexpected problems during compilation or at runtime, we suggest either updating the dependencies to NV, or specifying the API version / language version arguments explicitly. Otherwise, the compiler detects that something can go wrong and reports a warning.

For example, if OV = 1.0 and NV = 1.1, you can observe one of the following warnings:

- Runtime JAR files in the classpath have the version 1.0, which is older than the API version 1.1. Consider using the runtime of version 1.1, or pass '-api-version 1.0' explicitly to restrict the available APIs to the runtime of version 1.0. This means that you're using the Kotlin compiler 1.1 against the standard or reflection library of version 1.0. This can be handled in different ways:

- If you intend to use the APIs from the 1.1 Standard Library, or language features that depend on those APIs, you should upgrade the dependency to the version 1.1.
- If you want to keep your code compatible with the 1.0 standard library, you can pass -api-version 1.0.
- If you've just upgraded to Kotlin 1.1 but can not use new language features yet (e.g. because some of your teammates may not have upgraded), you can pass -language-version 1.0, which will restrict all APIs and language features to 1.0.
- Runtime JAR files in the classpath should have the same version. These files were found in the classpath: kotlin-reflect.jar (version 1.0) kotlin-stdlib.jar (version 1.1) Consider providing an explicit dependency on kotlin-reflect 1.1 to prevent strange errors Some runtime JAR files in the classpath have an incompatible version. Consider removing them from the classpath
This means that you have a dependency on libraries of different versions, for example the 1.1 standard library and the 1.0 reflection library. To prevent subtle errors at runtime, we recommend you to use the same version of all Kotlin libraries. In this case, consider adding an explicit dependency on the 1.1 reflection library.
- Some JAR files in the classpath have the Kotlin Runtime library bundled into them. This may cause difficult to debug problems if there's a different version of the Kotlin Runtime library in the classpath. Consider removing these libraries from the classpath
This means that there's a library in the classpath which does not depend on the Kotlin standard library as a Gradle/Maven dependency, but is distributed in the same artifact with it (i.e. has it bundled). Such a library may cause issues because standard build tools do not consider it an instance of the Kotlin standard library, thus it's not subject to the dependency version resolution mechanisms, and you can end up with several versions of the same library in the classpath. Consider contacting the authors of such a library and suggesting to use the Gradle/Maven dependency instead.

Kotlin Foundation

The Kotlin Foundation was created by JetBrains and Google with the mission to protect, promote and advance the development of the Kotlin programming language. The Foundation secures Kotlin's development and distribution as Free Software, meaning that it is able to be freely copied, modified and redistributed, including modifications to the official versions.

Scope

The main functions of the Kotlin Foundation are:

- Holding the trademarks associated with the project
- Appointing the Lead Language Designer
- Controlling incompatible changes to the language

Development. The code for the project is developed openly on GitHub primarily by the team employed at JetBrains, with contributions from Google and others. The Foundation has no developers on its payroll. The respective contributors hold the copyright for the code they author. The code is distributed under the [Apache License, Version 2.0](#).

Trademarks. The Kotlin trademark and associated marks, logos and domain names belong to the Kotlin Foundation. The Foundation permits the usage of the trademarks according to [the Kotlin Brand Usage Guidelines](#). All usages not permitted automatically, require explicit written permission from the Foundation.

Lead Language Designer is in charge of all decisions regarding the development of the project and evolution of the language. The Foundation appoints the Lead Language Designer, but does not directly make decisions about the course of the project.

Incompatible changes. A special Language Committee appointed by the Foundation reviews the changes proposed by the Lead Language Designer and has the power to reject [incompatible changes](#) or require certain deprecation procedures to be carried out. The Language Committee has no power over compatible changes such as adding new features to the language. Compatible changes are made at the sole discretion of the Lead Language Designer. The Committee maintains a set of [guidelines](#) that are being consulted when making decisions. Meeting minutes of the Language Committee are available [online](#).

Structure

The Kotlin Foundation is a nonprofit nonstock corporation registered in the state of Delaware, USA. The Foundation is governed by the Board of Directors that makes all decisions by voting. The Board appoints the Lead Language Designer and the members of the Language Committee.

