Kotlin Language Documentation

Table of Contents

Overview

Using Kotlin for Server-side Development

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.

— 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

— The Creating Web Applications with Http Servlets and Creating a RESTful Web Service with Spring Boot tutorials show you how you can build and run very small Web applications in Kotlin.

— For a more in-depth introduction to the language, check out the reference documentation on this site and Kotlin Koans.

Using Kotlin for Android Development

Kotlin is a great fit for developing Android applications, bringing all of the advantages of a modern language to the Android platform without introducing any new restrictions:

— **Compatibility**: Kotlin is fully compatible with JDK 6, ensuring that Kotlin applications can run on older Android devices with no issues. The Kotlin tooling is fully supported in Android Studio and compatible with the Android build system.

— **Performance**: A Kotlin application runs as fast as an equivalent Java one, thanks to very similar bytecode structure. With Kotlin's support for inline functions, code using lambdas often runs even faster than the same code written in Java.

— **Interoperability**: Kotlin is 100% interoperable with Java, allowing to use all existing Android libraries in a Kotlin application. This includes annotation processing, so databinding and Dagger work too.

— **Footprint**: Kotlin has a very compact runtime library, which can be further reduced through the use of ProGuard. In a [real application](#), the Kotlin runtime adds only a few hundred methods and less than 100K to the size of the .apk file.

— **Compilation Time**: Kotlin supports efficient incremental compilation, so while there's some additional overhead for clean builds, [incremental builds are usually as fast or faster than with Java](#).

— **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.

Kotlin for Android Case Studies

Kotlin has been successfully adopted by major companies, and a few of them have shared their experiences:

— Pinterest has successfully [introduced Kotlin into their application](#), used by 150M people every month.

— Basecamp's Android app is [100% Kotlin code](#), and they report a huge difference in programmer happiness and great improvements in work quality and speed.

— Keepsafe's App Lock app has also been [converted to 100% Kotlin](#), leading to a 30% decrease in source line count and 10% decrease in method count.

Kotlin Android Extensions

[Kotlin Android Extensions](#) is a compiler extension that allows you to get rid of `findViewById()` calls in your code and to replace them with synthetic compiler-generated properties.

Next Steps

— Download and install [Android Studio](#) which includes Kotlin support out of the box.

— Follow the [Getting Started with Android and Kotlin](#) tutorial to create your first Kotlin application.

— For a more in-depth introduction, check out the [reference documentation](#) on this site and [Kotlin Koans](#).

— Another great resource is [Kotlin for Android Developers](#), a book that guides you step by step through the process of creating a real Android application in Kotlin.

— Check out Google's [sample projects written in Kotlin](#).

Kotlin JavaScript Overview

Kotlin provides the ability to target JavaScript. It does so by transpiling Kotlin to JavaScript. The current implementation targets ECMAScript 5.1 but there are plans to eventually target ECMAScript 2015 as well.

When you choose the JavaScript target, any Kotlin code that is part of the project as well as the standard library that ships with Kotlin is transpiled to JavaScript. However, this excludes the JDK and any JVM or Java framework or library used. Any file that is not Kotlin will be ignored during compilation.

The Kotlin compiler tries to comply with the following goals:

— Provide output that is optimal in size
— Provide output that is readable JavaScript
— Provide interoperability with existing module systems
— Provide the same functionality in the standard library whether targeting JavaScript or the JVM (to the largest possible degree).

How it can be used

You may want to compile Kotlin to JavaScript in the following scenarios:

— Creating Kotlin code that targets client-side JavaScript

  — **Interacting with DOM elements**. Kotlin provides a series of statically typed interfaces to interact with the Document Object Model, allowing creation and update of DOM elements.

  — **Interacting with graphics such as WebGL**. You can use Kotlin to create graphical elements on a web page using WebGL.

— Creating Kotlin code that targets server-side JavaScript

  — **Working with server-side technology**. You can use Kotlin to interact with server-side JavaScript such as Node.js

Kotlin can be used together with existing third-party libraries and frameworks, such as jQuery or React. To access third-party frameworks with a strongly-typed API, you can convert TypeScript definitions from the Definitely Typed type definitions repository to Kotlin using the dukat tool. Alternatively, you can use the dynamic type to access any framework without strong typing.

JetBrains develops and maintains several tools specifically for the React community: React bindings as well as Create React Kotlin App. The latter helps you start building React apps with Kotlin with no build configuration.

Kotlin is compatible with CommonJS, AMD and UMD, making interaction with different module systems straightforward.

Getting Started with Kotlin to JavaScript

To find out how to start using Kotlin for JavaScript, please refer to the [tutorials](#).

Kotlin/Native for 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:

— iOS (arm32, arm64, simulator x86_64)

— MacOS (x86_64)

— Android (arm32, arm64)

— Windows (mingw x86_64, x86)

— Linux (x86_64, arm32, MIPS, MIPS little endian, Raspberry Pi)

— WebAssembly (wasm32)

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 are supported between different Kotlin and Kotlin/Native targets. This is the way to share common Kotlin code between many platforms, including Android, iOS, server-side, JVM, client-side, JavaScript, CSS, and native.

Multiplatform libraries provide the necessary APIs for the common Kotlin code and help to develop shared parts of a project in Kotlin code once and share it with all of the target platforms.

How to Start

Tutorials and Documentation

New to Kotlin? Take a look at the Getting Started page.

Suggested documentation pages:

— C interop
— Swift/Objective-C interop

Recommended tutorials:

— A basic Kotlin/Native application
— Multiplatform Project: iOS and Android
— Types mapping between C and Kotlin/Native
— Kotlin/Native as a Dynamic Library
— Kotlin/Native as an Apple Framework

Example Projects

— Kotlin/Native sources and examples
— KotlinConf app
— KotlinConf Spinner app
— Kotlin/Native sources and examples (.tgz)
— Kotlin/Native sources and examples (.zip)

Even more examples are on GitHub.

Kotlin for Data Science

Data science has a special place in information technology: it includes aspects of both software development and scientific research at the same time. As a discipline, data science covers a broad range of areas: data engineering, data analysis, machine learning, visualization, and many more.

To cover all these different areas, the software industry has many technologies and tools for data science. These include frameworks, specific IDEs (called *notebooks*), plotting tools, and programming languages designed specifically for data analysis and mathematical research.

General-purpose languages can also be applied to data science, Kotlin is already being adopted to data science. Here we'll introduce you to some things that are useful to know about when using Kotlin for data science.

Tools

Modern software developers very rarely write their code in plain text editors and run it from the command line anymore. Instead, they tend to use integrated development environments (IDE) that can handle all the development tasks in a single tool. Data scientists have similar tools called *notebooks*. Notebooks let users conduct research and store it in a single environment. In a notebook, you can write narrative text next to a code block, execute the code block, and see the results in any format that you need: output text, tables, data visualization, and so on.

Kotlin provides integration with two popular notebooks: Jupyter and Apache Zeppelin, which both allow you to write and run Kotlin code blocks.

Jupyter kernel

The open-source project [Jupyter](#) offers the well-known web-based development environment **Jupyter Notebook**. For code execution, Jupyter uses the concept of *kernels* - components that run separately and execute the code upon request, for example, when you click **Run** in a notebook.

There is a kernel that the Jupyter team maintains themselves - IPython for running the Python code. However, there are other community-maintained kernels for different languages. Among them is the **Kotlin kernel for Jupyter notebooks**. With this kernel, you can write and run Kotlin code in Jupyter notebooks and use third-party data science frameworks written in Java and Kotlin.

Setting up the Kotlin kernel

The Kotlin kernel requires Java 8 to be installed.

To install the kernel, use [Conda](#):

```
conda install kotlin-jupyter-kernel -c jetbrains
```

Once the kernel is installed, feel free to run the Jupyter notebook and switch to the Kotlin kernel. And that's all there is to it, you can then write and run Kotlin in your notebook!

You can find more information about the Kotlin kernel for Jupyter [here](#).

Zeppelin interpreter

[Apache Zeppelin](#) is a popular web-based solution for interactive data analytics. Zeppelin provides strong support for the [Apache Spark](#) cluster computing system, which is particularly useful for data engineering. Spark provides a high-level API in multiple languages.

The language support in Zeppelin is provided by *interpreters* - plugins that enable users to use a specific language or data-processing-backend. There are numerous community-maintained interpreters for different programming languages. The one we offer you is the **Kotlin interpreter for Apache Zeppelin** that adds the Kotlin support.

Setting up Zeppelin with Kotlin interpreter

Currently, the latest release of Zeppelin (0.8.2) doesn't come with bundled Kotlin interpreter. But anyway, it's available in the master branch of Zeppelin. So, to add Kotlin support to Zeppelin, you need to build your own version from the sources.

To build a custom version of Zeppelin, you will need:

— [Git](#)
— [Maven](#),
— [JDK 8](#)
— The dependencies listed [here](#)

First, checkout the master branch from the Zeppelin repository:

```
git clone --depth=1 git@github.com:apache/zeppelin.git
```

or

```
git clone --depth=1 https://github.com/apache/zeppelin.git
```

To build Zeppelin with Maven, go to the Zeppelin directory and run the following command:

```
mvn clean package -DskipTests -Pspark-2.4 -Pscala-2.11
```

Then run Zeppelin with the following command:

```
./bin/zeppelin-daemon.sh start
```

Now the Zeppelin UI is available on `http://localhost:8089`.

To learn about how to deploy Zeppelin with Kotlin support in a Spark cluster, see [this instruction](#).

Libraries

For software engineering, an important part of any domain is the availability of frameworks for related areas. For data science, this includes such areas as machine learning, data analysis, visualization, and many others. Fortunately, there are already plenty of frameworks written in Kotlin for data science. There are even more frameworks written in Java, which is perfect as they can be called from Kotlin code seamlessly.

Below are two short lists of libraries that you may find useful for data science.

Kotlin libraries

— 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. For more information, see below.

— kravis is another library for the visualization of tabular data inspired by Python's ggplot.

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

— 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.

Lets-Plot for Kotlin

Lets-Plot for Kotlin is a Kotlin API for the [Lets-Plot](#) library - an open-source plotting library for statistical data written entirely in Kotlin. Lets-Plot was built on the concept of layered graphics first described in Leland Wilkinson's work [The Grammar of Graphics](#) and later implemented in the [ggplot2](#) package for R.

Lets-Plot for Kotlin is tightly integrated with the [Kotlin kernel for Jupyter notebooks](#). Once you have the Kotlin kernel installed and enabled, add the following line to a Jupyter notebook:

```
%use lets-plot
```

That's it, now you can call functions from Lets-Plot and see the results.

Kotlin bindings for NumPy

[KNumpy](#) (**Kotlin Bindings for NumPy**) is a Kotlin library that enables calling NumPy functions from the Kotlin code. [NumPy](#) is a popular package for scientific computing with Python. It provides powerful capabilities for multi-dimensional array processing, linear algebra, Fourier transform, random numbers, and other mathematical tasks.

KNumpy provides statically typed wrappers for NumPy functions. Thanks to the functional capabilities of Kotlin, the API of KNumpy is very similar to the one for NumPy. This lets developers that are experienced with NumPy easily switch to KNumpy. Here are two equal code samples:

```python
# Python

import numpy as np
a = np.arange(15).reshape(3, 5)

print(a.shape == (3, 5))        # True
print(a.ndim == 2)              # True
print(a.dtype.name)             # 'int64'

b = (np.arange(15) ** 2).reshape(3, 5)
```

```kotlin
// Kotlin

import org.jetbrains.numkt.*

fun main() {
    val a = arange(15).reshape(3, 5)

    println(a.shape.contentEquals(intArrayOf(3, 5))) // true
    println(a.ndim == 2)                             // true
    println(a.dtype)                                 // class java.lang.Integer

    // create an array of ints, we square each element and the shape to (3, 5)
    val b = (arange(15) `**` 2).reshape(3, 5)
}
```

Unlike Python, Kotlin is a statically typed language. This lets you avoid entire classes of runtime errors with KNumpy: the Kotlin compiler detects them at earlier stages.

```python
# Python

import numpy as np

# ...

a = np.ones((3, 3), dtype=int) * 3
b = np.random.random((3, 3))

b *= a # Success
a *= b # TypeError at runtime
```

```kotlin
// Kotlin

// ...

val a = ones<Int>(3, 3) * 3
val b = Random.random(3, 3)

b *= a // Success
a *= b // Compilation error:
// Type mismatch: inferred type is KtNDArray<Double> but KtNDArray<Int> was expected
```

Coroutines for asynchronous programming and more

Asynchronous or non-blocking programming is the new reality. Whether we're creating server-side, desktop or mobile applications, it's important that we provide an experience that is not only fluid from the user's perspective, but scalable when needed.

There are many approaches to this problem, and in Kotlin we take a very flexible one by providing Coroutine support at the language level and delegating most of the functionality to libraries, much in line with Kotlin's philosophy.

As a bonus, coroutines not only open the doors to asynchronous programming, but also provide a wealth of other possibilities such as concurrency, actors, etc.

How to Start

Tutorials and Documentation

New to Kotlin? Take a look at the Getting Started page.

Selected documentation pages:

— Coroutines Guide

— Basics

— Channels

— Coroutine Context and Dispatchers

— Shared Mutable State and Concurrency

— Asynchronous Flow

Recommended tutorials:

— Your first coroutine with Kotlin

— Asynchronous Programming

— Introduction to Coroutines and Channels hands-on lab

Example Projects

— kotlinx.coroutines Examples and Sources

— KotlinConf app

Even more examples are on GitHub

Multiplatform Programming

> Multiplatform projects are an experimental feature in Kotlin 1.2 and 1.3. All of the language and tooling features described in this document are subject to change in future Kotlin versions.

Working on all platforms is an explicit goal for Kotlin, but we see it as a premise to a much more important goal: sharing code between platforms. With support for JVM, Android, JavaScript, iOS, Linux, Windows, Mac and even embedded systems like STM32, Kotlin can handle any and all components of a modern application. And this brings the invaluable benefit of reuse for code and expertise, saving the effort for tasks more challenging than implementing everything twice or multiple times.

How it works

Overall, multiplatform is not about compiling all code for all platforms. This model has its obvious limitations, and we understand that modern applications need access to unique features of the platforms they are running on. Kotlin doesn't limit you to the common subset of all APIs in the world. Every component can share as much code as needed with others but can access platform APIs at any time through the [expect/actual mechanism](#) provided by the language.

Here's an example of code sharing and interaction between the common and platform logic in a minimalistic logging framework. The common code would look like this:

```kotlin
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)
```

├ *compiled for all platforms*

├ *expected platform-specific API*

├ *expected API can be used in the common code*

It expects the targets to provide platform-specific implementations for `writeLogMessage`, and the common code can now use this declaration without any consideration of how it is implemented.

On the JVM, one could provide an implementation that writes the log to the standard output:

```kotlin
internal actual fun writeLogMessage(message: String, logLevel: LogLevel) {
    println("[$logLevel]: $message")
}
```

In the JavaScript world, a completely different set of APIs is availiable, so one could instead implement logging to the console:

```
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)
    }
}
```

In 1.3 we reworked the entire multiplatform model. The new DSL we have for describing multiplatform Gradle projects is much more flexible, and we'll keep working on it to make project configuration straightforward.

Multiplatform Libraries

Common code can rely on a set of libraries that cover everyday tasks such as HTTP, serialization, and managing coroutines. Also, an extensive standard library is available on all platforms.

You can always write your own library providing a common API and implementing it differently on every platform.

Use cases

Android — iOS

Sharing code between mobile platforms is one of the major Kotlin Multiplatform use cases, and it is now possible to build mobile applications with parts of the code, such as business logic, connectivity, and more, shared between Android and iOS.

See:

— Mobile Multiplatform features, case studies and examples
— Setting up a Mobile Multiplatform Project

Client — Server

Another scenario when code sharing may bring benefits is a connected application where the logic may 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.

How to start

Tutorials and Documentation

New to Kotlin? Take a look at the [Getting Started](#) page.

Suggested documentation pages:

— [Setting up a Multiplatform Project](#)

— [Platform-Specific Declarations](#)

Recommended tutorials:

— [Multiplatform Kotlin Library](#)

— [Multiplatform Project: iOS and Android](#)

Example Projects

— [KotlinConf app](#)

— [KotlinConf Spinner app](#)

Even more examples are on [GitHub](#)

What's New in Kotlin 1.1

Table of Contents

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 front-end 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:

```
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
intln(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 [coroutine documentation](#) and [tutorial](#).

Note that coroutines are currently considered an **experimental feature**, meaning that the Kotlin team is not committing to supporting the backwards compatibility of this feature after the final 1.1 release.

Other Language Features

Type aliases

A type alias allows you to define an alternative name for an existing type. This is most useful for generic types such as collections, as well as for function types. Here is an example:

```
typealias OscarWinners = Map<String, String>

fun countLaLaLand(oscarWinners: OscarWinners) =
        oscarWinners.count { it.value.contains("La La Land") }

// Note that the type names (initial and the type alias) are interchangeable:
fun checkLaLaLandIsTheBestMovie(oscarWinners: Map<String, String>) =
        oscarWinners["Best picture"] == "La La Land"
```

See the [documentation](#) and [KEEP](#) for more details.

Bound callable references

You can now use the `::` operator to get a [member reference](#) pointing to a method or property of a specific object instance. Previously this could only be expressed with a lambda. Here's an example:

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

Read the documentation and KEEP for more details.

Sealed and data classes

Kotlin 1.1 removes some of the restrictions on sealed and data classes that were present in Kotlin 1.0. Now you can define subclasses of a top-level sealed class on the top level in the same file, and not just as nested classes of the sealed class. Data classes can now extend other classes. This can be used to define a hierarchy of expression classes nicely and cleanly:

```
sealed class Expr

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

fun eval(expr: Expr): Double = when (expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
}
val e = eval(Sum(Const(1.0), Const(2.0)))
```

Read the documentation or sealed class and data class KEEPs 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:

```
val map = mapOf(1 to "one", 2 to "two")
// before
println(map.mapValues { entry ->
  val (key, value) = entry
  "$key -> $value!"
})
// now
println(map.mapValues { (key, value) -> "$key -> $value!" })
```

Read the 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:

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

This also works in destructuring declarations:

```
val (_, status) = getResult()
```

Read the KEEP for more details.

Underscores in numeric literals

Just as in Java 8, Kotlin now allows to use underscores in numeric literals to separate groups of digits:

```
val oneMillion = 1_000_000
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

Read the KEEP for more details.

Shorter syntax for properties

For properties with the getter defined as an expression body, the property type can now be omitted:

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

Inline property accessors

You can now mark property accessors with the `inline` modifier if the properties don't have a backing field. Such accessors are compiled in the same way as inline functions.

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

You can also mark the entire property as `inline` - then the modifier is applied to both accessors.

Read the 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:

```
val answer by lazy {
    println("Calculating the answer...")
    42
}
if (needAnswer()) {                        // returns the random value
    println("The answer is $answer.")    // answer is calculated at this point
}
else {
    println("Sometimes no answer is the answer...")
}
```

Read the KEEP for more details.

Interception of delegated property binding

For delegated properties, it is now possible to intercept delegate to property binding using the `provideDelegate` operator. For example, if we want to check the property name before binding, we can write something like this:

```
class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(thisRef: MyUI, prop: KProperty<*>):
ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        ... // property creation
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ... }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ... }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

The `provideDelegate` method will be called for each property during the creation of a `MyUI` instance, and it can perform the necessary validation right away.

Read the documentation for more details.

Generic enum value access

It is now possible to enumerate the values of an enum class in a generic way.

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

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

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 [documentation](#) and [KEEP](#) for more details.

rem operator

The `mod` operator is now deprecated, and `rem` is used instead. See [this issue](#) for motivation.

Standard library

String to number conversions

There is a bunch of new extensions on the String class to convert it to a number without throwing an exception on invalid number: `String.toIntOrNull(): Int?`, `String.toDoubleOrNull(): Double?` etc.

```
val port = System.getenv("PORT")?.toIntOrNull() ?: 80
```

Also integer conversion functions, like `Int.toString()`, `String.toInt()`, `String.toIntOrNull()`, each got an overload with `radix` parameter, which allows to specify the base of conversion (2 to 36).

onEach()

`onEach` is a small, but useful extension function for collections and sequences, which allows to perform some action, possibly with side-effects, on each element of the collection/sequence in a chain of operations. On iterables it behaves like `forEach` but also returns the iterable instance further. And on sequences it returns a wrapping sequence, which applies the given action lazily as the elements are being iterated.

```
inputDir.walk()
        .filter { it.isFile && it.name.endsWith(".txt") }
        .onEach { println("Moving $it to $outputDir") }
        .forEach { moveFile(it, File(outputDir, it.toRelativeString(inputDir))) }
```

also(), takeIf() and takeUnless()

These are three general-purpose extension functions applicable to any receiver.

`also` is like `apply`: it takes the receiver, does some action on it, and returns that receiver. The difference is that in the block inside `apply` the receiver is available as `this`, while in the block inside `also` it's available as `it` (and you can give it another name if you want). This comes handy when you do not want to shadow `this` from the outer scope:

```
fun Block.copy() = Block().also {
    it.content = this.content
}
```

`takeIf` is like `filter` for a single value. It checks whether the receiver meets the predicate, and returns the receiver, if it does or `null` if it doesn't. Combined with an elvis-operator and early returns it allows to write constructs like:

```
val outDirFile = File(outputDir.path).takeIf { it.exists() } ?: return false
// do something with existing outDirFile
```

```
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
```

`takeUnless` is the same as `takeIf`, but it takes the inverted predicate. It returns the receiver when it *doesn't* meet the predicate and `null` otherwise. So one of the examples above could be rewritten with `takeUnless` as following:

```
val index = input.indexOf(keyword).takeUnless { it < 0 } ?: error("keyword not found")
```

It is also convenient to use when you have a callable reference instead of the lambda:

```
val result = string.takeUnless(String::isEmpty)
```

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:

```
val frequencies = words.groupingBy { it.first() }.eachCount()
```

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.

```
val map = mapOf("key" to 42)
val emptyMap = map - "key"
```

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.

```
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 })
```

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:

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

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.

```
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
```

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).

```
val array = arrayOf("a", "b", "c")
println(array.toString())  // JVM implementation: type-and-hash gibberish
println(array.contentToString())  // nicely formatted as list
```

JVM Backend

Java 8 bytecode support

Kotlin has now the option of generating Java 8 bytecode (`-jvm-target 1.8` command line option or the corresponding options in 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:

```kotlin
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:

```kotlin
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:

```kotlin
fun main(args: Array<String>) {
    jquery(".toggle-button").click {
        jquery(".toggle-panel").toggle(300)
    }
}
```

What's New in Kotlin 1.2

Table of Contents

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 common code:

```
// expected platform-specific API:
expect fun hello(world: String): String

fun greet() {
    // usage of the expected API:
    val greeting = hello("multi-platform world")
    println(greeting)
}

expect class URL(spec: String) {
    open fun getHost(): String
    open fun getPath(): String
}
```

In JVM platform code:

```kotlin
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 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:

```kotlin
@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:

```kotlin
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() }},
...")
}
```

Checking whether a lateinit var is initialized

You can now check whether a lateinit var has been initialized using `isInitialized` on the property reference:

```kotlin
    println("isInitialized before assignment: " + this::lateinitVar.isInitialized)
    lateinitVar = "value"
    println("isInitialized after assignment: " + this::lateinitVar.isInitialized)
```

Inline functions with default functional parameters

Inline functions are now allowed to have default values for their inlined functional parameters:

```kotlin
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)" }
```

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:

```kotlin
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:

```kotlin
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<*>
```

Also, smart casts in a lambda are now allowed for local variables that are only modified before the lambda:

```kotlin
val flag = args.size == 0
var x: String? = null
if (flag) x = "Yahoo!"

run {
    if (x != null) {
        println(x.length) // x is smart cast to String
    }
}
```

Support for ::foo as a shorthand for this::foo

A bound callable reference to a member of `this` can now be written without explicit receiver, `::foo` instead of `this::foo`. This also makes callable references more convenient to use in lambdas where you refer to a member of the outer receiver.

Breaking change: sound smart casts after try blocks

Earlier, Kotlin used assignments made inside a `try` block for smart casts after the block, which could break type- and null-safety and lead to runtime failures. This release fixes this issue, making the smart casts more strict, but breaking some code that relied on such smart casts.

To switch to the old smart casts behavior, pass the fallback flag `-Xlegacy-smart-cast-after-try` as the compiler argument. It will become deprecated in Kotlin 1.3.

Deprecation: data classes overriding copy

When a data class derived from a type that already had the `copy` function with the same signature, the `copy` implementation generated for the data class used the defaults from the supertype, leading to counter-intuitive behavior, or failed at runtime if there were no default parameters in the supertype.

Inheritance that leads to a `copy` conflict has become deprecated with a warning in Kotlin 1.2 and will be an error in Kotlin 1.3.

Deprecation: nested types in enum entries

Inside enum entries, defining a nested type that is not an `inner class` has been deprecated due to issues in the initialization logic. This causes a warning in Kotlin 1.2 and will become an error in Kotlin 1.3.

Deprecation: single named argument for vararg

For consistency with array literals in annotations, passing a single item for a vararg parameter in the named form (`foo(items = i)`) has been deprecated. Please use the spread operator with the corresponding array factory functions:

```
foo(items = *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`):

```
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 }
```

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`:

```
val items = (1..5).toMutableList()

items.shuffle()
println("Shuffled items: $items")

items.replaceAll { it * 2 }
println("Items doubled: $items")

items.fill(5)
println("Items filled with 5: $items")
```

Math operations in kotlin-stdlib

Satisfying the longstanding request, Kotlin 1.2 adds the `kotlin.math` API for math operations that is common for JVM and JS and contains the following:

— Constants: `PI` and `E`;

— Trigonometric: `cos`, `sin`, `tan` and inverse of them: `acos`, `asin`, `atan`, `atan2`;

— Hyperbolic: `cosh`, `sinh`, `tanh` and their inverse: `acosh`, `asinh`, `atanh`

— Exponentation: `pow` (an extension function), `sqrt`, `hypot`, `exp`, `expm1`;

— Logarithms: `log`, `log2`, `log10`, `ln`, `ln1p`;

— Rounding:

 — `ceil`, `floor`, `truncate`, `round` (half to even) functions;

 — `roundToInt`, `roundToLong` (half to integer) extension functions;

— Sign and absolute value:

 — `abs` and `sign` functions;

 — `absoluteValue` and `sign` extension properties;

 — `withSign` extension function;

— `max` and `min` of two values;

— Binary representation:

 — `ulp` extension property;

 — `nextUp`, `nextDown`, `nextTowards` extension functions;

 — `toBits`, `toRawBits`, `Double.fromBits` (these are in the `kotlin` package).

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

Operators and conversions for BigInteger and BigDecimal

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

— `toBigInteger` for `Int` and `Long`;

— `toBigDecimal` for `Int`, `Long`, `Float`, `Double`, and `BigInteger`;

— Arithmetic and bitwise operator functions:

  — Binary operators `+`, `-`, `*`, `/`, `%` and infix functions `and`, `or`, `xor`, `shl`, `shr`;

  — Unary operators `-`, `++`, `--`, and a function `inv`.

Floating point to bits conversions

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

— `toBits` and `toRawBits` returning `Long` for `Double` and `Int` for `Float`;

— `Double.fromBits` and `Float.fromBits` for creating floating point numbers from the bit representation.

Regex is now serializable

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

Closeable.use calls Throwable.addSuppressed if available

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

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

JVM Backend

Constructor calls normalization

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

To mitigate this problem for the users of such bytecode processing tools, we've added a command-line 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 1.3

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 compiler:
    // "if this function returns successfully, then passed 'condition' is true"
    contract { returns() implies condition }
    if (!condition) throw IllegalArgumentException(...)
}

fun foo(s: String?) {
    require(s is String)
    // s is smartcasted to 'String' here, because otherwise
    // 'require' would have throw an exception
}
```

— Improving the variable initialization analysis in the presence of high-order functions:

```
fun synchronize(lock: Any?, block: () -> Unit) {
    // It tells compiler:
    // "This function will invoke 'block' here and now, and exactly one time"
    contract { callsInPlace(block, EXACTLY_ONCE) }
}

fun foo() {
    val x: Int
    synchronize(lock) {
        x = 42 // Compiler knows that lambda passed to 'synchronize' is called
               // exactly once, so no reassignment is reported
    }
    println(x) // Compiler knows that lambda will be definitely called, performing
               // initialization, so 'x' is considered to be initialized here
}
```

Contracts in stdlib

`stdlib` already makes use of contracts, which leads to improvements in the analyses described above. This part of contracts is **stable**, meaning that you can benefit from the improved analysis right now without any additional opt-ins:

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

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 a programmer's responsibility to write correct and sound contracts.

Custom contracts are introduced by the 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 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 companion 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:

```kotlin
interface Foo {
    companion object {
        @JvmField
        val answer: Int = 42

        @JvmStatic
        fun sayHello() {
            println("Hello, world!")
        }
    }
}
```

It is equivalent to this Java code:

```java
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:

```kotlin
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!

```kotlin
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:

```kotlin
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 *progressive* compiler mode, which can be enabled by passing the argument `-progressive` to the compiler.

In 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 which are 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 available only since Kotlin 1.3 and currently are *experimental*. 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:

```kotlin
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:

```
fun main() {
    // In the next line no constructor call happens, and
    // at the runtime 'name' contains just string "Kotlin"
    val name = Name("Kotlin")
    println(name.s)
}
```

See reference for inline classes for details.

Unsigned integers

Unsigned integers are available only since Kotlin 1.3 and currently are *experimental*. See details in the reference.

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:

```
// You can define unsigned types using literal suffixes
val uint = 42u
val ulong = 42uL
val ubyte: UByte = 255u

// You can convert signed types to unsigned and vice versa via stdlib extensions:
val int = uint.toInt()
val byte = ubyte.toByte()
val ulong2 = byte.toULong()

// Unsigned types support similar operators:
val x = 20u + 22u
val y = 1u shl 8
val z = "128".toUByte()
val range = 1u..5u
```

See reference for details.

@JvmDefault

@JvmDefault is only available since Kotlin 1.3 and currently is *experimental*. See details in the reference page.

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:

```kotlin
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:

```kotlin
val number = Random.nextInt(42)  // number is in range [0, limit)
println(number)
```

isNullOrEmpty/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.

Copying elements between two existing arrays

The `array.copyInto(targetArray, targetOffset, startIndex, endIndex)` functions for the existing array types, including the unsigned arrays, make it easier to implement array-based containers in pure Kotlin.

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

sourceArr.copyInto(targetArr, startIndex = 0, endIndex = 3)
println(targetArr.contentToString())
```

associateWith

It is quite a common situation to have a list of keys and want to build a map by associating each of these keys with some value. It was possible to do it before with the `associate { it to getValue(it) }` function, but now we're introducing a more efficient and easy to explore alternative: `keys.associateWith { getValue(it) }`.

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

ifEmpty and ifBlank functions

Collections, maps, object arrays, char sequences, and sequences now have an `ifEmpty` function, which allows specifying a fallback value that will be used instead of the receiver if it is empty:

```
fun printAllUppercase(data: List<String>) {
    val result = data
    .filter { it.all { c -> c.isUpperCase() } }
        .ifEmpty { listOf("<no uppercase>") }
    result.forEach { println(it) }
}

printAllUppercase(listOf("foo", "Bar"))
printAllUppercase(listOf("FOO", "BAR"))
```

Char sequences and strings in addition have an `ifBlank` extension that does the same thing as `ifEmpty`, but checks for a string being all whitespace instead of empty.

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

Sealed classes in reflection

We've added a new API to `kotlin-reflect` that can be used to enumerate all the direct subtypes of a `sealed` class, namely `KClass.sealedSubclasses`.

Smaller changes

— `Boolean` type now has companion.

— `Any?.hashCode()` extension, which returns 0 for `null`.

— `Char` now provides `MIN_VALUE` / `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 the IDE. 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](#).

> Please, note, that even though kotlinx.serialization now ships with the Kotlin Compiler distribution, it is still considered to be an experimental feature.

Scripting update

> Please note, that scripting is an experimental feature, meaning that no compatibility guarantees on the API are given.

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 a .kts extension which you can run and get evaluation results directly in the editor.

Consult the general [Scratches documentation](#) for details.

Getting Started

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!")
}
```

Functions

Function having two `Int` parameters with `Int` return type:

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

Function with an expression body and inferred return type:

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

Function returning no meaningful value:

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

`Unit` return type can be omitted:

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

See Functions.

Variables

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

```
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
```

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

```
var x = 5 // `Int` type is inferred
x += 1
```

Top-level variables:

```
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
```

See also Properties And Fields.

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

```kotlin
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"
```

See String templates for details.

Conditional expressions

```kotlin
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}
```

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

```kotlin
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

See if-expressions.

Nullable values and `null` checks

A reference must be explicitly marked as nullable when `null` value is possible.

Return `null` if `str` does not hold an integer:

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

Use a function returning nullable value:

```kotlin
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")
    }
}
```

or

```
// ...
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)
```

See Null-safety.

Type checks and automatic casts

The `is` operator checks if an expression is an instance of a type. If an immutable local variable or property is checked for a specific type, there's no need to cast it explicitly:

```
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` is automatically cast to `String` in this branch
        return obj.length
    }

    // `obj` is still of type `Any` outside of the type-checked branch
    return null
}
```

or

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

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

or even

```
fun getStringLength(obj: Any): Int? {
    // `obj` is automatically cast to `String` on the right-hand side of `&&`
    if (obj is String && obj.length > 0) {
        return obj.length
    }

    return null
}
```

See Classes and Type casts.

for loop

```kotlin
val items = listOf("apple", "banana", "kiwifruit")
for (item in items) {
    println(item)
}
```

or

```kotlin
val items = listOf("apple", "banana", "kiwifruit")
for (index in items.indices) {
    println("item at $index is ${items[index]}")
}
```

See [for loop](#).

while loop

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

See [while loop](#).

when expression

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

See [when expression](#).

Ranges

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

```kotlin
val x = 10
val y = 9
if (x in 1..y+1) {
    println("fits in range")
}
```

Check if a number is out of range:

```
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")
}
```

Iterating over a range:

```
for (x in 1..5) {
    print(x)
}
```

or over a progression:

```
for (x in 1..10 step 2) {
    print(x)
}
println()
for (x in 9 downTo 0 step 3) {
    print(x)
}
```

See Ranges.

Collections

Iterating over a collection:

```
for (item in items) {
    println(item)
}
```

Checking if a collection contains an object using `in` operator:

```
when {
    "orange" in items -> println("juicy")
    "apple" in items -> println("apple is fine too")
}
```

Using lambda expressions to filter and map collections:

```
val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
fruits
  .filter { it.startsWith("a") }
  .sortedBy { it }
  .map { it.toUpperCase() }
  .forEach { println(it) }
```

See Collections overview.

Creating basic classes and their instances

```
val rectangle = Rectangle(5.0, 2.0)
val triangle = Triangle(3.0, 4.0, 5.0)
```

See classes and objects and instances.

Idioms

A collection of random and frequently used idioms in Kotlin. If you have a favorite idiom, contribute it by sending a pull request.

Creating 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 = "") { ... }
```

Filtering a list

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

Or alternatively, even shorter:

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

Checking element presence in a collection.

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

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

String Interpolation

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

Instance Checks

```
when (x) {
    is Foo -> ...
    is Bar -> ...
    else   -> ...
}
```

Traversing a map/list of pairs

```
for ((k, v) in map) {
    println("$k -> $v")
}
```

`k`, `v` can be called anything.

Using ranges

```
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) { ... }
```

Read-only list

```
val list = listOf("a", "b", "c")
```

Read-only map

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

Accessing a map

```
println(map["key"])
map["key"] = value
```

Lazy property

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

Extension Functions

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

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

Creating a singleton

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

If not null shorthand

```
val files = File("Test").listFiles()

println(files?.size)
```

If not null and else shorthand

```
val files = File("Test").listFiles()

println(files?.size ?: "empty")
```

Executing 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. E.g. with the when-expression:

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

Calling multiple methods on an object instance (`with`)

```kotlin
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()
}
```

Configuring properties of an object (`apply`)

```kotlin
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

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

Convenient form for a 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)
```

Consuming a nullable Boolean

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

Swapping two variables

```
var a = 1
var b = 2
a = b.also { b = a }
```

TODO(): Marking code as incomplete

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 toolwindow.

Coding Conventions

This page contains the current coding style for the Kotlin language.

— [Source code organization](#)

— [Naming rules](#)

— [Formatting](#)

— [Documentation comments](#)

— [Avoiding redundant constructs](#)

— [Idiomatic use of language features](#)

— [Coding conventions for libraries](#)

Applying the style guide

To configure the IntelliJ formatter according to this style guide, please install Kotlin plugin version 1.2.20 or newer, go to **Settings | Editor | Code Style | Kotlin**, click **Set from...** link in the upper right corner, and select **Predefined style | Kotlin style guide** from the menu.

To verify that your code is formatted according to the style guide, go to the inspection settings and enable the **Kotlin | Style issues | 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 the 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 the [camel case](#) with an uppercase first letter (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 where the class itself is defined. When defining extension functions that make sense only for a specific client, put them next to the code of that client. Do not create files just to hold "all extensions of Foo".

Class layout

Generally, the contents of a class is sorted in the following order:

— Property declarations and initializer blocks
— Secondary constructors
— Method declarations
— 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 lower case 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 simply concatenate them together or use the
camel case (`org.example.myProject`).

— Names of classes and objects start with an upper case letter and use the camel case:

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

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

Function names

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

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

Exception: factory functions used to create instances of classes can have the same name as the
class being created:

```
abstract class Foo { /*...*/ }

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

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

Names for test methods

In tests (and **only** in tests), it's acceptable to 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:

```
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 (`enum class Color { RED, GREEN }`) or regular camel-case names starting with an uppercase first letter, 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
}
```

Choosing 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` etc.) 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

Use 4 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) {
        // ...
    }
}
```

(Note: 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 header formatting

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 we use inheritance, then the superclass constructor call or 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 (4 spaces) for constructor parameters.

Rationale: This ensures that properties declared in the primary constructor have the same indentation as properties declared in the body of a class.

Modifiers

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
companion
inline
infix
operator
data
```

Place all annotations before modifiers:

```
@Named("Foo")
private val foo: Foo
```

Unless you're working on a library, omit redundant modifiers (e.g. `public`).

Annotation formatting

Annotations are typically placed 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
```

Function formatting

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 (4 spaces) for function parameters.

> Rationale: 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 body formatting

If the function has an expression body that doesn't fit in the same line as the declaration, put the `=` sign on the first line. Indent the expression body by 4 spaces.

```
fun f(x: String) =
    x.length
```

Property formatting

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 equals sign and indent the initializer by four spaces:

```
private val defaultCharset: Charset? =
    EncodingRegistry.getInstance().getDefaultCharsetForPropertiesFiles(file)
```

Formatting 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 4 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)
}
```

> Rationale: Tidy alignment and clear separation of condition and statement body

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 call formatting

In long argument lists, put a line break after the opening parenthesis. Indent arguments by 4 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.

Chained call wrapping

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.

Lambda formatting

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, it should be passed 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)
}
```

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) { /*...*/ }
```

Avoiding 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

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>
```

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, parameters should be always declared 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)
```

Using conditional statements

Prefer using the expression form of `try`, `if` and `when`. Examples:

```
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`. Instead of

```
when (x) {
    null -> // ...
    else -> // ...
}
```

use `if (x == null) ... else ...`

Prefer using `when` if there are three or more options.

Using 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.