The President and the Secretary are elected by the Board of Directors as well. The President acts as the chief officer of the Foundation and supervises its affairs and operations. The Secretary keeps record of all meetings of the Board and its committees, and keeps track of membership and notices to the members, directors and third parties.

Current personnel

President: Maxim Shafirov (JetBrains)

Secretary: David Winer (Google)

Board of Directors:

- Maxim Shafirov (JetBrains)
- Roman Elizarov (JetBrains)
- William R. Cook (University of Texas at Austin)
- Grace Kloba (Google)

- Anwar Ghuloum (Google)

Lead Language Designer: Roman Elizarov (JetBrains)

Language Committee:

- Roman Elizarov (JetBrains)
- William R. Cook (University of Texas at Austin)
- Jeffrey van Gogh (Google)

Language Committee guidelines

The Kotlin Language Committee has to approve every incompatible change before it lands in a stable version. The committee determines whether the change is appropriate and motivated enough to be introduced and what deprecation procedures have to be carried out to ensure comfortable migration for end users.

These guidelines describe the process the Committee follows and the criteria used for decision making. We realize that these criteria can not be fully formalized and treat them as general guidelines, not a mechanical set of rules.

Review process

Incompatible changes are submitted to the Language Committee by the Lead Language Designer. The Committee evaluates the implications of such changes, considering the following:

- Substance: what is going to be changed
- Impact: who will be affected by the change and in what ways
- Motivation: why the change is needed
- Detectability: can problematic code be automatically detected
- Migration: to what extent can the code be migrated automatically

Estimating impact

When in doubt, we inspect large bodies of code available to us and test the implementations in EAP builds to collect information about the expected impact of a change.

Proposals

A proposal is usually written by a developer responsible for the change or the relevant subsystems. See the Guide to submitting incompatible changes.

When an issue has all the necessary details, it can be scheduled for review by the Language Committee. Approved proposals are to be implemented by the development team and shipped in the appropriate stable release.

Scope

Only features and components published as [Stable](#) are in the scope of Language Committee. Besides, the Language Committee's scope is limited to the following:

- Language: syntax, static checks, execution semantics of language constructs
- The interop subsystem of the language: how foreign declarations are seen from Kotlin, and how Kotlin declarations are seen from other languages
- Compatibility of binary artifacts produced by kotlinc with one another and with other binaries (e.g. Java binaries)
- Standard library: API and contracts of the declarations in kotlin-stdlib (and its extensions such as for kotlin-stdlib-jdk7)
- CLI parameters of the compiler except for the -X/-XX keys
- KDoc syntax

For example, the following matters are out of scope for the Language Committee:

- Pre-stable language features & APIs
- Build tools and plugins for them (e.g. Gradle support)
- IDE and static analysis tools (other than the compiler)
- Java2Kotlin converter and other source code manipulation tools
- APIs and contracts of libraries other than the standard library
- API of the compiler
- Scripting support and Compiler REPL
- Internal packages of the standard library

Also, see [Appendix A](#) for examples of changes that are not considered to be compatibility issues.

Migration aids and deprecation

Normally, incompatible changes to stable features only land in feature releases. We announce the changes in a previous release (it may be an incremental update to the previous feature release, but should allow enough time for users to migrate their code).

The announcement is best done through compiler warnings, or in some cases IDE inspections and/or other tools. If places in the code that should be changed in advance to prevent compatibility issues in the future can be detected automatically, we announce the change and publish detection tools.

If the necessary changes to user code can be automated, we publish a migration tool along with the announcement. Preferably, the IDE should suggest running the migration when it encounters deprecated constructs in the code.

When automatic detection and/or migration is not practically possible, we publish instructions on how user code should be adjusted.