Using 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
```

Using strings

Prefer using string templates to string concatenation.

Prefer to use multiline strings instead of 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:

```
assertEquals(
    """
    Foo
    Bar
    """.trimIndent(),
    value
)

val a = """if(a > 1) {
        |    return a
        |}""".trimMargin()
```

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

Using 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.

Using 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`.

Don't 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.

Example:

```
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) initialised 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)
}
```

Using scope functions apply/with/run/also/let

Kotlin provides a variety 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)

Basics

Basic Types

In Kotlin, everything is an object in the sense that we can call member functions and properties on any variable. Some of the 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, characters, booleans, arrays, and strings.

Numbers

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
```

For floating-point numbers, Kotlin provides 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
Double	64	53	11	15-16

For variables initialized with fractional numbers, the compiler infers the `Double` type. 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 pi = 3.14 // Double
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 = 1.1
    val f = 1.1f

    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`

NOTE: 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`

Underscores in numeric literals (since 1.1)

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
```

Representation

On the Java platform, numbers are physically stored as JVM primitive types, unless we need a nullable number reference (e.g. `Int?`) or generics are involved. In the latter cases numbers are boxed.

Note that boxing of numbers does not necessarily preserve identity:

```
val a: Int = 10000
println(a === a) // Prints 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
println(boxedA === anotherBoxedA) // !!!Prints 'false'!!!
```

On the other hand, it preserves equality:

```
val a: Int = 10000
println(a == a) // Prints 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
println(boxedA == anotherBoxedA) // Prints 'true'
```

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 all over the place, not to mention identity.

As a consequence, smaller types are NOT implicitly converted to bigger types. This means that we cannot assign a value of type `Byte` to an `Int` variable without an explicit conversion

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

We can use explicit conversions to widen numbers

```
val i: Int = b.toInt() // OK: explicitly widened
print(i)
```

Every number type supports the following conversions:

— `toByte(): Byte`

— `toShort(): Short`

— `toInt(): Int`

— `toLong(): Long`

— `toFloat(): Float`

— `toDouble(): Double`

— `toChar(): Char`

Absence of implicit conversions is rarely noticeable 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 arithmetic operations over numbers, which are declared as members of appropriate classes (but the compiler optimizes the calls down to the corresponding instructions). See Operator overloading.

As of bitwise operations, there're no special characters for them, but just named functions that can be called in infix form, for example:

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

Here is the complete list of bitwise operations (available for `Int` and `Long` only):

— `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`

Characters

Characters are represented by the type `Char`. They can not be treated directly as numbers

```
fun check(c: Char) {
    if (c == 1) { // ERROR: incompatible types
        // ...
    }
}
```

Character literals go in single quotes: `'1'`. Special characters can be escaped using a backslash. The following escape sequences are supported: `\t`, `\b`, `\n`, `\r`, `\'`, `\"`, `\\` and `\$`. To encode any other character, use the Unicode escape sequence syntax: `'\uFF00'`.

We can explicitly convert a character to an `Int` number:

```
fun decimalDigitValue(c: Char): Int {
    if (c !in '0'..'9')
        throw IllegalArgumentException("Out of range")
    return c.toInt() - '0'.toInt() // Explicit conversions to numbers
}
```

Like numbers, characters are boxed when a nullable reference is needed. Identity is not preserved by the boxing operation.

Booleans

The type `Boolean` represents booleans, and has two values: `true` and `false`.

Booleans are boxed if a nullable reference is needed.

Built-in operations on booleans include

— `||` – lazy disjunction

— `&&` – lazy conjunction

— `!` - negation

Arrays

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

```kotlin
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, we can use a library function `arrayOf()` and pass the item values to it, so that `arrayOf(1, 2, 3)` creates an array `[1, 2, 3]`. Alternatively, the `arrayOfNulls()` library 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 can return the initial value of each array element given its index:

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

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 specialized classes to 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:

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

```kotlin
// 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 }
```

Unsigned integers

Kotlin introduces following types for unsigned integers:

— `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

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

Unsigned types are implemented using another experimental feature, namely [inline classes](#).

Specialized classes

Same as for primitives, each of unsigned type has corresponding type that represents array, specialized for that unsigned type:

— `kotlin.UByteArray` : an array of unsigned bytes

— `kotlin.UShortArray` : an array of unsigned shorts

— `kotlin.UIntArray` : an array of unsigned ints

— `kotlin.ULongArray` : an array of unsigned longs

Same as for signed integer arrays, they provide similar API to `Array` class without boxing overhead.

Also, [ranges and progressions](#) supported for `UInt` and `ULong` by classes `kotlin.ranges.UIntRange`, `kotlin.ranges.UIntProgression`, `kotlin.ranges.ULongRange`, `kotlin.ranges.ULongProgression`

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/Long):

— suffixes `u` and `U` tag literal as unsigned. Exact type will be determined based on the expected type. If no expected type is provided, `UInt` or `ULong` will be chosen based 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
```

— suffixes `uL` and `UL` explicitly tag literal as unsigned long.

```
val a = 1UL // ULong, even though no expected type provided and constant fits into UInt
```

Experimental status of unsigned integers

The design of unsigned types is experimental, meaning that this feature is moving fast and no compatibility guarantees are given. When using unsigned arithmetics in Kotlin 1.3+, warning will be reported, indicating that this feature is experimental. To remove warning, you have to opt-in for experimental usage of unsigned types.

There are two possible ways to opt-in for unsigned types: with marking your API as experimental too, or without doing that.

— to propagate experimentality, either annotate declarations which use unsigned integers with `@ExperimentalUnsignedTypes` or pass `-Xexperimental=kotlin.ExperimentalUnsignedTypes` to the compiler (note that the latter will make *all* declaration in compiled module experimental)
— to opt-in without propagating experimentality, either annotate declarations with `@UseExperimental(ExperimentalUnsignedTypes::class)` or pass `-Xuse-experimental=kotlin.ExperimentalUnsignedTypes`

It's up to you to decide if your clients have to explicitly opt-in into usage of your API, but bear in mind that unsigned types are an experimental feature, so API which uses them can be suddenly broken due to changes in language.

See also or Experimental API [KEEP](#) for technical details.

Further discussion

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

Strings

Strings are represented by the type `String`. Strings are immutable. Elements of a string are characters that can be accessed by the indexing operation: `s[i]`. A string can be iterated over with a `for`-loop:

```
for (c in str) {
    println(c)
}
```

You can concatenate strings using 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:

```
val s = "abc" + 1
println(s + "def")
```

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 have escaped characters in them and 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)
"""
```

You can remove leading whitespace with `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, i.e. 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 simple name:

```
val i = 10
println("i = $i") // prints "i = 10"
```

or an arbitrary expression in curly braces:

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

Templates are supported both inside raw strings and inside escaped strings. If you need to represent a literal $ character in a raw string (which doesn't support backslash escaping), you can use the following syntax:

```
val price = """
${'$'}9.99
"""
```

Packages

A source file may start with a package declaration:

```kotlin
package org.example

fun printMessage() { /*...*/ }
class Message { /*...*/ }

// ...
```

All the contents (such as classes and functions) of the source file are contained by the package declared. 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 that has no name.

Default Imports

A number of packages are imported into every Kotlin file by default:

— [kotlin.*](#)
— [kotlin.annotation.*](#)
— [kotlin.collections.*](#)
— [kotlin.comparisons.*](#) (since 1.1)
— [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. Syntax for imports is described in the [grammar](#).

We can import either a single name, e.g.

```
import org.example.Message // Message is now accessible without qualification
```

or all the accessible contents of a scope (package, class, object etc):

```
import org.example.* // everything in 'org.example' becomes accessible
```

If there is a name clash, we 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).

Control Flow: if, when, for, while

If Expression

In Kotlin, `if` is an expression, i.e. it returns a value. Therefore there is no ternary operator (condition ? then : else), because ordinary `if` works fine in this role.

```
// Traditional usage
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
```

`if` branches can be blocks, and 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 rather than a statement (for example, returning its value or assigning it to a variable), the expression is required to have an `else` branch.

See the [grammar for `if`](#).

When Expression

`when` replaces the switch operator of C-like languages. In the simplest form it 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 satisfied 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 (as, for example, with enum class entries and sealed class subtypes).

If many cases should be handled in the same way, the branch conditions may be combined with a comma:

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

We 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")
}
```

We 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 possibility is to check 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")
    x.isEven() -> print("x is even")
    else -> print("x is funny")
}
```

Since Kotlin 1.3, it is possible to 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)
        }
```

Scope of variable, introduced in when subject, is restricted to when body.

See the grammar for when.

For Loops

for loop iterates through anything that provides an iterator. This is equivalent to the `foreach` loop in languages like C#. The syntax is as follows:

```
for (item in collection) print(item)
```

The body can be a block.

```
for (item: Int in ints) {
    // ...
}
```

As mentioned before, for iterates through anything that provides an iterator, i.e.

— has a member- or extension-function `iterator()`, whose return type

   — has a member- or extension-function `next()`, and

   — has a member- or 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:

```
for (i in 1..3) {
    println(i)
}
for (i in 6 downTo 0 step 2) {
    println(i)
}
```

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:

```
for (i in array.indices) {
    println(array[i])
}
```

Alternatively, you can use the `withIndex` library function:

```
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

See the grammar for for.

While Loops

while and do..while work as usual

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

See the grammar for while.

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@` are valid labels (see the [grammar](#)). To label an expression, we just put 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 expression, functions can be nested in Kotlin. Qualified `return`s 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:

```
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")
}
```

The `return`-expression returns from the nearest enclosing function, i.e. `foo`. (Note that such non-local returns are supported only for lambda expressions passed to inline functions.) If we need to return from a lambda expression, we have to label it and qualify the `return`:

```kotlin
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach lit@{
        if (it == 3) return@lit // local return to the caller of the lambda, i.e. the
forEach loop
        print(it)
    }
    print(" done with explicit label")
}
```

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.

```kotlin
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return@forEach // local return to the caller of the lambda, i.e. the
forEach loop
        print(it)
    }
    print(" done with implicit label")
}
```

Alternatively, we can replace the lambda expression with an anonymous function. A `return` statement in an anonymous function will return from the anonymous function itself.

```kotlin
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach(fun(value: Int) {
        if (value == 3) return  // local return to the caller of the anonymous fun, i.e.
the forEach loop
        print(value)
    })
    print(" done with anonymous function")
}
```

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:

```kotlin
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")
}
```

When returning a value, the parser gives preference to the qualified return, i.e.

```kotlin
return@a 1
```

means "return `1` at label `@a` " and not "return a labeled expression `(@a 1)` ".

Classes and Objects

Classes and Inheritance

Classes

Classes in Kotlin are declared using the keyword `class`:

```
class Invoice { /*...*/ }
```

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:

```
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}")
    }
}
```

Note that 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.toUpperCase()
}
```

In fact, for declaring properties and initializing them from the primary constructor, Kotlin has a concise syntax:

```
class Person(val firstName: String, val lastName: String, var age: Int) { /*...*/ }
```

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) { /*...*/ }
```

For more details, see [Visibility Modifiers](#).

Secondary constructors

The class can also declare **secondary constructors**, which are prefixed with `constructor`:

```
class Person {
    var children: MutableList<Person> = mutableListOf<Person>();
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

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<Person>();
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Note that 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:

```
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor")
    }
}
```

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 do not want your class to have a public constructor, you need to declare an empty primary constructor with non-default visibility:

```
class DontCreateMe private constructor () { /*...*/ }
```

> **NOTE**: 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, we call the constructor as if it were a regular function:

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

Note that 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

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.

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

As we mentioned before, we stick to making things explicit in Kotlin. So, Kotlin requires explicit modifiers for overridable members (we call them *open*) and for overrides:

```kotlin
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`:

```kotlin
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.

```kotlin
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.

```
open class Base(val name: String) {

    init { println("Initializing Base") }

    open val size: Int =
        name.length.also { println("Initializing size in Base: $it") }
}

class Derived(
    name: String,
    val lastName: String
) : Base(name.capitalize().also { println("Argument for Base: $it") }) {

    init { println("Initializing Derived") }

    override val size: Int =
        (super.size + lastName.length).also { println("Initializing size in Derived:
$it") }
}
```

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:

```kotlin
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`:

```kotlin
class FilledRectangle: Rectangle() {
    fun draw() { /* ... */ }
    val borderColor: String get() = "black"

    inner class Filler {
        fun fill() { /* ... */ }
        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()
        }
    }
}
```

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, we 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 we have to override `draw()` in `Square` and provide its own implementation that eliminates the ambiguity.

Abstract classes

A class and some of its members may be declared `abstract`. An abstract member does not have an implementation in its class. Note that we do not need to annotate an abstract class or function with open – it goes without saying.

We can override a non-abstract open member with an abstract one

```
open class Polygon {
    open fun draw() {}
}

abstract class Rectangle : Polygon() {
    override abstract 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.

Properties and Fields

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 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

```
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).

Examples:

```
var allByDefault: Int? // error: explicit initializer required, default getter and setter
implied
var initialized = 1 // has type Int, default getter and setter
```

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
```

We can define custom accessors for a property. If we define a custom getter, it will be called every time we access the property (this allows us to implement a computed property). Here's an example of a custom getter:

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

If we define a custom setter, it will be called every time we assign a value to the property. 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.

Since Kotlin 1.1, 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

Fields cannot be declared directly in Kotlin classes. 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 // Note: the initializer assigns the backing field directly
    set(value) {
        if (value >= 0) field = value
    }
```

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*:

```kotlin
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null) {
            _table = HashMap() // Type parameters are inferred
        }
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

> **On the JVM**: The access to private properties with default getters and setters is optimized so no function call overhead is introduced in this case.

Compile-Time Constants

Properties the value of which is known at compile time can be marked as *compile time constants* 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:

```kotlin
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:

```kotlin
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) and, since Kotlin 1.2, 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 (since 1.2)

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, i.e. 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 we declare many types in our supertype list, it may appear that we inherit more than one implementation of the same method. For 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 we derive a concrete class *C* from *A*, we, obviously, have to override *bar()* and provide an implementation.

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

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, used if there is no explicit modifier, 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 on the "top-level", i.e. 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.

Note: 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` — same as `private` + visible in subclasses too;

— `internal` — any client *inside this module* who sees the declaring class sees its `internal` members;

— `public` — any client who sees the declaring class sees its `public` members.

Note that 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:

```kotlin
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 (note that you need to add an explicit `constructor` keyword):

```kotlin
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 (i.e. a constructor of an `internal` class is only visible within the same module).

Local declarations

Local variables, functions and classes can not 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 the 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, we need to prefix its name with a *receiver type*, i.e. the type being extended. The following adds a `swap` function to `MutableList<Int>`:

```kotlin
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, we can call such a function on any `MutableList<Int>`:

```kotlin
val list = mutableListOf(1, 2, 3)
list.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'list'
```

Of course, this function makes sense for any `MutableList<T>`, and we can make it generic:

```kotlin
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
}
```

We declare the generic type parameter before the function name for it to be 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.

We would like to emphasize that extension functions are dispatched **statically**, i.e. they are not virtual by receiver type. This means that the 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:

```
open class Shape

class Rectangle: Shape()

fun Shape.getName() = "Shape"

fun Rectangle.getName() = "Rectangle"

fun printClassName(s: Shape) {
    println(s.getName())
}

printClassName(Rectangle())
```

This example prints "*Shape*", because the extension function being called depends only on the declared type of the parameter `s`, which is the `Shape` class.

If a class has a member function, and an extension function is defined which has the same receiver type, the same name, and is applicable to given arguments, the **member always wins**. For example:

```
class Example {
    fun printFunctionType() { println("Class method") }
}

fun Example.printFunctionType() { println("Extension function") }

Example().printFunctionType()
```

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:

```
class Example {
    fun printFunctionType() { println("Class method") }
}

fun Example.printFunctionType(i: Int) { println("Extension function") }

Example().printFunctionType(1)
```

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 is what allows you to 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
```

Note that, 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

Most of the time we 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, we need to 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("kotl.in"), 443).connect()
    //Host("kotl.in").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.

```kotlin
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 on 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

We frequently create classes whose main purpose is to hold data. In such a class some standard functionality and utility functions are often mechanically derivable from the data. In Kotlin, this is called a *data class* and is marked as `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;

— (before 1.1) Data classes may only implement interfaces.

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;

— Deriving a data class from a type that already has a `copy(...)` function with a matching signature is deprecated in Kotlin 1.2 and is prohibited in Kotlin 1.3.

— Providing explicit implementations for the `componentN()` and `copy()` functions is not allowed.

Since 1.1, 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

Note that 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.

```
val person1 = Person("John")
val person2 = Person("John")
person1.age = 10
person2.age = 20
```

Copying

It's often the case that we need to copy an object altering *some* of its properties, but keeping the rest unchanged. This is what `copy()` function is generated for. For the `User` class above, its implementation would be as follows:

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

This allows us to write:

```
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 are used for representing restricted class hierarchies, when a value can have one of the types from a limited set, but cannot have any other type. They are, in a sense, an extension of 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 which can contain state.

To declare a sealed class, you put the `sealed` modifier before the name of the class. A sealed class can have subclasses, but all of them must be declared in the same file as the sealed class itself. (Before Kotlin 1.1, the rules were even more strict: classes had to be nested inside the declaration of the sealed class).

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

(The example above uses one additional new feature of Kotlin 1.1: the possibility for data classes to extend other classes, including sealed classes.)

A sealed class is abstract by itself, it cannot be instantiated directly and can have `abstract` members.

Sealed classes are not allowed to have non-`private` constructors (their constructors are `private` by default).

Note that classes which extend subclasses of a sealed class (indirect inheritors) can be placed anywhere, not necessarily in the same file.

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
}
```

Generics

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

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

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

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

But if the parameters may be inferred, e.g. from the constructor arguments or by some other means, one is allowed to omit the type arguments:

```
val box = Box(1) // 1 has type Int, so the compiler figures out that we are talking about
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; // !!! The cause of the upcoming problem sits here. Java
prohibits this!
objs.add(1); // Here we 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, we'd put it this way:

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

But then, we can't do the following simple thing (which is perfectly safe):

```java
// 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, we 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
// 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 we can safely **read** `E`'s from items (elements of this collection are instances of a subclass of E), but **cannot write** to it since we do not know what objects comply to that unknown subtype of `E`. In return for this limitation, we have the desired behaviour: `Collection<String>` *is* a subtype of `Collection<? extends Object>`. In "clever 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: in Java we have `List<? super String>` 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>` (e.g., you can call `add(String)` or `set(int, String)`), while 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.

NOTE: 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

Suppose we have a generic interface `Source<T>` that does not have any methods that take `T` as a parameter, only methods that return `T`:

```java
// Java
interface Source<T> {
  T nextT();
}
```

Then, it would be perfectly safe to store a reference to an instance of `Source<String>` in a variable of type `Source<Object>` – there are no consumer-methods to call. But Java does not know this, and still prohibits it:

```java
// Java
void demo(Source<String> strs) {
  Source<Object> objects = strs; // !!! Not allowed in Java
  // ...
}
```

To fix this, we have to declare objects of type `Source<? extends Object>`, which is sort of meaningless, because we 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**: we 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 we provide 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 "clever words" they 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, we talk about **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, we can assign x to a variable of type Comparable<Double>
    val y: Comparable<Double> = x // OK!
}
```

We believe that the words **in** and **out** are 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) {
    fun get(index: Int): T { ... }
    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 we 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 **might** be doing bad things, i.e. it might attempt to **write**, say, a String to `from`, and if we actually passed an array of `Int` there, a `ClassCastException` would have been thrown sometime later.

Then, the only thing we want to ensure is that `copy()` does not do any bad things. We want to prohibit it from **writing** to `from`, and we can:

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

What has happened here is called **type projection**: we said that `from` is not simply an array, but a restricted (**projected**) one: we can only call those methods that return the type parameter `T`, in this case it means that we 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>`, i.e. 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>` we 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?>`.

Note: 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>` may 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, we need a separate **where**-clause:

```kotlin
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, e.g. `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 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
```

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) { ... }
})
```

Note: on the JVM, if the object is an instance of a functional Java interface (i.e. 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.

Initialization

Since each enum is an instance of the enum class, they 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.

Enum entries cannot contain nested types other than inner classes (deprecated in Kotlin 1.2).

Implementing Interfaces in Enum Classes

An enum class may 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:

```kotlin
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)
}
```

Working with Enum Constants

Enum classes in Kotlin have synthetic methods allowing to list the defined enum constants and to get an enum constant by its name. The signatures of these methods are as follows (assuming the name of the enum class is `EnumClass`):

```kotlin
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.

Since Kotlin 1.1, it's possible to access the constants in an enum class in a generic way, using the `enumValues<T>()` and `enumValueOf<T>()` functions:

```kotlin
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:

```kotlin
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.

Object Expressions and Declarations

Sometimes we 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

To create an object of an anonymous class that inherits from some type (or types), we write:

```
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. Many supertypes may be specified as a comma-separated 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
}
```

If, by any chance, we need "just an object", with no nontrivial supertypes, we can simply say:

```
fun foo() {
    val adHoc = object {
        var x: Int = 0
        var y: Int = 0
    }
    print(adHoc.x + adHoc.y)
}
```

Note that anonymous objects can be used as types only in local and private declarations. If you use an anonymous object as a return type of a public function or the type of a public property, the actual type of that function or property will be the declared supertype of the anonymous object, or `Any` if you didn't declare any supertype. Members added in the anonymous object will not be accessible.

```kotlin
class C {
    // Private function, so the return type is the anonymous object type
    private fun foo() = object {
        val x: String = "x"
    }

    // Public function, so the return type is Any
    fun publicFoo() = object {
        val x: String = "x"
    }

    fun bar() {
        val x1 = foo().x        // Works
        val x2 = publicFoo().x  // ERROR: Unresolved reference 'x'
    }
}
```

The code in object expressions can access variables from the enclosing scope.

```kotlin
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 may be useful in several cases, and Kotlin (after Scala) makes it easy to declare singletons:

```kotlin
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.

To refer to the object, we use its name directly:

```
DataProviderManager.registerDataProvider(...)
```

Such objects can have supertypes:

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ... }

    override fun mouseEntered(e: MouseEvent) { ... }
}
```

NOTE: 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
```

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:

```kotlin
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.

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]"
}
```

Inline classes

> Inline classes are available only since Kotlin 1.3 and currently are *experimental*. See details [below](#)

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`, which is declared by placing an `inline` modifier before the name of the class:

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

An inline class must have a single property initialized in the primary constructor. At runtime, instances of the inline class will be represented using this single property (see details about runtime representation [below](#)):

```
// No actual instantiation of class 'Password' happens
// At runtime 'securePassword' contains just 'String'
val securePassword = Password("Don't try this in production")
```

This is the main feature of inline classes, which inspired the name "inline": data of the class is "inlined" into its usages (similar to how content of [inline functions](#) is inlined to call sites).

Members

Inline classes support some functionality of regular classes. In particular, they are allowed to declare properties and functions:

```
inline class Name(val s: String) {
    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
}
```

However, there are some restrictions for inline class members:

— inline classes cannot have `init` blocks

— inline class properties cannot have [backing fields](#)

— it follows that inline classes can only have simple computable properties (no lateinit/delegated properties)

Inheritance

Inline classes are allowed to inherit from interfaces:

```kotlin
interface Printable {
    fun prettyPrint(): String
}

inline 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.

```kotlin
interface I

inline 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:

```kotlin
inline 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.

> Note that `-` is an *invalid* symbol in Java, meaning that it's impossible to call functions which accept inline classes from Java.

Inline classes vs type aliases

At first sight, inline classes may appear to be 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:

```kotlin
typealias NameTypeAlias = String
inline 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
}
```

Experimental status of inline classes

The design of inline classes is experimental, meaning that this feature is *moving fast* and no compatibility guarantees are given. When using inline classes in Kotlin 1.3+, a warning will be reported, indicating that this feature is experimental.

To remove the warning you have to opt in to the usage of this experimental feature by passing the compiler argument `-Xinline-classes`.

Enabling inline classes in Gradle

```
compileKotlin {
    kotlinOptions.freeCompilerArgs += ["-Xinline-classes"]
}
```

See Compiler options in Gradle for details. For Multiplatform Projects settings, see building Multiplatform Projects with Gradle section.

Enabling inline classes in Maven

```
<configuration>
    <args>
        <arg>-Xinline-classes</arg>
    </args>
</configuration>
```

See Compiler options in Maven for details.

Further discussion

See this language proposal for inline classes for other technical details and discussion.

Delegation

Property Delegation

Delegated properties are described on a separate page: Delegated Properties.

Implementation by 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:

```kotlin
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 might expect: the compiler will use your `override` implementations instead of those in the delegate object. If we were to add `override fun printMessage() { print("abc") }` to `Derived`, the program would print "abc" instead of "10" when `printMessage` is called:

```kotlin
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:

```kotlin
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)
}
```

Delegated Properties

There are certain common kinds of properties, that, though we can implement them manually every time we need them, would be very nice to implement once and for all, and put into 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 the *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 we 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 we read `p` from and the second parameter holds a description of `p` itself (e.g. you can take its name). For example:

```
val e = Example()
println(e.p)
```

This prints:

```
Example@33a17727, thank you for delegating 'p' to me!
```

Similarly, when we 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 since Kotlin 1.1 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 [the example](#).

Standard Delegates

The Kotlin standard library provides factory methods for several useful kinds of delegates.

Lazy

[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.

```kotlin
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

[Delegates.observable()](#) takes two arguments: the initial value and a handler for modifications. The handler is called every time we 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 has been performed.

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):

```
println(user.name) // Prints "John Doe"
println(user.age)  // Prints 25
```

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 (since 1.1)

You can declare local variables as delegated properties. For instance, 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 we summarize requirements to delegate objects.

For a **read-only** property (i.e. a `val`), a delegate has to provide a function named `getValue` that takes the following parameters:

— `thisRef` — must be the same or a supertype of the *property owner* (for extension properties — the type being extended);

— `property` — must be of type `KProperty<*>` or its supertype.

this function must return the same type as property (or its subtype).

For a **mutable** property (a `var`), a delegate has to *additionally* provide a function named `setValue` that takes the following parameters:

— `thisRef` — same as for `getValue()`;

— `property` — same as for `getValue()`;

— `new value` — must be of the same type as the property or its subtype.

`getValue()` and/or `setValue()` functions may 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.

The delegate class may implement one of the interfaces `ReadOnlyProperty` and `ReadWriteProperty` containing the required `operator` methods. These interfaces are declared in the Kotlin standard library:

```
interface ReadOnlyProperty<in R, out T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
}

interface ReadWriteProperty<in R, T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
    operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
}
```

Translation Rules

Under the hood for every delegated property the Kotlin compiler generates an auxiliary property and delegates to it. For instance, for the property `prop` the hidden property `prop$delegate` is generated, and the code of the accessors simply delegates to this additional property:

```kotlin
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.

Note that the syntax `this::prop` to refer a bound callable reference in the code directly is available only since Kotlin 1.1.

Providing a delegate (since 1.1)

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 property consistency when the property is created, not only in its getter or setter.

For example, if you want to check the property name before binding, you can write something like this:

```kotlin
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
    — 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:

```kotlin
// 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.

Functions and Lambdas

Functions

Function declarations

Functions in Kotlin are declared using the `fun` keyword:

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

Function usage

Calling functions uses the traditional approach:

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

Calling member functions uses the dot notation:

```kotlin
Stream().read() // create instance of class Stream and call read()
```

Parameters

Function parameters are defined using Pascal notation, i.e. *name*: *type*. Parameters are separated using commas. Each parameter must be explicitly typed:

```kotlin
fun powerOf(number: Int, exponent: Int) { /*...*/ }
```

Default arguments

Function parameters can have default values, which are used when a corresponding argument is omitted. This allows for a reduced number of overloads compared to other languages:

```kotlin
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size) { /*...*/ }
```

Default values are defined using the **=** after type along with the value.

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 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, it can be passed in 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

Function parameters can be named when calling functions. This is very convenient when a function has a high number of parameters or default ones.

Given the following function:

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = false,
             wordSeparator: Char = ' ') {
/*...*/
}
```

We could call this using default arguments:

```
reformat(str)
```

However, when calling it with non-default, the call would look something like:

```
reformat(str, true, true, false, '_')
```

With named arguments we can make the code much more readable:

```
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
```

and if we do not need all arguments:

```
reformat(str, wordSeparator = '_')
```

When a function is called with both positional and named arguments, all the positional arguments should be placed before the first named one. For example, the call `f(1, y = 2)` is allowed, but `f(x = 1, 2)` is not.

[Variable number of arguments (`vararg`)](#) can be passed in the named form by using the **spread** operator:

```
fun foo(vararg strings: String) { /*...*/ }

foo(strings = *arrayOf("a", "b", "c"))
```

> **On the JVM**: the named argument syntax cannot be used 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)

A parameter of a function (normally the last one) may be marked with `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
}
```

allowing a variable number of arguments to be passed 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 may 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 we call a `vararg`-function, we can pass arguments one-by-one, e.g. `asList(1, 2, 3)`, or, if we already have an array and want to pass its contents to the function, we 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 satisfy 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`- and `in`-checks, and some other operators. These expressions are equivalent as well:

— `a && b xor c` is equivalent to `a && (b xor c)`

— `a xor b in c` is equivalent to `(a xor b) in c`

See the [Grammar reference](#) for the complete operators precedence hierarchy.

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, you need to use `this` explicitly; unlike regular method calls, it cannot be omitted. 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

In Kotlin functions can be declared at 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 local, as member functions and extension functions.

Local functions

Kotlin supports local functions, i.e. a function inside another function:

```kotlin
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())
}
```

Local function can access local variables of outer functions (i.e. the closure), so in the case above, the *visited* can be a local variable:

```kotlin
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:

```kotlin
class Sample() {
    fun foo() { print("Foo") }
}
```

Member functions are called with dot notation:

```kotlin
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:

```kotlin
fun <T> singletonList(item: T): List<T> { /*...*/ }
```

For more information on generic functions see [Generics](#).

Inline functions

Inline functions are explained [here](#).

Extension functions

Extension functions are explained in [their own section](#).

Higher-order functions and lambdas

Higher-Order functions and Lambdas are explained in [their own section](#).

Tail recursive functions

Kotlin supports a style of functional programming known as [tail recursion](#). This allows some algorithms that would normally be written using loops to instead be written using a recursive function, but without the risk of stack overflow. When a function is marked with the `tailrec` modifier and meets the required form, the compiler optimises 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 any more, 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.

Higher-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) -> 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`, we 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:

```
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)
```

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, i.e. 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:

```
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
```

Note that 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:

```
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
```

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', i.e. 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, i.e. 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 we leave all the optional annotations out, what's left looks like this:

```
val sum = { x, y -> 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

We 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.toUpperCase() }
```

Underscore for unused variables (since 1.1)

If the lambda parameter is unused, you can place an underscore instead of its name:

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

Destructuring in lambdas (since 1.1)

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.

Note that 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*, i.e. 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](#):

```kotlin
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, i.e. 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. I.e., 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()
}
```

Isn't it what we wanted from the very beginning?

To make the compiler do this, we 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 we do it in a reasonable way (i.e. 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 we like: stored in fields, passed around etc.

Note that 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, we can only use a normal, unqualified `return` to exit a named function or an anonymous function. This means that to exit a lambda, we have to use a [label](#), and a bare `return` is forbidden inside a lambda, because a lambda cannot make the enclosing function return:

```
fun foo() {
    ordinaryFunction {
        return // ERROR: cannot make `foo` return here
    }
}
```

But if the function the lambda is passed to is inlined, the return can be inlined as well, so it is allowed:

inline fun inlined(block: () -> Unit) { println("hi!") }

```
fun foo() {
    inlined {
        return // OK: the lambda is inlined
    }
}
```

Such returns (located in a lambda, but exiting the enclosing function) are called *non-local* returns. We are used to 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 we need to access a type passed to us as a parameter:

```kotlin
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, we 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:

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

What we actually want is simply pass a type to this function, i.e. call it like this:

```kotlin
treeNode.findParentOfType<MyTreeNode>()
```

To enable this, inline functions support *reified type parameters*, so we can write something like this:

```kotlin
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

We qualified the type parameter with the `reified` modifier, now it's 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, we can call it as mentioned above: `myTree.findParentOfType<MyTreeNodeType>()`.

Though reflection may not be needed in many cases, we can still use it with a reified type parameter:

```kotlin
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 (e.g. a non-reified type parameter or a fictitious type like `Nothing`) cannot be used as an argument for a reified type parameter.

For a low-level description, see the spec document.

Inline properties (since 1.1)

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.

Collections

Kotlin 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 <u>var</u>: 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.

```
val numbers = mutableListOf("one", "two", "three", "four")
numbers.add("five")   // this is OK
//numbers = mutableListOf("six", "seven")     // compilation error
```

The read-only collection types are <u>covariant</u>. 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:

Let's walk through the interfaces and their implementations.

Collection

<u>Collection\<T></u> 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: <u>List</u> and <u>Set</u>.

```
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)
}
```

<u>MutableCollection</u> is a `Collection` with write operations, such as `add` and `remove`.

```kotlin
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)`.

```kotlin
val numbers = listOf("one", "two", "three", "four")
println("Number of elements: ${numbers.size}")
println("Third element: ${numbers.get(2)}")
println("Fourth element: ${numbers[3]}")
println("Index of element \"two\" ${numbers.indexOf("two")}")
```

List elements (including nulls) can duplicate: a list can contain any number of equal objects or occurrences of a single object. Two lists are considered equal if they have the same sizes and structurally equal elements at the same positions.

```kotlin
val bob = Person("Bob", 31)
val people = listOf<Person>(Person("Adam", 20), bob, bob)
val people2 = listOf<Person>(Person("Adam", 20), Person("Bob", 31), bob)
println(people == people2)
bob.age = 32
println(people == people2)
```

`MutableList` is a `List` with list-specific write operations, for example, to add or remove an element at a specific position.

```kotlin
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
numbers.removeAt(1)
numbers[0] = 0
numbers.shuffle()
println(numbers)
```

As you see, in some aspects lists are very similar to arrays. However, there is one important difference: an array's size is defined upon initialization and is never changed; in turn, a list doesn't have a predefined size; a list's size can be changed as a result of write operations: adding, updating, or removing elements.

In Kotlin, the default implementation of `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.

```
val numbers = setOf(1, 2, 3, 4)
println("Number of elements: ${numbers.size}")
if (numbers.contains(1)) println("1 is in the set")

val numbersBackwards = setOf(4, 3, 2, 1)
println("The sets are equal: ${numbers == numbersBackwards}")
```

[MutableSet](#) is a `Set` with write operations from `MutableCollection`.

The default implementation of `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.

```
val numbers = setOf(1, 2, 3, 4)  // LinkedHashSet is the default implementation
val numbersBackwards = setOf(4, 3, 2, 1)

println(numbers.first() == numbersBackwards.first())
println(numbers.first() == numbersBackwards.last())
```

An alternative implementation – [HashSet](#) – says nothing about the elements order, so calling such functions on it returns unpredictable results. However, `HashSet` requires less memory to store the same number of elements.

Map

[Map<K, V>](#) is not an inheritor of the `Collection` interface; however, it's a Kotlin collection type as well. A `Map` stores *key-value* pairs (or *entries*); keys are unique, but different keys can be paired with equal values. The `Map` interface provides specific functions, such as access to value by key, searching keys and values, and so on.

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)

println("All keys: ${numbersMap.keys}")
println("All values: ${numbersMap.values}")
if ("key2" in numbersMap) println("Value by key \"key2\": ${numbersMap["key2"]}")
if (1 in numbersMap.values) println("The value 1 is in the map")
if (numbersMap.containsValue(1)) println("The value 1 is in the map") // same as previous
```

Two maps containing the equal pairs are equal regardless of the pair order.

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
val anotherMap = mapOf("key2" to 2, "key1" to 1, "key4" to 1, "key3" to 3)

println("The maps are equal: ${numbersMap == anotherMap}")
```

MutableMap is a `Map` with map write operations, for example, you can add a new key-value pair or update the value associated with the given key.

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap.put("three", 3)
numbersMap["one"] = 11

println(numbersMap)
```

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

Constructing 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.

```
val doubled = List(3, { it * 2 })  // or MutableList if you want to change its content
later
println(doubled)
```

Concrete type constructors

To create a concrete type collection, such as an `ArrayList` or `LinkedList`, you can use the available constructors for these types. Similar constructors are available for implementations of `Set` and `Map`.

```
val linkedList = LinkedList<String>(listOf("one", "two", "three"))
val presizedSet = HashSet<Int>(32)
```

Copying

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.

```
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}")
```

These functions can also be used for converting collections to other types, for example, build a set from a list or vice versa.

```
val sourceList = mutableListOf(1, 2, 3)
val copySet = sourceList.toMutableSet()
copySet.add(3)
copySet.add(4)
println(copySet)
```

Alternatively, you can create new references to the same collection instance. New references are created when you initialize a collection variable with an existing collection. So, when the collection instance is altered through a reference, the changes are reflected in all its references.

```
val sourceList = mutableListOf(1, 2, 3)
val referenceList = sourceList
referenceList.add(4)
println("Source size: ${sourceList.size}")
```

Collection initialization can be used for restricting mutability. For example, if you create a `List` reference to a `MutableList`, the compiler will produce errors if you try to modify the collection through this reference.

```
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
```

Invoking 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:

```kotlin
val numbers = listOf("one", "two", "three", "four")
val longerThan3 = numbers.filter { it.length > 3 }
println(longerThan3)
```

Mapping produces a list of a transformation results:

```kotlin
val numbers = setOf(1, 2, 3)
println(numbers.map { it * 3 })
println(numbers.mapIndexed { idx, value -> value * idx })
```

Association produces maps:

```kotlin
val numbers = listOf("one", "two", "three", "four")
println(numbers.associateWith { it.length })
```

For more information about operations on collections in Kotlin, see Collection Operations Overview.

Iterators

For traversing collection elements, the Kotlin standard library supports the commonly used mechanism of *iterators* – objects that provide access to the elements sequentially without exposing the underlying structure of the collection. Iterators are useful when you need to process all the elements of a collection one-by-one, for example, print values or make similar updates to them.

Iterators can be obtained for inheritors of the `Iterable<T>` interface, including `Set` and `List`, by calling the `iterator()` function. Once you obtain an iterator, it points to the first element of a collection; calling the `next()` function returns this element and moves the iterator position to the following element if it exists. Once the iterator passes through the last element, it can no longer be used for retrieving elements; neither can it be reset to any previous position. To iterate through the collection again, create a new iterator.

```
val numbers = listOf("one", "two", "three", "four")
val numbersIterator = numbers.iterator()
while (numbersIterator.hasNext()) {
    println(numbersIterator.next())
}
```

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:

```
val numbers = listOf("one", "two", "three", "four")
for (item in numbers) {
    println(item)
}
```

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:

```
val numbers = listOf("one", "two", "three", "four")
numbers.forEach {
    println(it)
}
```

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()`.

```kotlin
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()}")
}
```

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.

```kotlin
val numbers = mutableListOf("one", "two", "three", "four")
val mutableIterator = numbers.iterator()

mutableIterator.next()
mutableIterator.remove()
println("After removal: $numbers")
```

In addition to removing elements, the MutableListIterator can also insert and replace elements while iterating the list.

```kotlin
val numbers = mutableListOf("one", "four", "four")
val mutableListIterator = numbers.listIterator()

mutableListIterator.next()
mutableListIterator.add("two")
mutableListIterator.next()
mutableListIterator.set("three")
println(numbers)
```

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 the `for` loops.

```
for (i in 1..4) print(i)
```

To iterate numbers in reverse order, use the downTo function instead of `..`.

```
for (i in 4 downTo 1) print(i)
```

It is also possible to iterate over numbers with an arbitrary step (not necessarily 1). This is done via the `step` function.

```
for (i in 1..8 step 2) print(i)
println()
for (i in 8 downTo 1 step 2) print(i)
```

To iterate a number range which does not include its end element, use the `until` function:

```
for (i in 1 until 10) {      // i in [1, 10), 10 is excluded
    print(i)
}
```

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 `..`.

```
val versionRange = Version(1, 11)..Version(1, 30)
println(Version(0, 9) in versionRange)
println(Version(1, 20) in versionRange)
```

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`. The last element is always hit by iteration unless the progression is empty. 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.

```
for (i in 1..10) print(i)
```

To define a custom progression step, use the `step` function on a range.

```
for (i in 1..8 step 2) print(i)
```

The last element of the progression is calculated to find the maximum value not greater than the end value for a positive step or the minimum value not less than the end value for a negative step such that `(last - first) % step == 0`.

To create a progression for iterating in reverse order, use `downTo` instead of `..` when defining the range for it.

```
for (i in 4 downTo 1) print(i)
```

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.

```
println((1..10).filter { it % 2 == 0 })
```

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.

Constructing

From elements

To create a sequence, call the `sequenceOf()` function listing the elements as its arguments.

```
val numbersSequence = sequenceOf("four", "three", "two", "one")
```

From `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 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.

```
val oddNumbers = generateSequence(1) { it + 2 } // `it` is the previous element
println(oddNumbers.take(5).toList())
//println(oddNumbers.count())      // error: the sequence is infinite
```

To create a finite sequence with `generateSequence()`, provide a function that returns `null` after the last element you need.

```
val oddNumbersLessThan10 = generateSequence(1) { if (it < 10) it + 2 else null }
println(oddNumbersLessThan10.count())
```

From chunks

Finally, there is a function that lets you produce sequence elements one by one or by chunks of arbitrary sizes – the `sequence()` function. This function takes a lambda expression containing calls of `yield()` and `yieldAll()` functions. They return an element to the sequence consumer and suspend the execution of `sequence()` until the next element is requested by the consumer. `yield()` takes a single element as an argument; `yieldAll()` can take an `Iterable` object, an `Iterator`, or another `Sequence`. A `Sequence` argument of `yieldAll()` can be infinite. However, such a call must be the last: all subsequent calls will never be executed.

```
val oddNumbers = sequence {
    yield(1)
    yieldAll(listOf(3, 5))
    yieldAll(generateSequence(7) { it + 2 })
}
println(oddNumbers.take(5).toList())
```

Sequence operations

The sequence operations can be classified into the following groups regarding their state requirements:

— *Stateless* operations require no state and process each element independently, for example, `map()` or `filter()`. Stateless operations can also require a small constant amount of state to process an element, for example, `take() or drop()`.

— *Stateful* operations require a significant amount of state, usually proportional to the number of elements in a sequence.

If a sequence operation returns another sequence, which is produced lazily, it's called *intermediate*. Otherwise, the operation is *terminal*. Examples of terminal operations are `toList()` or `sum()`. Sequence elements can be retrieved only with terminal operations.

Sequences can be iterated multiple times; however some sequence implementations might constrain themselves to be iterated only once. That is mentioned specifically in their documentation.

Sequence processing example

Let's take a look at the difference between `Iterable` and `Sequence` with an example.

Iterable

Assume that you have a list of words. The code below filters the words longer than three characters and prints the lengths of first four such words.