Insignificant changes

Some types of changes can be made without a deprecation cycle:

- Small fixes that virtually no users will encounter can normally be made right away (but still require committee review)
- Bugs causing compiler errors in good code can be fixed right away
- Bugs causing bad code to compile, but always fail at runtime can be fixed right away

Incompatible changes to the language

The typical deprecation cycle for an incompatible language change:

- The change planned for version B is announced in release notes to a previous version A
 - The compiler reports deprecation warnings (messages provide some guidance to possible replacements)
 - Automated migration tools are available in the IDE
 - For eligible changes, the -progressive flag enables the change before version B (e.g. in version A or an incremental update to it)
- Version B ships with the change enabled (e.g. warnings turn into errors)
 - Migrations aids are kept available in the IDE
 - Backward compatibility flags in the compiler (-language-version and -api-version) support the behavior of version A
- Future versions can remove migration aids and compatibility flags, but a version capable of automated migration should be kept available for a reasonable amount of time

Incompatible changes to the libraries

The typical deprecation cycle for libraries:

- The change planned for version B is announced in release notes to a previous version A
 - Relevant declarations are marked as @Deprecated(level = WARNING)
 - Migration is automated through ReplaceWith or sometimes custom tooling

- In some cases an optional support dependency that exposes the same API is published
- Version B marks the declarations as `@Deprecated(level = ERROR)`
- Version C can mark the declarations as `@Deprecated(level = HIDDEN)`
 - Note: for inline functions, complete removal is sometimes possible at this point

Changes to these guidelines

Changes to these guidelines need to be approved by the Kotlin Language Committee.

Any proposed change needs to be published in advance providing a reasonable time to allow for comments on the change by the Kotlin Community.

Appendix A. Examples of non-issues

The following cases are not considered to be compatibility issues and thus are out of scope for the Language Committee.

Changes related to binaries and translation

- A binary compiled against `kotlin-stdlib` fails to link or run because `kotlin-stdlib` is absent or outdated (older version than used during compilation).
- Adding generic parameters to existing declarations does not change the ABI on the JVM (due to erasure).
- Changes to signatures of functions marked `@InlineOnly` are not changing the ABI on the JVM.
- Adding supertypes to existing library classes/interfaces.

Source language changes

- Code compilable with a newer version fails to compile with an older version (e.g. due to usages of new language features).
- The code breaks only if the user alters the build configuration or compiler settings explicitly (i.e. in addition to advancing the compiler version).

Library changes

- Relaxing a contract on existing APIs.
- Clarification for unspecified behaviors.
- Changes in `hashCode()` are not breaking changes.

- Changes in `toString()` on other than Boolean, Numeric, and String types are not breaking changes.
- Issues connected with loading of two different versions of the same library at runtime or link time.

Changes affecting performance

We recognize that runtime performance and bytecode size are important metrics, and will make reasonable effort to keep them in a good shape, but we don't consider every slowdown (e.g. in edge cases or in very cold code) and every extra byte in the classfile a breaking change.

Appendix B. Assumptions for pragmatic language evolution

We make decisions on language changes under the assumption that most user code is designed with the following considerations in mind. We do not optimize for smooth migration that disregards these considerations.

Availability of the standard library

All Kotlin code should be linked and run against the Standard Library of at least the same version as specified by the `-api-version` upon compilation.

Type inference and overload resolution algorithms

Improvements in type inference algorithms may result in more precise static types known for some expressions, this may cause changes in overload resolution and even in signatures of declarations that don't specify return types explicitly.

Overloads of the same function should be intended to do the same thing. Language improvements may cause a different overload to be selected in a new language version.

Declarations sensitive to API/ABI stability (e.g. public APIs) should specify return types explicitly. Language improvements may cause a different type to be inferred for a given body expression. Also, some innocuous-looking changes in the source code, done by the user, may cause similar effects.

Non-public API

Internal declarations have no separate compilation guarantees. While sometimes accessible from other languages (e.g. Java), declarations marked internal in Kotlin should not be called from outside the module they are declared in.

Private and synthetic declarations should not be relied upon. On some platforms, reflection has access to declarations marked private and synthetic. We do not provide any compatibility guarantees for code that relies on such access.