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
val lengthsList = words.filter { println("filter: $it"); it.length > 3 }
    .map { println("length: ${it.length}"); it.length }
    .take(4)

println("Lengths of first 4 words longer than 3 chars:")
println(lengthsList)
```

When you run this code, you'll see that the `filter()` and `map()` functions are executed in the same order as they appear in the code. First, you see `filter:` for all elements, then `length:` for the elements left after filtering, and then the output of the two last lines. This is how the list processing goes:

Sequence

Now let's write the same with sequences:

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
//convert the List to a Sequence
val wordsSequence = words.asSequence()

val lengthsSequence = wordsSequence.filter { println("filter: $it"); it.length > 3 }
    .map { println("length: ${it.length}"); it.length }
    .take(4)

println("Lengths of first 4 words longer than 3 chars")
// terminal operation: obtaining the result as a List
println(lengthsSequence.toList())
```

The output of this code shows that the `filter()` and `map()` functions are called only when building the result list. So, you first see the line of text "Lengths of.." and then the sequence processing starts. Note that for elements left after filtering, the map executes before filtering the next element. When the result size reaches 4, the processing stops because it's the largest possible size that `take(4)` can return.

The sequence processing goes like this:

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 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.

```kotlin
val numbers = listOf("one", "two", "three", "four")
numbers.filter { it.length > 3 }  // nothing happens with `numbers`, result is lost
println("numbers are still $numbers")
val longerThan3 = numbers.filter { it.length > 3 } // result is stored in `longerThan3`
println("numbers longer than 3 chars are $longerThan3")
```

For certain collection operations, there is an option to specify the *destination* object. Destination is a mutable collection to which the function appends its resulting items instead of returning them in a new object. For performing operations with destinations, there are separate functions with the `To` postfix in their names, for example, <u>filterTo()</u> instead of <u>filter()</u> or <u>associateTo()</u> instead of <u>associate()</u>. These functions take the destination collection as an additional parameter.

```kotlin
val numbers = listOf("one", "two", "three", "four")
val filterResults = mutableListOf<String>()  //destination object
numbers.filterTo(filterResults) { it.length > 3 }
numbers.filterIndexedTo(filterResults) { index, _ -> index == 0 }
println(filterResults) // contains results of both operations
```

For convenience, these functions return the destination collection back, so you can create it right in the corresponding argument of the function call:

```kotlin
// filter numbers right into a new hash set,
// thus eliminating duplicates in the result
val result = numbers.mapTo(HashSet()) { it.length }
println("distinct item lengths are $result")
```

Functions with destination are available for filtering, association, grouping, flattening, and other operations. For the complete list of destination operations see the <u>Kotlin collections reference</u>.

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 <u>Write operations</u> and corresponding sections of <u>List specific operations</u> and <u>Map specific operations</u>.

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, <u>sort()</u> sorts a mutable collection in-place, so it's state changes; <u>sorted()</u> creates a new collection that contains the same elements in the sorted order.

```kotlin
val numbers = mutableListOf("one", "two", "three", "four")
val sortedNumbers = numbers.sorted()
println(numbers == sortedNumbers)  // false
numbers.sort()
println(numbers == sortedNumbers)  // true
```

Collection Transformations

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.

Mapping

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()`.

```
val numbers = setOf(1, 2, 3)
println(numbers.map { it * 3 })
println(numbers.mapIndexed { idx, value -> value * idx })
```

If the transformation produces `null` on certain elements, you can filter out the `null`s from the result collection by calling the `mapNotNull()` function instead of `map()`, or `mapIndexedNotNull()` instead of `mapIndexed()`.

```
val numbers = setOf(1, 2, 3)
println(numbers.mapNotNull { if ( it == 2) null else it * 3 })
println(numbers.mapIndexedNotNull { idx, value -> if (idx == 0) null else value * idx })
```

When transforming maps, you have two options: transform keys leaving values unchanged and vice versa. To apply a given transformation to keys, use `mapKeys()`; in turn, `mapValues()` transforms values. Both functions use the transformations that take a map entry as an argument, so you can operate both its key and value.

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
println(numbersMap.mapKeys { it.key.toUpperCase() })
println(numbersMap.mapValues { it.value + it.key.length })
```

Zipping

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`.

```
val colors = listOf("red", "brown", "grey")
val animals = listOf("fox", "bear", "wolf")
println(colors zip animals)

val twoAnimals = listOf("fox", "bear")
println(colors.zip(twoAnimals))
```

You can also call `zip()` with a transformation function that takes two parameters: the receiver element and the argument element. In this case, the result `List` contains the return values of the transformation function called on pairs of the receiver and the argument elements with the same positions.

```
val colors = listOf("red", "brown", "grey")
val animals = listOf("fox", "bear", "wolf")

println(colors.zip(animals) { color, animal -> "The ${animal.capitalize()} is $color"})
```

When you have a `List` of `Pair`s, you can do the reverse transformation – *unzipping* – that builds two lists from these pairs:

— The first list contains the first elements of each `Pair` in the original list.

— The second list contains the second elements.

To unzip a list of pairs, call [unzip()](#).

```
val numberPairs = listOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)
println(numberPairs.unzip())
```

Association

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.

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.associateWith { it.length })
```

For building maps with collection elements as values, there is the function [associateBy()](#). It takes a function that returns a key based on an element's value. If two elements are equal, only the last one remains in the map. `associateBy()` can also be called with a value transformation function.

```
val numbers = listOf("one", "two", "three", "four")

println(numbers.associateBy { it.first().toUpperCase() })
println(numbers.associateBy(keySelector = { it.first().toUpperCase() }, valueTransform =
{ it.length }))
```

Another way to build maps in which both keys and values are somehow produced from collection elements is the function associate(). It takes a lambda function that returns a `Pair`: the key and the value of the corresponding map entry.

Note that `associate()` produces short-living `Pair` objects which may affect the performance. Thus, `associate()` should be used when the performance isn't critical or it's more preferable than other options.

An example of the latter is when a key and the corresponding value are produced from an element together.

```
val names = listOf("Alice Adams", "Brian Brown", "Clara Campbell")
println(names.associate { name -> parseFullName(name).let { it.lastName to it.firstName }
})
```

Here we call a transform function on an element first, and then build a pair from the properties of that function's result.

Flattening

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.

```
val numberSets = listOf(setOf(1, 2, 3), setOf(4, 5, 6), setOf(1, 2))
println(numberSets.flatten())
```

Another function – flatMap() provides a flexible way to process nested collections. It takes a function that maps a collection element to another collection. As a result, `flatMap()` returns a single list of its return values on all the elements. So, `flatMap()` behaves as a subsequent call of `map()` (with a collection as a mapping result) and `flatten()`.

```
val containers = listOf(
    StringContainer(listOf("one", "two", "three")),
    StringContainer(listOf("four", "five", "six")),
    StringContainer(listOf("seven", "eight"))
)
println(containers.flatMap { it.values })
```

String representation

If you need to retrieve the collection content in a readable format, use functions that transform the collections to strings: joinToString() and joinTo().

joinToString() builds a single String from the collection elements based on the provided arguments. joinTo() does the same but appends the result to the given Appendable object.

When called with the 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.

```
val numbers = listOf("one", "two", "three", "four")

println(numbers)
println(numbers.joinToString())

val listString = StringBuffer("The list of numbers: ")
numbers.joinTo(listString)
println(listString)
```

To build a custom string representation, you can specify its parameters in function arguments separator, prefix, and postfix. The resulting string will start with the prefix and end with the postfix. The separator will come after each element except the last.

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.joinToString(separator = " | ", prefix = "start: ", postfix = ": end"))
```

For bigger collections, you may want to specify the limit – a number of elements that will be included into result. If the collection size exceeds the limit, all the other elements will be replaced with a single value of the truncated argument.

```
val numbers = (1..100).toList()
println(numbers.joinToString(limit = 10, truncated = "<...>"))
```

Finally, to customize the representation of elements themselves, provide the transform function.

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.joinToString { "Element: ${it.toUpperCase()}"})
```

Filtering

Filtering is one of the most popular tasks in the 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.

Filtering 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.

```
val numbers = listOf("one", "two", "three", "four")
val longerThan3 = numbers.filter { it.length > 3 }
println(longerThan3)

val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
println(filteredMap)
```

The predicates in `filter()` can only check the values of the elements. If you want to use element positions in the filter, use `filterIndexed()`. It takes a predicate with two arguments: the index and the value of an element.

To filter collections by negative conditions, use `filterNot()`. It returns a list of elements for which the predicate yields `false`.

```
val numbers = listOf("one", "two", "three", "four")

val filteredIdx = numbers.filterIndexed { index, s -> (index != 0) && (s.length < 5)  }
val filteredNot = numbers.filterNot { it.length <= 3 }

println(filteredIdx)
println(filteredNot)
```

There are also functions that narrow the element type by filtering elements of a given type:

— `filterIsInstance()` returns collection elements of a given type. Being called on a
   `List<Any>`, `filterIsInstance<T>()` returns a `List<T>`, thus allowing you to call
   functions of the `T` type on its items.

```
val numbers = listOf(null, 1, "two", 3.0, "four")
println("All String elements in upper case:")
numbers.filterIsInstance<String>().forEach {
    println(it.toUpperCase())
}
```

— 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.

```
val numbers = listOf(null, "one", "two", null)
numbers.filterNotNull().forEach {
    println(it.length)   // length is unavailable for nullable Strings
}
```

Partitioning

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.

```
val numbers = listOf("one", "two", "three", "four")
val (match, rest) = numbers.partition { it.length > 3 }

println(match)
println(rest)
```

Testing 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*.

```
val numbers = listOf("one", "two", "three", "four")

println(numbers.any { it.endsWith("e") })
println(numbers.none { it.endsWith("a") })
println(numbers.all { it.endsWith("e") })

println(emptyList<Int>().all { it > 5 })   // vacuous truth
```

`any()` and `none()` can also be used without a predicate: in this case they just check the
collection emptiness. `any()` returns `true` if there are elements and `false` if there aren't;
`none()` does the opposite.

```kotlin
val numbers = listOf("one", "two", "three", "four")
val empty = emptyList<String>()

println(numbers.any())
println(empty.any())

println(numbers.none())
println(empty.none())
```

`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.

```
val numbers = listOf("one", "two", "three", "four")

val plusList = numbers + "five"
val minusList = numbers - listOf("three", "four")
println(plusList)
println(minusList)
```

For the details on `plus` and `minus` operators for maps, see Map Specific Operations. The augmented assignment operators plusAssign (+=) and minusAssign (-=) are also defined for collections. However, for read-only collections, they actually use the `plus` or `minus` operators and try to assign the result to the same variable. Thus, they are available only on `var` read-only collections. For mutable collections, they modify the collection if it's a `val`. For more details see Collection Write Operations.

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.

```
val numbers = listOf("one", "two", "three", "four", "five")

println(numbers.groupBy { it.first().toUpperCase() })
println(numbers.groupBy(keySelector = { it.first() }, valueTransform = { it.toUpperCase() }))
```

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.

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.groupingBy { it.first() }.eachCount())
```

Retrieving 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.

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.slice(1..3))
println(numbers.slice(0..4 step 2))
println(numbers.slice(setOf(3, 5, 0)))
```

Take and drop

To get the specified number of elements starting from the first, use the take() function. For getting the last elements, use takeLast(). When called with a number larger than the collection size, both functions return the whole collection.

To take all the elements except a given number of first or last elements, call the drop() and dropLast() functions respectively.

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.take(3))
println(numbers.takeLast(3))
println(numbers.drop(1))
println(numbers.dropLast(5))
```

You can also use predicates to define the number of elements for taking or dropping. There are four functions similar to the ones described above:

— takeWhile() is `take()` with a predicate: it takes the elements up to but excluding the first one not matching the predicate. If the first collection element doesn't match the predicate, the result is empty.

— takeLastWhile() is similar to `takeLast()`: it takes the range of elements matching the predicate from the end of the collection. The first element of the range is the element next to the last element not matching the predicate. If the last collection element doesn't match the predicate, the result is empty;

— dropWhile() is the opposite to `takeWhile()` with the same predicate: it returns the elements from the first one not matching the predicate to the end.

— dropLastWhile() is the opposite to `takeLastWhile()` with the same predicate: it returns the elements from the beginning to the last one not matching the predicate.

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.takeWhile { !it.startsWith('f') })
println(numbers.takeLastWhile { it != "three" })
println(numbers.dropWhile { it.length == 3 })
println(numbers.dropLastWhile { it.contains('i') })
```

Chunked

To break a collection 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.

```
val numbers = (0..13).toList()
println(numbers.chunked(3))
```

You can also apply a transformation for the returned chunks right away. To do this, provide the transformation as a lambda function when calling `chunked()`. The lambda argument is a chunk of the collection. When `chunked()` is called with a transformation, the chunks are short-living `List`s that should be consumed right in that lambda.

```
val numbers = (0..13).toList()
println(numbers.chunked(3) { it.sum() })  // `it` is a chunk of the original collection
```

Windowed

You can retrieve all possible ranges of the collection elements of a given size. The function for getting them is called [windowed()](#): it returns a list of element ranges that you would see if you were looking at the collection through a sliding window of the given size. Unlike `chunked()`, `windowed()` returns element ranges (*windows*) starting from *each* collection element. All the windows are returned as elements of a single `List`.

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.windowed(3))
```

`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, an 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()`.

```
val numbers = (1..10).toList()
println(numbers.windowed(3, step = 2, partialWindows = true))
println(numbers.windowed(3) { it.sum() })
```

To build two-element windows, there is a separate function - `zipWithNext()`. It creates pairs of adjacent elements of the receiver collection. Note that `zipWithNext()` doesn't break the collection into pairs; it creates a `Pair` for *each* element except the last one, so its result on `[1, 2, 3, 4]` is `[[1, 2], [2, 3], [3, 4]]`, not `[[1, 2], [3, 4]]`. `zipWithNext()` can be called with a transformation function as well; it should take two elements of the receiver collection as arguments.

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.zipWithNext())
println(numbers.zipWithNext() { s1, s2 -> s1.length > s2.length})
```

Retrieving 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.

Retrieving 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 `[]`).

```
val numbers = linkedSetOf("one", "two", "three", "four", "five")
println(numbers.elementAt(3))

val numbersSortedSet = sortedSetOf("one", "two", "three", "four")
println(numbersSortedSet.elementAt(0)) // elements are stored in the ascending order
```

There are also useful aliases for retrieving the first and the last element of the collection: `first()` and `last()`.

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.first())
println(numbers.last())
```

To avoid exceptions when retrieving element with non-existing positions, use safe variations of `elementAt()`:

— `elementAtOrNull()` returns null when the specified position is out of the collection bounds.

— `elementAtOrElse()` additionally takes a lambda function that maps an `Int` argument to an instance of the collection element type. When called with an out-of-bounds position, the `elementAtOrElse()` returns the result of the lambda on the given value.

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.elementAtOrNull(5))
println(numbers.elementAtOrElse(5) { index -> "The value for index $index is undefined"})
```

Retrieving 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.

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.first { it.length > 3 })
println(numbers.last { it.startsWith("f") })
```

If no elements match the predicate, both functions throw exceptions. To avoid them, use `firstOrNull()` and `lastOrNull()` instead: they return `null` if no matching elements are found.

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.firstOrNull { it.length > 6 })
```

Alternatively, you can use the aliases if their names suit your situation better:

— `find()` instead of `firstOrNull()`

— `findLast()` instead of `lastOrNull()`

```
val numbers = listOf(1, 2, 3, 4)
println(numbers.find { it % 2 == 0 })
println(numbers.findLast { it % 2 == 0 })
```

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.

```
val numbers = listOf(1, 2, 3, 4)
println(numbers.random())
```

Checking 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.

```kotlin
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.contains("four"))
println("zero" in numbers)

println(numbers.containsAll(listOf("four", "two")))
println(numbers.containsAll(listOf("one", "zero")))
```

Additionally, you can check if the collection contains any elements by calling isEmpty() or isNotEmpty().

```kotlin
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.isEmpty())
println(numbers.isNotEmpty())

val empty = emptyList<String>()
println(empty.isEmpty())
println(empty.isNotEmpty())
```

Collection 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`, an 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 that can be used for ordering versions that consist of the major and the minor part.

```kotlin
class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int {
        if (this.major != other.major) {
            return this.major - other.major
        } 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.

```
val lengthComparator = Comparator { str1: String, str2: String -> str1.length -
str2.length }
println(listOf("aaa", "bb", "c").sortedWith(lengthComparator))
```

Having the `lengthComparator`, you are able to arrange strings by their length instead of the default lexicographical order.

A shorter way to define a `Comparator` is the [compareBy()](#) function from the standard library. `compareBy()` takes a lambda function that produces a `Comparable` value from an instance and defines the custom order as the natural order of the produced values. With `compareBy()`, the length comparator from the example above looks like this:

```
tln(listOf("aaa", "bb", "c").sortedWith(compareBy { it.length }))
```

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.

```
val numbers = listOf("one", "two", "three", "four")

println("Sorted ascending: ${numbers.sorted()}")
println("Sorted descending: ${numbers.sortedDescending()}")
```

Custom orders

For sorting in custom orders or sorting non-comparable objects, there are the functions [sortedBy()](#) and [sortedByDescending()](#). They take a selector function that maps collection elements to `Comparable` values and sort the collection in natural order of that values.

```
val numbers = listOf("one", "two", "three", "four")

val sortedNumbers = numbers.sortedBy { it.length }
println("Sorted by length ascending: $sortedNumbers")
val sortedByLast = numbers.sortedByDescending { it.last() }
println("Sorted by the last letter descending: $sortedByLast")
```

To define a custom order for the collection sorting, you can provide your own `Comparator`. To do this, call the sortedWith() function passing in your `Comparator`. With this function, sorting strings by their length looks like this:

```
val numbers = listOf("one", "two", "three", "four")
println("Sorted by length ascending: ${numbers.sortedWith(compareBy { it.length })}")
```

Reverse order

You can retrieve the collection in the reversed order using the reversed() function.

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.reversed())
```

`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.

```
val numbers = listOf("one", "two", "three", "four")
val reversedNumbers = numbers.asReversed()
println(reversedNumbers)
```

If the original list is mutable, all its changes reflect in its reversed views and vice versa.

```
val numbers = mutableListOf("one", "two", "three", "four")
val reversedNumbers = numbers.asReversed()
println(reversedNumbers)
numbers.add("five")
println(reversedNumbers)
```

However, if the mutability of the list is unknown or the source is not a list at all, `reversed()` is more preferable since its result is a copy that won't change in the future.

Random order

Finally, there is a function that returns a new `List` containing the collection elements in a random order - shuffled(). You can call it without arguments or with a `Random` object.

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.shuffled())
```

Collection 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:

— `min()` and `max()` return the smallest and the largest element respectively;

— `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;

```kotlin
val numbers = listOf(6, 42, 10, 4)

println("Count: ${numbers.count()}")
println("Max: ${numbers.max()}")
println("Min: ${numbers.min()}")
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`:

— `maxBy()`/`minBy()` take a selector function and return the element for which it returns the largest or the smallest value.

— `maxWith()`/`minWith()` take a `Comparator` object and return the largest or smallest element according to that `Comparator`.

```kotlin
val numbers = listOf(5, 42, 10, 4)
val min3Remainder = numbers.minBy { it % 3 }
println(min3Remainder)

val strings = listOf("one", "two", "three", "four")
val longestString = strings.maxWith(compareBy { it.length })
println(longestString)
```

Additionally, there are advanced summation functions that take a function and return the sum of its return values on all elements:

— `sumBy()` applies functions that return `Int` values on collection elements.

— `sumByDouble()` works with functions that return `Double`.

```kotlin
val numbers = listOf(5, 42, 10, 4)
println(numbers.sumBy { it * 2 })
println(numbers.sumByDouble { it.toDouble() / 2 })
```

Fold and reduce

For more specific cases, there are the functions reduce() and fold() that apply the provided operation to the collection elements sequentially and return the accumulated result. The operation takes two arguments: the previously accumulated value and the collection element.

The difference between the two functions is that `fold()` takes an initial value and uses it as the accumulated value on the first step, whereas the first step of `reduce()` uses the first and the second elements as operation arguments on the first step.

```
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)
```

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.

```
val numbers = listOf(5, 2, 10, 4)
val sumDoubledRight = numbers.foldRight(0) { element, sum -> sum + element * 2 }
println(sumDoubledRight)
```

You can also apply operations that take element indices as parameters. For this purpose, use functions reduceIndexed() and foldIndexed() passing element index as the first argument of the operation.

Finally, there are functions that apply such operations to collection elements from right to left - reduceRightIndexed() and foldRightIndexed().

```
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)
```

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.

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
println(numbers)
```

`addAll()` adds every element of the argument object to a list or a set. The argument can be an `Iterable`, a `Sequence`, or an `Array`. The types of the receiver and the argument may be different, for example, you can add all items from a `Set` to a `List`.

When called on lists, `addAll()` adds new elements in the same order as they go in the argument. You can also call `addAll()` specifying an element position as the first argument. The first element of the argument collection will be inserted at this position. Other elements of the argument collection will follow it, shifting the receiver elements to the end.

```
val numbers = mutableListOf(1, 2, 5, 6)
numbers.addAll(arrayOf(7, 8))
println(numbers)
numbers.addAll(2, setOf(3, 4))
println(numbers)
```

You can also add elements using the in-place version of the plus operator - plusAssign (`+=`) When applied to a mutable collection, `+=` appends the second operand (an element or another collection) to the end of the collection.

```
val numbers = mutableListOf("one", "two")
numbers += "three"
println(numbers)
numbers += listOf("four", "five")
println(numbers)
```

Removing elements

To remove an element from a mutable collection, use the `remove()` function. `remove()` accepts the element value and removes one occurrence of this value.

```
val numbers = mutableListOf(1, 2, 3, 4, 3)
numbers.remove(3)                    // removes the first `3`
println(numbers)
numbers.remove(5)                    // removes nothing
println(numbers)
```

For removing multiple elements at once, there are the following functions :

— `removeAll()` removes all elements that are present in the argument collection. Alternatively, you can call it with a predicate as an argument; in this case the function removes all elements for which the predicate yields `true` .

— `retainAll()` is the opposite of `removeAll()` : it removes all elements except the ones from the argument collection. When used with a predicate, it leaves only elements that match it.

— `clear()` removes all elements from a list and leaves it empty.

```
val numbers = mutableListOf(1, 2, 3, 4)
println(numbers)
numbers.retainAll { it >= 3 }
println(numbers)
numbers.clear()
println(numbers)

val numbersSet = mutableSetOf("one", "two", "three", "four")
numbersSet.removeAll(setOf("one", "two"))
println(numbersSet)
```

Another way to remove elements from a collection is with the `minusAssign` (`-=`) operator – the in-place version of `minus`. The second argument can be a single instance of the element type or another collection. With a single element on the right-hand side, `-=` removes the *first* occurrence of it. In turn, if it's a collection, *all* occurrences of its elements are removed. For example, if a list contains duplicate elements, they are removed at once. The second operand can contain elements that are not present in the collection. Such elements don't affect the operation execution.

```
val numbers = mutableListOf("one", "two", "three", "three", "four")
numbers -= "three"
println(numbers)
numbers -= listOf("four", "five")
//numbers -= listOf("four")    // does the same as above
println(numbers)
```

Updating elements

Lists and sets 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.

Retrieving elements by index

Lists support all common operations for element retrieval: `elementAt()`, `first()`, `last()`, and others listed in Retrieving Single Elements. What is specific for lists is index access to the elements, so the simplest way to read an element is retrieving it by index. That is done with the get() function with the index passed in the argument or the shorthand `[index]` syntax.

If the list size is less than the specified index, an exception is thrown. There are two other functions that help you avoid such exceptions:

— getOrElse() lets you provide the function for calculating the default value to return if the index isn't present in the collection.

— getOrNull() returns `null` as the default value.

```
val numbers = listOf(1, 2, 3, 4)
println(numbers.get(0))
println(numbers[0])
//numbers.get(5)                        // exception!
println(numbers.getOrNull(5))           // null
println(numbers.getOrElse(5, {it}))      // 5
```

Retrieving 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.

```
val numbers = (0..13).toList()
println(numbers.subList(3, 6))
```

Finding 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`.

```
val numbers = listOf(1, 2, 3, 4, 2, 5)
println(numbers.indexOf(2))
println(numbers.lastIndexOf(2))
```

There is also a pair of functions that take a predicate and search for elements matching it:

- indexOfFirst() returns the *index of the first* element matching the predicate or `-1` if there are no such elements.

- indexOfLast() returns the *index of the last* element matching the predicate or `-1` if there are no such elements.

```
val numbers = mutableListOf(1, 2, 3, 4)
println(numbers.indexOfFirst { it > 2})
println(numbers.indexOfLast { it % 2 == 1})
```

Binary search in sorted lists

There is one more way to search elements in lists – binary search. It works significantly faster than other built-in search functions but *requires the list to be sorted* in ascending order according to a certain ordering: natural or another one provided in the function parameter. Otherwise, the result is undefined.

To search an element in a sorted list, call the binarySearch() function passing the value as an argument. If such an element exists, the function returns its index; otherwise, it returns `(-insertionPoint - 1)` where `insertionPoint` is the index where this element should be inserted so that the list remains sorted. If there is more than one element with the given value, the search can return any of their indices.

You can also specify an index range to search in: in this case, the function searches only between two provided indices.

```
val numbers = mutableListOf("one", "two", "three", "four")
numbers.sort()
println(numbers)
println(numbers.binarySearch("two"))  // 3
println(numbers.binarySearch("z")) // -5
println(numbers.binarySearch("two", 0, 2))  // -3
```

Comparator binary search

When list elements aren't `Comparable`, you should provide a Comparator to use in the binary search. The list must be sorted in ascending order according to this `Comparator`. Let's have a look at an example:

```
val productList = listOf(
    Product("WebStorm", 49.0),
    Product("AppCode", 99.0),
    Product("DotTrace", 129.0),
    Product("ReSharper", 149.0))

println(productList.binarySearch(Product("AppCode", 99.0), compareBy<Product> { it.price
}.thenBy { it.name }))
```

Here's a list of `Product` instances that aren't `Comparable` and a `Comparator` that defines the order: product `p1` precedes product `p2` if `p1`'s price is less than `p2`'s price. So, having a list sorted ascending according to this order, we use `binarySearch()` to find the index of the specified `Product`.

Custom comparators are also handy when a list uses an order different from natural one, for example, a case-insensitive order for `String` elements.

```
val colors = listOf("Blue", "green", "ORANGE", "Red", "yellow")
println(colors.binarySearch("RED", String.CASE_INSENSITIVE_ORDER)) // 3
```

Comparison binary search

Binary search with *comparison* function lets you find elements without providing explicit search values. Instead, it takes a comparison function mapping elements to `Int` values and searches for the element where the function returns zero. The list must be sorted in the ascending order according to the provided function; in other words, the return values of comparison must grow from one list element to the next one.

```
data class Product(val name: String, val price: Double)

fun priceComparison(product: Product, price: Double) = sign(product.price -
price).toInt()

fun main() {
    val productList = listOf(
        Product("WebStorm", 49.0),
        Product("AppCode", 99.0),
        Product("DotTrace", 129.0),
        Product("ReSharper", 149.0))

    println(productList.binarySearch { priceComparison(it, 99.0) })
}
```

Both comparator and comparison binary search can be performed for list ranges as well.

List write operations

In addition to the collection modification operations described in Collection Write Operations, mutable lists support specific write operations. Such operations use the index to access elements to broaden the list modification capabilities.

Adding

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.

```
val numbers = mutableListOf("one", "five", "six")
numbers.add(1, "two")
numbers.addAll(2, listOf("three", "four"))
println(numbers)
```

Updating

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.

```
val numbers = mutableListOf("one", "five", "three")
numbers[1] =  "two"
println(numbers)
```

`fill()` simply replaces all the collection elements with the specified value.

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.fill(3)
println(numbers)
```

Removing

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.

```
val numbers = mutableListOf(1, 2, 3, 4, 3)
numbers.removeAt(1)
println(numbers)
```

Sorting

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:

```
val numbers = mutableListOf("one", "two", "three", "four")

numbers.sort()
println("Sort into ascending: $numbers")
numbers.sortDescending()
println("Sort into descending: $numbers")

numbers.sortBy { it.length }
println("Sort into ascending by length: $numbers")
numbers.sortByDescending { it.last() }
println("Sort into descending by the last letter: $numbers")

numbers.sortWith(compareBy<String> { it.length }.thenBy { it })
println("Sort by Comparator: $numbers")

numbers.shuffle()
println("Shuffle: $numbers")

numbers.reverse()
println("Reverse: $numbers")
```

Set Specific Operations

The Kotlin collections package contains extension functions for popular operations on sets: finding intersections, merging, or subtracting collections from each other.

To merge two collections into one, use the union() function. It can be used in the infix form `a union b`. Note that for ordered collections the order of the operands is important: in the resulting collection, the elements of the first operand go before the elements of the second.

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`.

```
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
```

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.

Retrieving 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.

```
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!
```

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.

```
val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
println(numbersMap.keys)
println(numbersMap.values)
```

Filtering

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.

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
println(filteredMap)
```

There are also two specific ways for filtering maps: by keys and by values. For each way, there is a function: `filterKeys()` and `filterValues()`. Both return a new map of entries which match the given predicate. The predicate for `filterKeys()` checks only the element keys, the one for `filterValues()` checks only values.

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
val filteredKeysMap = numbersMap.filterKeys { it.endsWith("1") }
val filteredValuesMap = numbersMap.filterValues { it < 10 }

println(filteredKeysMap)
println(filteredValuesMap)
```

`plus` and `minus` operators

Due to the key access to elements, `plus` (`+`) and `minus` (`-`) operators work for maps differently than for other collections. `plus` returns a `Map` that contains elements of its both operands: a `Map` on the left and a `Pair` or another `Map` on the right. When the right-hand side operand contains entries with keys present in the left-hand side `Map`, the result map contains the entries from the right side.

```
val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
println(numbersMap + Pair("four", 4))
println(numbersMap + Pair("one", 10))
println(numbersMap + mapOf("five" to 5, "one" to 11))
```

`minus` creates a `Map` from entries of a `Map` on the left except those with keys from the right-hand side operand. So, the right-hand side operand can be either a single key or a collection of keys: list, set, and so on.

```
val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
println(numbersMap - "one")
println(numbersMap - listOf("two", "four"))
```

For details on using `plusAssign` (`+=`) and `minusAssign` (`-=`) operators on mutable maps, see Map write operations below.

Map write operations

Mutable maps offer map-specific write operations. These operations let you change the map content using the key-based access to the values.

There are certain rules that define write operations on maps:

— Values can be updated. In turn, keys never change: once you add an entry, its key is constant.

— For each key, there is always a single value associated with it. You can add and remove whole entries.

Below are descriptions of the standard library functions for write operations available on mutable maps.

Adding and updating 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.

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap.put("three", 3)
println(numbersMap)
```

To add multiple entries at a time, use putAll(). Its argument can be a Map or a group of Pair s: Iterable , Sequence , or Array .

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
numbersMap.putAll(setOf("four" to 4, "five" to 5))
println(numbersMap)
```

Both put() and putAll() overwrite the values if the given keys already exist in the map. Thus, you can use them to update values of map entries.

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
val previousValue = numbersMap.put("one", 11)
println("value associated with 'one', before: $previousValue, after:
${numbersMap["one"]}")
println(numbersMap)
```

You can also add new entries to maps using the shorthand operator form. There are two ways:

— plusAssign (+=) operator.

— the [] operator alias for put() .

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap["three"] = 3     // calls numbersMap.put("three", 3)
numbersMap += mapOf("four" to 4, "five" to 5)
println(numbersMap)
```

When called with the key present in the map, operators overwrite the values of the corresponding entries.

Removing 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.

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
numbersMap.remove("one")
println(numbersMap)
numbersMap.remove("three", 4)            //doesn't remove anything
println(numbersMap)
```

You can also remove entries from a mutable map by their keys or values. To do this, call `remove()` on the map's keys or values providing the key or the value of an entry. When called on values, `remove()` removes only the first entry with the given value.

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3, "threeAgain" to 3)
numbersMap.keys.remove("one")
println(numbersMap)
numbersMap.values.remove(3)
println(numbersMap)
```

The `minusAssign` (`-=`) operator is also available for mutable maps.

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
numbersMap -= "two"
println(numbersMap)
numbersMap -= "five"            //doesn't remove anything
println(numbersMap)
```

Multiplatform Programming

Platform-Specific Declarations

> Multiplatform projects are an experimental feature in Kotlin 1.2 and 1.3. All of the language and tooling features described in this document are subject to change in future Kotlin versions.

One of the key capabilities of Kotlin's multiplatform code is a way for common code to depend on platform-specific declarations. In other languages, this can often be accomplished by building a set of interfaces in the common code and implementing these interfaces in platform-specific modules. However, this approach is not ideal in cases when you have a library on one of the platforms that implements the functionality you need, and you'd like to use the API of this library directly without extra wrappers. Also, it requires common declarations to be expressed as interfaces, which doesn't cover all possible cases.

As an alternative, Kotlin provides a mechanism of *expected and actual declarations*. With this mechanism, a common module can define *expected declarations*, and a platform module can provide *actual declarations* corresponding to the expected ones. To see how this works, let's look at an example first. This code is part of a common module:

```
package org.jetbrains.foo

expect class Foo(bar: String) {
    fun frob()
}

fun main() {
    Foo("Hello").frob()
}
```

And this is the corresponding JVM module:

```
package org.jetbrains.foo

actual class Foo actual constructor(val bar: String) {
    actual fun frob() {
        println("Frobbing the $bar")
    }
}
```

This illustrates several important points:

— An expected declaration in the common module and its actual counterparts always have exactly the same fully qualified name.

— An expected declaration is marked with the `expect` keyword; the actual declaration is marked with the `actual` keyword.

— All actual declarations that match any part of an expected declaration need to be marked as `actual`.

— Expected declarations never contain any implementation code.

Note that expected declarations are not restricted to interfaces and interface members. In this example, the expected class has a constructor and can be created directly from common code. You can apply the `expect` modifier to other declarations as well, including top-level declarations and annotations:

```
// Common
expect fun formatString(source: String, vararg args: Any): String

expect annotation class Test

// JVM
actual fun formatString(source: String, vararg args: Any) =
    String.format(source, *args)

actual typealias Test = org.junit.Test
```

The compiler ensures that every expected declaration has actual declarations in all platform modules that implement the corresponding common module, and reports an error if any actual declarations are missing. 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 common 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>
```

Building Multiplatform Projects with Gradle

> Multiplatform projects are an experimental feature in Kotlin 1.2 and 1.3. All of the language and tooling features described in this document are subject to change in future Kotlin versions.

This document explains the structure of Kotlin multiplatform projects and describes how those are configured and built using Gradle.

Table of Contents

Project Structure

The layout of a Kotlin multiplatform project is constructed out of the following building blocks:

— A target is a part of the build that is responsible for building, testing, and packaging a complete piece of software for one of the platforms. Therefore, a multiplatform project usually contains multiple targets.

— Building each target involves compiling Kotlin sources once or multiple times. In other words, a target may have one or more [compilations](). For example, one compilation for production sources, the other for tests.

— The Kotlin sources are arranged into [source sets](). In addition to Kotlin source files and resources, each source set may have its own dependencies. Source sets form a hierarchy that is built with the *"depends on"* relation. A source set by itself is platform agnostic, but it may contain platform-specific code and dependencies if it's only compiled for a single platform.

Each compilation has a default source set, which is the place for sources and dependencies that are specific to that compilation. The default source set is also used for directing other source sets to the compilation by the means of the "depends on" relation.

Here's an illustration of what a project targeting the JVM and JS looks like:

Here, the two targets, `jvm` and `js`, each compile the production and test sources, and some of the sources are shared. This layout is achieved by just creating the two targets, with no additional configuration for the compilations and source sets: those are [created by default]() for these targets.

In the example above, the production sources for the JVM target are compiled by its `main` compilation and therefore include the sources and dependencies from the source sets `jvmMain` and `commonMain` (due to the *depends on* relation):

Here, the `jvmMain` source set provides [plaform-specific implementations]() for the expected API in the shared `commonMain` sources. This is how the code is shared between the platforms in a flexible way with platform-specific implementations where needed.

In further sections, these concepts are described in more detail along with the DSL to configure them in a project.

Setting up a Multiplatform Project

You can create a new multiplatform project in the IDE by selecting one of the multiplatform project templates in the New Project dialog under the "Kotlin" section.

For example, if you choose "Kotlin (Multiplatform Library)", a library project is created that has three [targets](), one for the JVM, one for JS, and one for the Native platform that you are using. These are configured in the `build.gradle` script in the following way:

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.3.61'
}

repositories {
    mavenCentral()
}

kotlin {
    jvm() // Creates a JVM target with the default name 'jvm'
    js()  // JS target named 'js'
    mingwX64("mingw") // Windows (MinGW X64) target named 'mingw'

    sourceSets { /* ... */ }
}
```

```
plugins {
    kotlin("multiplatform") version "1.3.61"
}

repositories {
    mavenCentral()
}

kotlin {
    jvm() // Creates a JVM target with the default name 'jvm'
    js()  // JS target named 'js'
    mingwX64("mingw") // Windows (MinGW X64) target named 'mingw'

    sourceSets { /* ... */ }
}
```

The three targets are created with the preset functions `jvm()`, `js()`, and `mingwX64()` that provide some default configuration. There are presets for each of the supported platforms.

The source sets and their dependencies are then configured as follows:

```
plugins { /* ... */ }

kotlin {
    /* Targets declarations omitted */

    sourceSets {
        commonMain {
            dependencies {
                implementation kotlin('stdlib-common')
            }
        }
        commonTest {
            dependencies {
                implementation kotlin('test-common')
                implementation kotlin('test-annotations-common')
            }
        }

        // Default source set for JVM-specific sources and dependencies.
        // Alternatively, jvmMain { ... } would work as well:
        jvm().compilations.main.defaultSourceSet {
            dependencies {
```

```
                        implementation kotlin('stdlib-jdk8')
                }
        }
        // JVM-specific tests and their dependencies:
        jvm().compilations.test.defaultSourceSet {
                dependencies {
                        implementation kotlin('test-junit')
                }
        }

        js().compilations.main.defaultSourceSet  { /* ... */ }
        js().compilations.test.defaultSourceSet { /* ... */ }

        mingwX64('mingw').compilations.main.defaultSourceSet { /* ... */ }
        mingwX64('mingw').compilations.test.defaultSourceSet { /* ... */ }
    }
}
```

```
plugins { /* ... */ }

kotlin {
    /* Targets declarations omitted */

    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation(kotlin("stdlib-common"))
            }
        }
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test-common"))
                implementation(kotlin("test-annotations-common"))
            }
        }

        // Default source set for JVM-specific sources and dependencies:
        jvm().compilations["main"].defaultSourceSet {
            dependencies {
                implementation(kotlin("stdlib-jdk8"))
            }
        }
        // JVM-specific tests and their dependencies:
        jvm().compilations["test"].defaultSourceSet {
            dependencies {
                implementation(kotlin("test-junit"))
            }
        }

        js().compilations["main"].defaultSourceSet  { /* ... */ }
        js().compilations["test"].defaultSourceSet { /* ... */ }

        mingwX64("mingw").compilations["main"].defaultSourceSet { /* ... */ }
        mingwX64("mingw").compilations["test"].defaultSourceSet { /* ... */ }
    }
}
```

These are the [default source set names](#) for the production and test sources for the targets configured above. The source sets `commonMain` and `commonTest` are included into production and test compilations, respectively, of all targets. Note that the dependencies for common source sets `commonMain` and `commonTest` are the common artifacts, and the platform libraries go to the source sets of the specific targets.

Gradle Plugin

Kotlin Multiplatform projects require Gradle version 4.7 and above, older Gradle versions are not supported.

To setup a multiplatform project from scratch in a Gradle project, first apply the `kotlin-multiplatform` plugin to the project by adding the following to the beginning of the `build.gradle` file:

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.3.61'
}
```

```
plugins {
    kotlin("multiplatform") version "1.3.61"
}
```

This creates the `kotlin` extension at the top level. You can then access it in the build script for:

— [setting up the targets](#) for multiple platforms (no targets are created by default);

— [configuring the source sets](#) and their [dependencies](#);

Setting up Targets

A target is a part of the build responsible for compiling, testing, and packaging a piece of software aimed for one of the [supported platforms](#).

All of the targets may share some of the sources and may have platform-specific sources as well.

As the platforms are different, targets are built in different ways as well and have various platform-specific settings. The Gradle plugin bundles a number of presets for the supported platforms.

To create a target, use one of the preset functions, which are named according to the target platforms and optionally accept the target name and a configuring code block:

```
kotlin {
    jvm() // Create a JVM target with the default name 'jvm'
    js("nodeJs") // Create a JS target with a custom name 'nodeJs'

    linuxX64("linux") {
        /* Specify additional settings for the 'linux' target here */
    }
}
```

The preset functions return an existing target if there is one. This can be used to configure an existing target:

```kotlin
kotlin {
    /* ... */

    // Configure the attributes of the 'jvm6' target:
    jvm("jvm6").attributes { /* ... */ }
}
```

Note that both the target platform and the name matter: if a target was created as `jvm('jvm6')`, using `jvm()` will create a separate target (with the default name `jvm`). If the preset function used to create the target under that name was different, an error is reported.