Impedance mismatches across language boundaries

A Kotlin API exposed to another language may yield values that work in that language differently from Kotlin. For example, Kotlin's unsigned integers will look signed for Java clients, and the programmer that works with the same API

in the Java code will be surprised by getting different result. While an undesirable situation, this is sometimes inevitable, and should not be considered a breaking change (it does not fall under the intuitive definition of one anyway).

Guide to submitting incompatible changes

A proposal is usually written by a developer responsible for the change or the relevant subsystems. It has to be described in the [issue tracker](#) and marked with the "for-language-committee" tag. To provide all the necessary details, issue in the tracker should follow the template below.

Issue template

The issue should follow this template:

```
# Classification
Type of change: <see guide below>
Motivation types: <see guide below>
Impact types: <see guide below>
Detection and Migration modes: <see guide below>

# Background
> Optional section. Describe the background of the issue and possibly some motivation
# Existing behavior
> 1. (Minimal) code example
> 2. What's wrong with it
> 3. Detailed explanation of why and how this happens
> 4. Any known reasoning behind such behavior
> 5. Affected versions: when did the problem appear first?

# Proposed changes
> 1. Proposed changes
> 2. How it fixes the issue?

# Impact
> Examples of negative effects: changes of behavior, previously compiled code forbidden, etc.
> If possible:
> * number of impacted users
> * how common this pattern is
> * if we believe it's a rare case, why
> * is automated migration possible

# Migration plan
> Provide a detailed step-by-step migration plan
> Can the proposed changes be implemented immediately in the progressive mode?
```

Classification

The Classification section needs to classify the change, without describing its essence. The classification is used by the Committee to quickly assess the severity of the issue. Here are the common categories to be used.

This section can be used as a checklist of things that need to be evaluated/thought of.

Type of change

Specify one or more of the following:

- New errors are introduced
- Some valid language constructs change their meaning
- Change in the standard library
 - API removal
 - Affecting ABI
 - Not affecting ABI
 - API moved to different artifact
 - API type signature change
 - Affecting ABI
 - Not affecting ABI
- Contract refinement
 - Other contract changes
- Change in compilation strategy
 - Change in ABI
 - Change in the behavior of generated code
- Change in interoperability layer(s)

Motivation types

Specify one or more of the following:

- User code fails with exception(s)
- Compiler/Tooling fails with exception(s)
- The implementation does not abide by a published spec or documentation
- Type safety guarantees are not met (including fail-fast behavior for non-null types)
- Separate/incremental compilation guarantees are not met

- API stability guarantees are not met
- ABI stability guarantees are not met
- Implementation changes are required for implementation design/architectural reasons
- Problematic/meaningless usage patterns need to be discouraged/blocked (e.g. counterintuitive behaviors)
 - Code is error-prone
 - Code can't be compiled correctly
- Some prospective language changes are blocked
- Inconsistency in the design (things are done differently in different contexts)
- Redundant/obsolete mechanism

Impact types

- Compilation. Some code that used to compile won't compile any more
 - Such code always caused the compiler to fail with exception
 - Such code always threw an exception immediately upon execution or could not link
 - Such code inevitably caused exceptions to be thrown somewhere down the line after it was executed
 - There were cases when such code worked with no exceptions
 - There was a warning reported on all such code
 - Some such code could compile without any warnings
- Binaries. Some binary metadata will change after recompilation
 - Internal (and maybe private) signatures
 - None of them could have been called from other languages
 - Some of them may have been callable from other languages (Java/JS/C/Swift/...)
 - Public and/or protected signatures may change
- Behavior changes
 - Exceptions
 - Some exceptions may change their type or place, but all code that ran without exceptions keeps doing so
 - Some exceptions can be thrown earlier than before
 - Some exceptions can be thrown later than before
 - Some exceptions change their type, but not the point of execution at which they are thrown
 - The new type is a subtype of the old one
 - The new type is not a subtype of the old one