The targets created from presets are added to the `kotlin.targets` domain object collection, which can be used to access them by their names or configure all targets:

```kotlin
kotlin {
    jvm()
    js("nodeJs")

    println(targets.names) // Prints: [jvm, metadata, nodeJs]

    // Configure all targets, including those which will be added later:
    targets.all {
        compilations["main"].defaultSourceSet { /* ... */ }
    }
}
```

To create or access several targets from multiple presets dynamically, you can use the `targetFromPreset` function which accepts a preset (those are contained in the `kotlin.presets` domain object collection) and, optionally, a target name and a configuration code block.

For example, to create a target for each of the Kotlin/Native supported platforms (see below), use this code:

```kotlin
kotlin {

presets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTargetPreset).each {
        targetFromPreset(it) {
            /* Configure each of the created targets */
        }
    }
}
```

```
import org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTargetPreset

/* ... */

kotlin {
    presets.withType<KotlinNativeTargetPreset>().forEach {
        targetFromPreset(it) {
            /* Configure each of the created targets */
        }
    }
}
```

Supported platforms

There are target presets that one can apply using the preset functions, as shown above, for the following target platforms:

— `jvm` for Kotlin/JVM;

— `js` for Kotlin/JS;

— `android` for Android applications and libraries. Note that one of the Android Gradle plugins should be applied before the target is created;

— Kotlin/Native target presets (see the [notes](#) below):

  — `androidNativeArm32` and `androidNativeArm64` for Android NDK;

  — `iosArm32`, `iosArm64`, `iosX64` for iOS;

  — `watchosArm32`, `watchosArm64`, `watchosX86` for watchOS;

  — `tvosArm64`, `tvosX64` for tvOS;

  — `linuxArm32Hfp`, `linuxMips32`, `linuxMipsel32`, `linuxX64` for Linux;

  — `macosX64` for MacOS;

  — `mingwX64` and `mingwX86` for Windows;

  — `wasm32` for WebAssembly.

  Note that some of the Kotlin/Native targets require an [appropriate host machine](#) to build on.

Some targets may require additional configuration. For Android and iOS examples, see the [Multiplatform Project: iOS and Android](#) tutorial.

Configuring compilations

Building a target requires compiling Kotlin once or multiple times. Each Kotlin compilation of a target may serve a different purpose (e.g. production code, tests) and incorporate different [source sets](#). The compilations of a target may be accessed in the DSL, for example, to get the tasks, configure [the Kotlin compiler options](#) or get the dependency files and compilation outputs:

```
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 {
    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:
```

```
    // Configure all compilations of all targets:
    targets.all {
        compilations.all {
            kotlinOptions {
                allWarningsAsErrors = true
            }
        }
    }
}
```

Each compilation is accompanied by a default [source set](#) that stores sources and dependencies specific to that compilation. The default source set for a compilation `foo` of a target `bar` has the name `barFoo`. It can also be accessed from a compilation using `defaultSourceSet`:

```
kotlin {
    jvm() // Create a JVM target with the default name 'jvm'

    sourceSets {
        // The default source set for the 'main` compilation of the 'jvm' target:
        jvmMain {
            /* ... */
        }
    }

    // Alternatively, access it from the target's compilation:
    jvm().compilations.main.defaultSourceSet {
        /* ... */
    }
}
```

```
kotlin {
    jvm() // Create a JVM target with the default name 'jvm'

    sourceSets {
        // The default source set for the 'main` compilation of the 'jvm' target:
        val jvmMain by getting {
            /* ... */
        }
    }

    // Alternatively, access it from the target's compilation:
    jvm().compilations["main"].defaultSourceSet {
        /* ... */
    }
}
```

To collect all source sets participating in a compilation, including those added via the depends-on relation, one can use the property `allKotlinSourceSets`.

For some specific use cases, creating a custom compilation may be required. This can be done within the target's `compilations` domain object collection. Note that the dependencies need to be set up manually for all custom compilations, and the usage of a custom compilation's outputs is up to the build author. For example, consider a custom compilation for integration tests of a `jvm()` target:

```
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.create('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 {
    jvm() {
        compilations {
            val main by getting

            val integrationTest by compilations.creating {
                defaultSourceSet {
                    dependencies {
```

```
                    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.create<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
                }
            }
        }
    }
}
```

Also note that the default source set of a custom compilation depends on neither `commonMain` nor `commonTest` by default.

Configuring Source Sets

A Kotlin source set is a collection of Kotlin sources, along with their resources, dependencies, and language settings, which may take part in Kotlin compilations of one or more [targets](.).

A source set is not bound to be platform-specific or "shared"; what it's allowed to contain depends on its usage: a source set added to multiple compilations is limited to the common language features and dependencies, while a source set that is only used by a single target can have platform-specific dependencies, and its code may use language features specific to that target's platform.

Some source sets are created and configured by default: `commonMain`, `commonTest`, and the default source sets for the compilations. See [Default Project Layout](.).

The source sets are configured within a `sourceSets { ... }` block of the `kotlin { ... }` extension:

```
kotlin {
    sourceSets {
        foo { /* ... */ } // create or configure a source set by the name 'foo'
        bar { /* ... */ }
    }
}
```

```
kotlin {
    sourceSets {
        val foo by creating { /* ... */ } // create a new source set by the name 'foo'
        val bar by getting { /* ... */ } // configure an existing source set by the name
'bar'
    }
}
```

> Note: creating a source set does not link it to any target. Some source sets are
> predefined and thus compiled by default. However, custom source sets always need to
> be explicitly directed to the compilations. See: Connecting source sets.

The source set names are case-sensitive. When referring to a default source set by its name,
make sure the name prefix matches a target's name, for example, a source set `iosX64Main` for
a target `iosX64`.

A source set by itself is platform-agnostic, but it can be considered platform-specific if it is only
compiled for a single platform. A source set can, therefore, contain either common code shared
between the platforms or platform-specific code.

Each source set has a default source directory for Kotlin sources: `src/<source set
name>/kotlin`. To add Kotlin source directories and resources to a source set, use its `kotlin`
and `resources` `SourceDirectorySet`s:

By default, the files of a source set are stored in the following directories:

— source files: `src/<source set name>/kotlin`

— resources files: `src/<source set name>/resources`

You should create these directories manually.

To add custom Kotlin source directories and resources to a source set, use its `kotlin` and
`resources` `SourceDirectorySet`s:

```
kotlin {
    sourceSets {
        commonMain {
            kotlin.srcDir('src')
            resources.srcDir('res')
        }
    }
}
```

```
kotlin {
    sourceSets {
        val commonMain by getting {
            kotlin.srcDir("src")
            resources.srcDir("res")
        }
    }
}
```

Connecting source sets

Kotlin source sets may be connected with the *'depends on'* relation, so that if a source set `foo` depends on a source set `bar` then:

— whenever `foo` is compiled for a certain target, `bar` takes part in that compilation as well and is also compiled into the same target binary form, such as JVM class files or JS code;

— sources of `foo` 'see' the declarations of `bar`, including the `internal` ones, and the [dependencies](#) of `bar`, even those specified as `implementation` dependencies;

— `foo` may contain [platform-specific implementations](#) for the expected declarations of `bar`;

— the resources of `bar` are always processed and copied along with the resources of `foo`;

— the [language settings](#) of `foo` and `bar` should be consistent;

Circular source set dependencies are prohibited.

The source sets DSL can be used to define these connections between the source sets:

```
kotlin {
    sourceSets {
        commonMain { /* ... */ }
        allJvm {
            dependsOn commonMain
            /* ... */
        }
    }
}
```

```
kotlin {
    sourceSets {
        val commonMain by getting { /* ... */ }
        val allJvm by creating {
            dependsOn(commonMain)
            /* ... */
        }
    }
}
```

Custom source sets created in addition to the [default ones](#) should be explicitly included into the dependencies hierarchy to be able to use declarations from other source sets and, most importantly, to take part in compilations. Most often, they need a `dependsOn(commonMain)` or `dependsOn(commonTest)` statement, and some of the default platform-specific source sets should depend on the custom ones, directly or indirectly:

```kotlin
kotlin {
    mingwX64()
    linuxX64()

    sourceSets {
        // custom source set with tests for the two targets
        desktopTest {
            dependsOn commonTest
            /* ... */
        }
        // Make the 'windows' default test source set for depend on 'desktopTest'
        mingwX64().compilations.test.defaultSourceSet {
            dependsOn desktopTest
            /* ... */
        }
        // And do the same for the other target:
        linuxX64().compilations.test.defaultSourceSet {
            dependsOn desktopTest
            /* ... */
        }
    }
}
```

```kotlin
kotlin {
    mingwX64()
    linuxX64()

    sourceSets {
        // custom source set with tests for the two targets
        val desktopTest by creating {
            dependsOn(getByName("commonTest"))
            /* ... */
        }
        // Make the 'windows' default test source set for depend on 'desktopTest'
        mingwX64().compilations["test"].defaultSourceSet {
            dependsOn(desktopTest)
            /* ... */
        }
        // And do the same for the other target:
        linuxX64().compilations["test"].defaultSourceSet {
            dependsOn(desktopTest)
            /* ... */
        }
    }
}
```

Adding Dependencies

To add a dependency to a source set, use a `dependencies { ... }` block of the source sets DSL. Four kinds of dependencies are supported:

— `api` dependencies are used both during compilation and at runtime and are exported to library consumers. If any types from a dependency are used in the public API of the current module, then it should be an `api` dependency;

— `implementation` dependencies are used during compilation and at runtime for the current module, but are not exposed for compilation of other modules depending on the one with the `implementation` dependency. The `implementation` dependency kind should be used for dependencies needed for the internal logic of a module. If a module is an endpoint application which is not published, it may use `implementation` dependencies instead of `api` ones.

— `compileOnly` dependencies are only used for compilation of the current module and are available neither at runtime nor during compilation of other modules. These dependencies should be used for APIs which have a third-party implementation available at runtime.

— `runtimeOnly` dependencies are available at runtime but are not visible during compilation of any module.

Dependencies are specified per source set as follows:

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                api 'com.example:foo-metadata:1.0'
            }
        }
        jvm6Main {
            dependencies {
                api 'com.example:foo-jvm6:1.0'
            }
        }
    }
}
```

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                api("com.example:foo-metadata:1.0")
            }
        }
        val jvm6Main by getting {
            dependencies {
                api("com.example:foo-jvm6:1.0")
            }
        }
    }
}
```

Note that for the IDE to correctly analyze the dependencies of the common sources, the common source sets need to have corresponding dependencies on the Kotlin metadata packages in addition to the platform-specific artifact dependencies of the platform-specific source sets. Usually, an artifact with a suffix `-common` (as in `kotlin-stdlib-common`) or `-metadata` is required when using a published library (unless it is published with Gradle metadata, as described below).

However, a `project('...')` dependency on another multiplatform project is resolved to an appropriate target automatically. It is enough to specify a single `project('...')` dependency in a source set's dependencies, and the compilations that include the source set will receive a corresponding platform-specific artifact of that project, given that it has a compatible target:

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                // All of the compilations that include source set 'commonMain'
                // will get this dependency resolved to a compatible target, if any:
                api project(':foo-lib')
            }
        }
    }
}
```

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                // All of the compilations that include source set 'commonMain'
                // will get this dependency resolved to a compatible target, if any:
                api(project(":foo-lib"))
            }
        }
    }
}
```

Likewise, if a multiplatform library is published in the experimental [Gradle metadata publishing mode](#) and the project is set up to consume the metadata as well, then it is enough to specify a dependency only once, for the common source set. Otherwise, each platform-specific source set should be provided with a corresponding platform module of the library, in addition to the common module, as shown above.

An alternative way to specify the dependencies is to use the Gradle built-in DSL at the top level with the configuration names following the pattern `<sourceSetName><DependencyKind>`:

```
dependencies {
    commonMainApi 'com.example:foo-common:1.0'
    jvm6MainApi 'com.example:foo-jvm6:1.0'
}
```

```
dependencies {
    "commonMainApi"("com.example:foo-common:1.0")
    "jvm6MainApi"("com.example:foo-jvm6:1.0")
}
```

Some of the Gradle built-in dependencies, like `gradleApi()`, `localGroovy()`, or
`gradleTestKit()` are not available in the source sets dependency DSL. You can, however, add
them within the top-level dependency block, as shown above.

A dependency on a Kotlin module like `kotlin-stdlib` or `kotlin-reflect` may be added
with the notation `kotlin("stdlib")`, which is a shorthand for
`"org.jetbrains.kotlin:kotlin-stdlib"`.

Language settings

The language settings for a source set can be specified as follows:

```
kotlin {
    sourceSets {
        commonMain {
            languageSettings {
                languageVersion = '1.3' // possible values: '1.0', '1.1', '1.2', '1.3'
                apiVersion = '1.3' // possible values: '1.0', '1.1', '1.2', '1.3'
                enableLanguageFeature('InlineClasses') // language feature name
                useExperimentalAnnotation('kotlin.ExperimentalUnsignedTypes') //
annotation FQ-name
                progressiveMode = true // false by default
            }
        }
    }
}
```

```
kotlin {
    sourceSets {
        val commonMain by getting {
            languageSettings.apply {
                languageVersion = "1.3" // possible values: '1.0', '1.1', '1.2', '1.3'
                apiVersion = "1.3" // possible values: '1.0', '1.1', '1.2', '1.3'
                enableLanguageFeature("InlineClasses") // language feature name
                useExperimentalAnnotation("kotlin.ExperimentalUnsignedTypes") //
annotation FQ-name
                progressiveMode = true // false by default
            }
        }
    }
}
```

It is possible to configure the language settings of all source sets at once:

```
kotlin.sourceSets.all {
    languageSettings.progressiveMode = true
}
```

Language settings of a source set affect how the sources are analyzed in the IDE. Due to the current limitations, in a Gradle build, only the language settings of the compilation's default source set are used and are applied to all of the sources participating in the compilation.

The language settings are checked for consistency between source sets depending on each other. Namely, if `foo` depends on `bar`:

— `foo` should set `languageVersion` that is greater than or equal to that of `bar`;

— `foo` should enable all unstable language features that `bar` enables (there's no such requirement for bugfix features);

— `foo` should use all experimental annotations that `bar` uses;

— `apiVersion`, bugfix language features, and `progressiveMode` can be set arbitrarily;

Default Project Layout

By default, each project contains two source sets, `commonMain` and `commonTest`, where one can place all the code that should be shared between all of the target platforms. These source sets are added to each production and test compilation, respectively.

Then, once a target is added, default compilations are created for it:

— `main` and `test` compilations for JVM, JS, and Native targets;

— a compilation per [Android build variant](), for Android targets;

For each compilation, there is a default source set under the name composed as `<targetName><CompilationName>`. This default source set participates in the compilation, and thus it should be used for the platform-specific code and dependencies, and for adding other source sets to the compilation by the means of 'depends on'. For example, a project with targets `jvm6` (JVM) and `nodeJs` (JS) will have source sets: `commonMain`, `commonTest`, `jvm6Main`, `jvm6Test`, `nodeJsMain`, `nodeJsTest`.

Numerous use cases are covered by just the default source sets and don't require custom source sets.

Each source set by default has its Kotlin sources under `src/<sourceSetName>/kotlin` directory and the resources under `src/<sourceSetName>/resources`.

In Android projects, additional Kotlin source sets are created for each [Android source set](). If the Android target has a name `foo`, the Android source set `bar` gets a Kotlin source set counterpart `fooBar`. The Kotlin compilations, however, are able to consume Kotlin sources from all of the directories `src/bar/java`, `src/bar/kotlin`, and `src/fooBar/kotlin`. Java sources are only read from the first of these directories.

Running Tests

Running tests in a Gradle build is currently supported by default for JVM, Android, Linux, Windows and macOS; JS and other Kotlin/Native targets need to be manually configured to run the tests with an appropriate environment, an emulator or a test framework.

A test task is created under the name `<targetName>Test` for each target that is suitable for testing. Run the `check` task to run the tests for all targets.

As the `commonTest` [default source set](#) is added to all test compilations, tests and test tools that are needed on all target platforms may be placed there.

The [kotlin.test API](#) is available for multiplatform tests. Add the `kotlin-test-common` and `kotlin-test-annotations-common` dependencies to `commonTest` to use the assertion functions like `kotlin.test.assertTrue(...)` and `@Test`/`@Ignore`/`@BeforeTest`/`@AfterTest` annotations in the common tests.

For JVM targets, use `kotlin-test-junit` or `kotlin-test-testng` for the corresponding asserter implementation and annotations mapping.

For Kotlin/JS targets, add `kotlin-test-js` as a test dependency. At this point, test tasks for Kotlin/JS are created but do not run tests by default; they should be manually configured to run the tests with a JavaScript test framework.

Kotlin/Native targets do not require additional test dependencies, and the `kotlin.test` API implementations are built-in.

Publishing a Multiplatform Library

> The set of target platforms is defined by a multiplatform library author, and they should provide all of the platform-specific implementations for the library. Adding new targets for a multiplatform library at the consumer's side is not supported.

A library built from a multiplatform project may be published to a Maven repository with the [maven-publish Gradle plugin](#), which can be applied as follows:

```
plugins {
    /* ... */
    id("maven-publish")
}
```

A library also needs `group` and `version` to be set in the project:

```
plugins { /* ... */ }

group = "com.example.my.library"
version = "0.0.1"
```

Compared to publishing a plain Kotlin/JVM or Java project, there is no need to create publications manually via the `publishing { ... }` DSL. The publications are automatically created for each of the targets that can be built on the current host, except for the Android target, which needs an additional step to configure publishing, see [Publishing Android libraries](#).

The repositories where the library will be published are added via the `repositories` block in the `publishing { ... }` DSL, as explained in [Maven Publish Plugin. Repositories](#).

The default artifact IDs follow the pattern `<projectName>-<targetNameToLowerCase>`, for example `sample-lib-nodejs` for a target named `nodeJs` in a project `sample-lib`.

By default, a sources JAR is added to each publication in addition to its main artifact. The sources JAR contains the sources used by the `main` compilation of the target. If you also need to publish a documentation artifact (like a Javadoc JAR), you need to configure its build manually and add it as an artifact to the relevant publications, as shown below.

Also, an additional publication under the name `metadata` is added by default which contains serialized Kotlin declarations and is used by the IDE to analyze multiplatform libraries. The default artifact ID of this publication is formed as `<projectName>-metadata`.

The Maven coordinates can be altered and additional artifact files may be added to the publications within the `targets { ... }` block or the `publishing { ... }` DSL:

```kotlin
kotlin {
    jvm('jvm6') {
        mavenPublication { // Setup the publication for the target 'jvm6'
            // The default artifactId was 'foo-jvm6', change it:
            artifactId = 'foo-jvm'
            // Add a docs JAR artifact (it should be a custom task):
            artifact(jvmDocsJar)
        }
    }
}

// Alternatively, configure the publications with the `publishing { ... }` DSL:
publishing {
    publications {
        jvm6 { /* Setup the publication for target 'jvm6' */ }
        metadata { /* Setup the publication for Kotlin metadata */ }
    }
}
```

```kotlin
kotlin {
    jvm("jvm6") {
        mavenPublication { // Setup the publication for the target 'jvm6'
            // The default artifactId was 'foo-jvm6', change it:
            artifactId = "foo-jvm"
            // Add a docs JAR artifact (it should be a custom task):
            artifact(jvmDocsJar)
        }
    }
}

// Alternatively, configure the publications with the `publishing { ... }` DSL:
publishing {
    publications.withType<MavenPublication>().apply {
        val jvm6 by getting { /* Setup the publication for target 'jvm6' */ }
        val metadata by getting { /* Setup the publication for Kotlin metadata */ }
    }
}
```

As assembling Kotlin/Native artifacts requires several builds to run on different host platforms, publishing a multiplatform library that includes Kotlin/Native targets needs to be done with that same set of host machines. To avoid duplicate publications of modules that can be built on more than one of the platforms (like JVM, JS, Kotlin metadata, WebAssembly), the publishing tasks for these modules may be configured to run conditionally.

This simplified example ensures that the JVM, JS, and Kotlin metadata publications are only uploaded when `-PisLinux=true` is passed to the build in the command line:

```kotlin
kotlin {
    jvm()
    js()
    mingwX64()
    linuxX64()

    // Note that the Kotlin metadata is here, too.
    // The mingwx64() target is automatically skipped as incompatible in Linux builds.
    configure([targets["metadata"], jvm(), js()]) {
        mavenPublication { targetPublication ->
            tasks.withType(AbstractPublishToMaven)
                .matching { it.publication == targetPublication }
                .all { onlyIf { findProperty("isLinux") == "true" } }
        }
    }
}
```

```
kotlin {
    jvm()
    js()
    mingwX64()
    linuxX64()

    // Note that the Kotlin metadata is here, too.
    // The mingwx64() target is automatically skipped as incompatible in Linux builds.
    configure(listOf(metadata(), jvm(), js())) {
        mavenPublication {
            val targetPublication = this@mavenPublication
            tasks.withType<AbstractPublishToMaven>()
                .matching { it.publication == targetPublication }
                .all { onlyIf { findProperty("isLinux") == "true" } }
        }
    }
}
```

Experimental metadata publishing mode

Gradle module metadata provides rich publishing and dependency resolution features that are used in Kotlin multiplatform projects to simplify dependencies configuration for build authors. In particular, the publications of a multiplatform library may include a special 'root' module that stands for the whole library and is automatically resolved to the appropriate platform-specific artifacts when added as a dependency, as described below.