- Some code that compiled and ran without exceptions will throw exceptions in the new version
- The behavior in question belongs under a previously released specification, contract or official documentation
 - By the contract, the old behavior is incorrect (thus a fix is needed)
 - The contract doesn't specify this behavior
 - The new behavior needs to be specified
 - The new behavior will remain unspecified
 - The new behavior contradicts the contract (i.e. the contract has to be changed)
- Performance and code size
 - Some code may run slower (including "small" changes like extra checks and indirections)
 - Some code may make more allocations or otherwise consume more memory
 - Code size may increase
 - The increase in code size is constant regardless of the source
 - The increase in code size is proportional to some parameters of the source

Detection and migration modes

- All code locations affected by this change can be accurately detected statically
- Some, but not all code locations affected by this change can be accurately detected statically
- No code locations affected by this change can be accurately detected statically
- Migration can be fully automated
- Some automated migration is possible
- Migration can't be automated

Kotlin brand usage guidelines

One of the goals of The Kotlin Foundation is to protect the use of the Kotlin word and to make sure it is not misused. The following guidelines provide guidance on how to reference the Kotlin word trademark ("Trademark") in an approved manner while ensuring the Trademark remains subject to protection under applicable laws around the world, and continues to serve as a source indicator for the Kotlin programming language and related offerings.

Any use of the Trademark other than those described in these Guidelines must be approved in advance. Please contact the Kotlin Foundation to request an express trademark license if you desire to use the Kotlin trademark, or a derivative of the Kotlin trademark, in a manner not permitted under these Guidelines. To submit a usage request, please [contact us](#), clearly describing the intended usage and duration.

General goals

We want the word Kotlin to refer to the Kotlin Programming Language, and we want to avoid it being used to refer to any other programming language, or used in any way that may imply that the language is not free nor open source.

Permitted uses

Given the open nature of Kotlin, you may use the Trademark to refer to the Kotlin programming language. Examples of these references include the following:

- Identify that your software is written in the Kotlin programming language
- Is compatible with the Kotlin programming language
- Targets the Kotlin programming language
- Is for use with the Kotlin programming language
- Contains the Kotlin programming language.
- Aims to promote or teach the Kotlin programming language

The use must be referential and describe the relationship of your products or services to the Kotlin programming language.

Unpermitted uses

The Trademark may never be used in a manner that would cause confusion as to JetBrains, Google, or the Kotlin Foundation's sponsorship, affiliation, or endorsement, including as part of a company name, product name, domain name, or business trading name. Other than as allowed herein, the Kotlin word mark should not be combined with other trademarks, terms, or source identifiers.

Under no circumstances can you claim any trademark rights in the Kotlin name or attempt to register a name or source identifier that incorporates the Kotlin trademark or logo, or anything confusingly thereto, as a trademark, domain name, business name or other source identifier.

Usage guidelines

Please follow these guidelines when referencing the Trademark:

- The first or most prominent mention of the Kotlin trademark should be immediately followed by the TM symbol. For example "Kotlin™ ..." This requirement is waived in all contexts where such marks are not normally included, such as non-marketing emails, online discussions, and academic papers. We encourage the use of the TM symbol whenever possible, but recognize that many non-commercial and informal uses will omit it.

- Where identifying that a product or service is built on the Kotlin programming language or runs the Kotlin programming language, use the product's own name followed by "in Kotlin™", "for Kotlin™", "compatible with Kotlin™", "running Kotlin™" etc. Do not incorporate Kotlin into the product name.
- User groups, communities and/or meet-ups should follow the naming convention " Kotlin User Group" where represents the location and optionally other wordings. These entities should also prominently disclaim any affiliation with or endorsement by The Kotlin Foundation.
- Do not display the word Kotlin using any different stylization, color, or font from the surrounding text.
- Don't use the trademark as verb ("Kotlin your code"), or create derivatives of it such as "Kotlinize", etc.
- Do not use Kotlin in possessive form (i.e. Kotlin's).
- Do not use alternative spellings (e.g., "Ktln").
- In certain usage context, and at the discretion of the Trademark Subcommittee of The Kotlin Foundation, you may need to add a disclaimer on your site, publication, book, product, etc. where it clearly indicates that the product and/or service provided is not associated with The Kotlin Foundation.

Example usages

User groups and Kotlin meetups A valid name for a user group could be "San Diego Kotlin User Group" If you're a user group or Kotlin meetup, please use the [User Group assets provided](#).

Events If you're organising a conference or educational event, you can use the word Kotlin to refer to the technology that the event covers. However, Kotlin cannot be contained in the brand name for your conference or educational event (such as KotlinConf or Kotlin Conference). Instead, you should use your own brand name, but indicate that it is a conference for "Kotlin Language Users." Your conference or educational event title also should not cause any confusion or suggest any affiliation with or sponsorship by the Kotlin Foundation. Example of appropriate usages could be "XYZ Conference for Kotlin" with XYZ being your own brand name.

Other commercial or non-commercial names Whether you're referring to a product, company or service, you shouldn't incorporate Kotlin as your brand name, i.e. your company cannot be called "Kotlin Consulting" or "Kotlin IDE". It can however indicate that you provide consulting for the Kotlin programming language as part of the description. For example, XYZ Consulting, which focuses on the Kotlin programming language.

Kotlin Foundation FAQ

How can I contribute or provide feedback to Kotlin?

Kotlin is an open source project, and there are many ways you can contribute:

- Language features. Please use the [Kotlin Evolution and Enhancement process \(KEEP\)](#) if you'd like to make a

proposal for the Kotlin programming language.

- Code contributions. Please refer to [Contributions section](#) at the Kotlin GitHub repository.
- Filing issues or suggestions. Please use the [Kotlin Issue Tracker](#) to file issues or suggestions.

How can I get more involved with the Kotlin community?

There are many online and in-person Kotlin communities. Please visit the [Kotlin community page](#) to see a list of user groups, Kotlin nights and other ways that you can get involved!

How is the Kotlin Foundation governed?

The Foundation is governed by the Board of Directors, who then appoint a president, Lead Language Designer and members of the Language Committee. JetBrains and Google each delegate two directors to the board. The independent member is elected by the rest of the board.

Do JetBrains and Google contribute financially to the foundation?

Yes, to cover the operational and administrative expenses.

Does the foundation generate any revenue?

No, Kotlin Foundation is a non-profit entity whose operational costs are covered by JetBrains and Google.

Can I license the Kotlin trademark?

Yes. You are free to use the Kotlin trademark in accordance with our [Kotlin brand usage guidelines](#). Any use of the Trademark other than those described in the Guidelines must be approved in advance. If you need a specific license, please [refer to the request form](#).

How can a company or an individual join the foundation?

We are discussing possible ways for companies and individuals to participate, and will provide updates here in the future.

What resources should I follow to stay up-to-date on Kotlin?

The best way to stay up-to-date on Kotlin is to visit the official [Kotlin website](#), and follow the official [Kotlin blog](#).

Security

We do our best to make sure our products are free of security vulnerabilities. To reduce the risk of introducing a vulnerability, you can follow these best practices:

- Always use the latest Kotlin release. For security purposes, we sign our releases published on [Maven Central](#) with these PGP keys:
 - Key ID: kt-a@jetbrains.com
 - Fingerprint: 2FBA 29D0 8D2E 25EE 84C1 32C3 0729 A0AF F899 9A87
 - Key size: RSA 3072
- Use the latest versions of your application's dependencies. If you need to use a specific version of a dependency, periodically check if any new security vulnerabilities have been discovered. You can follow [the guidelines from GitHub](#) or browse known vulnerabilities in the [CVE base](#).

We are very eager and grateful to hear about any security issues you find. To report vulnerabilities that you discover in Kotlin, please post a message directly to our [issue tracker](#) or send us an [email](#).

For more information on how our responsible disclosure process works, please check the [JetBrains Coordinated Disclosure Policy](#).

Kotlin documentation as PDF

Here you can download a PDF version of Kotlin documentation that includes everything except tutorials and API reference.

Kotlin documentation is constantly being updated. The Kotlin PDF includes Kotlin 1.4.30 documentation updates but doesn't include later changes.

We recommend using the latest version of Kotlin documentation available [online](#).

[Download Kotlin 1.4.30 documentation \(PDF\)](#)

[View the latest Kotlin documentation \(online\)](#)