In Gradle 5.3 and above, the module metadata is always used during dependency resolution, but publications don't include any module metadata by default. To enable module metadata publishing, add `enableFeaturePreview("GRADLE_METADATA")` to the root project's `settings.gradle` file. With older Gradle versions, this is also required for module metadata consumption.

> Note that the module metadata published by Gradle 5.3 and above cannot be read by Gradle versions older than 5.3.

With Gradle metadata enabled, an additional 'root' publication named `kotlinMultiplatform` is added to the project's publications. The default artifact ID of this publication matches the project name without any additional suffix. To configure this publication, access it via the `publishing { ... }` DSL of the `maven-publish` plugin:

```
kotlin { /* ... */ }

publishing {
    publications {
        kotlinMultiplatform {
            artifactId = "foo"
        }
    }
}
```

```
kotlin { /* ... */ }

publishing {
    publications {
        val kotlinMultiplatform by getting {
            artifactId = "foo"
        }
    }
}
```

This publication does not include any artifacts and only references the other publications as its variants. However, it may 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, which is accessed as shown above.

If a library has a 'root' publication, the consumer may specify a single dependency on the library as a whole in a common source set, and a corresponding platform-specific variant will be chosen, if available, for each of the compilations that include this dependency. Consider a `sample-lib` library built for the JVM and JS and published with a 'root' publication:

```
kotlin {
    jvm('jvm6')
    js('nodeJs')

    sourceSets {
        commonMain {
            dependencies {
                // This single dependency is resolved to the appropriate target modules,
                // for example, `sample-lib-jvm6` for JVM, `sample-lib-js` for JS:
                api 'com.example:sample-lib:1.0'
            }
        }
    }
}
```

```
kotlin {
    jvm("jvm6")
    js("nodeJs")

    sourceSets {
        val commonMain by getting {
            dependencies {
                // This single dependency is resolved to the appropriate target modules,
                // for example, `sample-lib-jvm6` for JVM, `sample-lib-js` for JS:
                api("com.example:sample-lib:1.0")
            }
        }
    }
}
```

This requires that the consumer's Gradle build can read Gradle module metadata, either using Gradle 5.3+ or explicitly enabling it by `enableFeaturePreview("GRADLE_METADATA")` in `settings.gradle`.

Disambiguating targets

It is possible to have more than one target for a single platform in a multiplatform library. For example, these targets may provide the same API and differ in the libraries they cooperate with at runtime, like testing frameworks or logging solutions.

However, dependencies on such a multiplatform library may be ambiguous and may thus fail to resolve because there is not enough information to decide which of the targets to choose.

The solution is to mark the targets with a custom attribute, which is taken into account by Gradle during dependency resolution. This, however, must be done on both the library author and the consumer sides, and it's the library author's responsibility to communicate the attribute and its possible values to the consumers.

Adding attributes is done symmetrically, to both the library and the consumer projects. 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:

```
def testFrameworkAttribute = Attribute.of('com.example.testFramework', String)

kotlin {
    jvm('junit') {
        attributes.attribute(testFrameworkAttribute, 'junit')
    }
    jvm('testng') {
        attributes.attribute(testFrameworkAttribute, 'testng')
    }
}
```

```
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 may only need to add the attribute to a single target where the ambiguity arises.

If the same kind of ambiguity arises when a dependency is added to a custom configuration rather than one of the configurations created by the plugin, you can add the attributes to the configuration in the same way:

```
def testFrameworkAttribute = Attribute.of('com.example.testFramework', String)

configurations {
    myConfiguration {
        attributes.attribute(testFrameworkAttribute, 'junit')
    }
}
```

```
val testFrameworkAttribute = Attribute.of("com.example.testFramework",
String::class.java)

configurations {
    val myConfiguration by creating {
        attributes.attribute(testFrameworkAttribute, "junit")
    }
}
```

Java Support in JVM Targets

This feature is available since Kotlin 1.3.40.

By default, a JVM target ignores Java sources and only compiles Kotlin source files.
To include Java sources in the compilations of a JVM target, or to apply a Gradle plugin that
requires the `java` plugin to work, you need to explicitly enable Java support for the target:

```
kotlin {
    jvm {
        withJava()
    }
}
```

This will apply the Gradle `java` plugin and configure the target to cooperate with it. Note that
just applying the Java plugin without specifying `withJava()` in a JVM target will have no effect
on the target.

The file system locations for the Java sources are different from the `java` plugin's defaults. The
Java source files need to be placed in the sibling directories of the Kotlin source roots. For
example, if the JVM target has the default name `jvm`, the paths are:

```
src
├── jvmMain
│   ├── java // production Java sources
│   ├── kotlin
│   └── resources
├── jvmTest
│   ├── java // test Java sources
│   ├── kotlin
…   └── resources
```

The common source sets cannot include Java sources.

Due to the current limitations, some tasks configured by the Java plugin are disabled, and the
corresponding tasks added by the Kotlin plugin are used instead:

— `jar` is disabled in favor of the target's JAR task (e.g. `jvmJar`)

— `test` is disabled, and the target's test task is used (e.g. `jvmTest`)

— `*ProcessResources` tasks are disabled, and the resources are processed by the equivalent tasks of the compilations

The publication of this target is handled by the Kotlin plugin and doesn't require the steps that are specific to the Java plugin, such as manually creating a publication and configuring it as `from(components.java)`.

Android Support

Kotlin Multiplatform projects support the Android platform by providing the `android` preset. Creating an Android target requires that one of the Android Gradle plugins, like `com.android.application` or `com.android.library` is manually applied to the project. Only one Android target may be created per Gradle subproject:

```
plugins {
    id("com.android.library")
    id("org.jetbrains.kotlin.multiplatform").version("1.3.61")
}

android { /* ... */ }

kotlin {
    android { // Create the Android target
        // Provide additional configuration if necessary
    }
}
```

```
plugins {
    id("com.android.library")
    kotlin("multiplatform").version("1.3.61")
}

android { /* ... */ }

kotlin {
    android { // Create the Android target
        // Provide additional configuration if necessary
    }
}
```

An Android target's compilations created 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 by the variants, a Kotlin source set is created under that source set name prepended by the target name, like 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 the 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 sets dependencies.

```
// ...

kotlin {
    android { /* ... */ }
}

dependencies {
    kapt("com.my.annotation:processor:1.0.0")
}
```

Publishing Android libraries

To publish an Android library as a part of a multiplatform library, one needs to [setup publishing for the library](#) and provide additional configuration for the Android library target.

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 as follows:

```
kotlin {
    android {
        publishLibraryVariants("release", "debug")
    }
}
```

The example above will work for Android libraries with no product flavors. For a library with product flavors, the variant names also contain the flavors, like `fooBarDebug` or `fooBazRelease`.

Note that 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, it will be necessary to provide a fallback for the consumers who have such a build type, specifying at least one of the build types that the library publishes:

```
android {
    buildTypes {
        staging {
            // ...
            matchingFallbacks = ['release', 'debug']
        }
    }
}
```

```
android {
    buildTypes {
        val staging by creating {
            // ...
            matchingFallbacks = listOf("release", "debug")
        }
    }
}
```

Similarly, a library consumer may need to provide matching fallbacks for custom product flavors if some are missing in the library publications.

There is an option to 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 to publish variants grouped by the product flavor in case they have different dependencies, as those will be merged into one dependencies list.

Using Kotlin/Native Targets

It is important to note that some of the Kotlin/Native targets may only be built with an appropriate host machine:

— Linux MIPS targets (`linuxMips32` and `linuxMipsel32`) require a Linux host. Other Linux targets can be built on any supported host;

— Windows targets require a Windows host;

— macOS and iOS targets can only be built on a macOS host;

— The 64-bit Android Native target require a Linux or macOS host. The 32-bit Android Native target can be built on any supported host.

A target that is not supported by the current host is ignored during build and therefore not published. A library author may want to set up builds and publishing from different hosts as required by the library target platforms.

Target shortcuts

Some native targets are often created together and use the same sources. For example, building for an iOS device and a simulator is represented by different targets (`iosArm64` and `iosX64` respectively) but their source codes are usually the same. A canonical way to express such shared code in the multiplatform project model is creating an intermediate source set (`iosMain`) and configuring links between it and the platform source sets:

```
sourceSets{
    iosMain {
        dependsOn(commonMain)
        iosDeviceMain.dependsOn(it)
        iosSimulatorMain.dependsOn(it)
    }
}
```

```
val commonMain by sourceSets.getting
val iosDeviceMain by sourceSets.getting
val iosSimulatorMain by sourceSets.getting

val iosMain by sourceSets.creating {
    dependsOn(commonMain)
    iosDeviceMain.dependsOn(this)
    iosSimulatorMain.dependsOn(this)
}
```

Since 1.3.60, the `kotlin-multiplaform` plugin provides shortcuts that automate such a configuration: they let users create a group of targets along with a common source set for them with a single DSL method.

The following shortcuts are available:

— `ios` creates targets for `iosArm64` and `iosX64`.

— `watchos` creates targets for `watchosArm32`, `watchosArm64`, and `watchosX86`.

— `tvos` creates targets for `tvosArm64` and `tvosX64`.

```
// Create two targets for iOS.
// Create common source sets: iosMain and iosTest.
ios {
    // Configure targets.
    // Note: this lambda will be called for each target.
}

// You can also specify a name prefix for created targets.
// Common source sets will also have this prefix:
// anotherIosMain and anotherIosTest.
ios("anotherIos")
```

Building 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 like executables or shared libraries a `binaries` property of a native target is used. 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.

Note that the `kotlin-multiplaform` plugin doesn't create any production binaries by default. The only binary available by default is a debug executable allowing one to run tests from the `test` compilation.

Declaring binaries

A set of factory methods is used for declaring elements of the `binaries` collection. These methods allow one to specify what kinds of binaries are to be created and configure them. The following binary kinds are supported (note that not all the kinds are available for all native platforms):

Factory method	Binary kind	Available for
executable	a product executable	all native targets
test	a test executable	all native targets
sharedLib	a shared native library	all native targets except wasm32
staticLib	a static native library	all native targets except wasm32
framework	an Objective-C framework	macOS, iOS, watchOS, and tvOS targets only

Each factory method exists in several versions. Consider them by example of the `executable` method. All the same versions are available for all other factory methods.

The simplest version doesn't require any additional parameters and creates one binary for each build type. Currently there a two build types available: `DEBUG` (produces a not optimized binary with a debug information) and `RELEASE` (produces an optimized binary without debug information). Consequently the following snippet creates two executable binaries: debug and release.

```
kotlin {
    linuxX64 { // Use your target instead.
        binaries {
            executable {
                // Binary configuration.
            }
        }
    }
}
```

A lambda expression accepted by the `executable` method in the example above is applied to each binary created and allows one to configure the binary (see the [corresponding section](#)). Note that this lambda can be dropped if there is no need for additional configuration:

```
binaries {
    executable()
}
```

It is possible to specify which build types will be used to create binaries and which won't. In the following example only debug executable is created.

```
binaries {
    executable([DEBUG]) {
        // Binary configuration.
    }
}
```

```
binaries {
    executable(listOf(DEBUG)) {
        // Binary configuration.
    }
}
```

Finally the last factory method version allows customizing the binary name.

```
binaries {
    executable('foo', [DEBUG]) {
        // Binary configuration.
    }

    // It's possible to drop the list of build types (all the available build types will
be used in this case).
    executable('bar') {
        // Binary configuration.
    }
}
```

```
binaries {
    executable("foo", listOf(DEBUG)) {
        // Binary configuration.
    }

    // It's possible to drop the list of build types (all the available build types will
be used in this case).
    executable("bar") {
        // Binary configuration.
    }
}
```

The first argument in this example allows one to set a name prefix for the created binaries which is used to access them in the buildscript (see the ["Accessing binaries"](#) section). Also this prefix is used as a default name for the binary file. For example on Windows the sample above produces files `foo.exe` and `bar.exe`.

Accessing binaries

The binaries DSL allows not only creating binaries but also accessing already created ones to configure them or get their properties (e.g. path to an output file). The `binaries` collection implements the [DomainObjectSet](#) interface and provides methods like `all` or `matching` allowing configuring groups of elements.

Also it's possible to get a certain element of the collection. There are two ways to do this. First, each binary has a unique name. This name is based on the name prefix (if it's specified), build type and binary kind according to the following pattern: `<optional-name-prefix><build-type><binary-kind>`, e.g. `releaseFramework` or `testDebugExecutable`.

> Note: static and shared libraries has suffixes `static` and `shared` respectively, e.g. `fooDebugStatic` or `barReleaseShared`

This name can be used to access the binary:

```
// Fails if there is no such a binary.
binaries['fooDebugExecutable']
binaries.fooDebugExecutable
binaries.getByName('fooDebugExecutable')

 // Returns null if there is no such a binary.
binaries.findByName('fooDebugExecutable')
```

```
// Fails if there is no such a binary.
binaries["fooDebugExecutable"]
binaries.getByName("fooDebugExecutable")

 // Returns null if there is no such a binary.
binaries.findByName("fooDebugExecutable")
```

The second way is using typed getters. These getters allow one to access a binary of a certain type by its name prefix and build type.

```
// Fails if there is no such a 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 a binary.
binaries.findExecutable('foo', DEBUG)

// Similar getters are available for other binary kinds:
// findFramework, findStaticLib and findSharedLib.
```

```
// Fails if there is no such a 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 a binary.
binaries.findExecutable("foo", DEBUG)

// Similar getters are available for other binary kinds:
// findFramework, findStaticLib and findSharedLib.
```

> Before 1.3.40, both test and product executables were represented by the same binary type. Thus to access the default test binary created by the plugin, the following line was used:
>
> ```
> binaries.getExecutable("test", "DEBUG")
> ```
>
> Since 1.3.40, test executables are represented by a separate binary type and have their own getter. To access the default test binary, use:
>
> ```
> binaries.getTest("DEBUG")
> ```

Configuring binaries

Binaries have a set of properties allowing one to configure them. The following options are available:

— **Compilation.** Each binary is built on basis of some compilation available in the same target. The default value of this parameter depends on the binary type: `Test` binaries are based on the `test` compilation while other binaries - on the `main` compilation.

— **Linker options.** Options passed to a system linker during binary building. One can use this setting for linking against some native library.

— **Output file name.** By default the output file name is based on binary name prefix or, if the name prefix isn't specified, on a project name. But it's possible to configure the output file name independently using the `baseName` property. Note that final file name will be formed by adding system-dependent prefix and postfix to this base name. E.g. a `libfoo.so` is produced for a Linux shared library with the base name `foo`.

— **Entry point** (for executable binaries only). By default the entry point for Kotlin/Native programs is a `main` function located in the root package. This setting allows one to change this default and use a custom function as an entry point. For example it can be used to move the `main` function from the root package.

— **Access to the output file.**

— **Access to a link task.**

— **Access to a run task** (for executable binaries only). The `kotlin-multiplatform` plugin creates run tasks for all executable binaries of host platforms (Windows, Linux and macOS). Names of such tasks are based on binary names, e.g. `runReleaseExecutable<target-name>` or `runFooDebugExecutable<target-name>`. A run task can be accessed using the `runTask` property of an executable binary.

— **Framework type** (only for Objective-C frameworks). By default a framework built by Kotlin/Native contains a dynamic library. But it's possible to replace it with a static library.

The following example shows how to use these settings.

```
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.absolutePath}")

        // 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
    }
}
```

```
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.absolutePath}")

        // 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
    }
}
```

Exporting dependencies in frameworks

When building an Objective-C framework, it is often necessary to pack not just the classes of the current project, but also the classes of some of its dependencies. The Binaries DSL allows one to specify which dependencies will be exported in the framework using the `export` method. Note that only API dependencies of a corresponding source set can be exported.

```
kotlin {
    sourceSets {
        macosMain.dependencies {
            // Will be exported in the framework.
            api project(':dependency')
            api 'org.example:exported-library:1.0'

            // Will not be exported in the framework.
            api 'org.example:not-exported-library:1.0'
        }
    }

    macosX64("macos").binaries {
        framework {
            export project(':dependency')
            export 'org.example:exported-library:1.0'
        }
    }
}
```

```
kotlin {
    sourceSets {
        macosMain.dependencies {
            // Will be exported in the framework.
            api(project(":dependency"))
            api("org.example:exported-library:1.0")

            // Will not be exported in the framework.
            api("org.example:not-exported-library:1.0")
        }
    }

    macosX64("macos").binaries {
        framework {
            export(project(":dependency"))
            export("org.example:exported-library:1.0")
        }
    }
}
```

As shown in this example, maven dependency also can be exported. But due to current limitations of Gradle metadata such a dependency should be either a platform one (e.g. `kotlinx-coroutines-core-native_debug_macos_x64` instead of `kotlinx-coroutines-core-native`) or be exported transitively (see below).

By default, export works non-transitively. If a library `foo` depending on library `bar` is exported, only methods of `foo` will be added in the output framework. This behaviour can by changed by the `transitiveExport` flag.

```
binaries {
    framework {
        export project(':dependency')
        // Export transitively.
        transitiveExport = true
    }
}
```

```
binaries {
    framework {
        export(project(":dependency"))
        // Export transitively.
        transitiveExport = true
    }
}
```

Building universal frameworks

By default, an Objective-C framework produced by Kotlin/Native supports only one platform. However, such frameworks can be merged into a single universal (fat) binary using the `lipo` utility. Particularly, such an operation makes sense for 32-bit and 64-bit iOS frameworks. In this case the resulting universal framework can be used on both 32-bit and 64-bit devices.

The Gradle plugin provides a separate task that creates a universal framework for iOS targets from several regular ones. The example below shows how to use this task. Note that the fat framework must have the same base name as the initial frameworks.

```kotlin
import org.jetbrains.kotlin.gradle.tasks.FatFrameworkTask

kotlin {
    // Create and configure the targets.
    targets {
        iosArm32("ios32")
        iosArm64("ios64")

        configure([ios32, ios64]) {
            binaries.framework {
                baseName = "my_framework"
            }
        }
    }

    // Create a task building a fat framework.
    task debugFatFramework(type: FatFrameworkTask) {
        // The fat framework must have the same base name as the initial frameworks.
        baseName = "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")
        )
    }
}
```

```
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 {
            baseName = "my_framework"
        }
    }

    // Create a task building a fat framework.
    tasks.create("debugFatFramework", FatFrameworkTask::class) {
        // The fat framework must have the same base name as the initial frameworks.
        baseName = "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")
        )
    }
}
```

CInterop support

Since Kotlin/Native provides [interoperability with native languages](#), there is a DSL allowing one to configure this feature for a specific compilation.

A compilation can interact with several native libraries. Interoperability with each of them can be configured in the `cinterops` block of the compilation:

```
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")

                    // Additional directories to search headers listed in the
'headerFilter' def-file option.
                    // -headerFilterAdditionalSearchPrefix command line option analogue.
                    includeDirs.headerFilterOnly("path1", "path2")

                    // A shortcut for includeDirs.allHeaders.
                    includeDirs("include/directory", "another/directory")
                }

                anotherInterop { /* ... */ }
            }
        }
    }
}
```

```
kotlin {
    linuxX64 {  // Replace with a target you need.
        compilations.getByName("main") {
            val myInterop by cinterops.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 analogue 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 analogue.
                    headerFilterOnly("path1", "path2")
                }
                // A shortcut for includeDirs.allHeaders.
                includeDirs("include/directory", "another/directory")
            }
```

```
            val anotherInterop by cinterops.creating { /* ... */ }
        }
    }
}
```

Often it's necessary to specify target-specific linker options for a binary which uses a native
library. It can by done using the `linkerOpts` property of the binary. See the Configuring
binaries section for details.

Other

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. We 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 etc.). 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.

Note that 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: when you say:

```
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

Let's say we 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.

NOTE: we 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, we 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 etc).

Underscore for unused variables (since 1.1)

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 (since 1.1)

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!" }
```

Type Checks and Casts: 'is' and 'as'

`is` and `!is` Operators

We can check whether an object conforms to a given type at runtime by using 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 many cases, one does not 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 do not work when the compiler cannot guarantee that the variable cannot 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 is not possible. Thus, we call it *unsafe*. The unsafe cast in Kotlin is done by the infix operator `as` (see [operator precedence](#)):

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

Note that `null` cannot be cast to `String` as this type is not [nullable](#), i.e. 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 an exception being thrown, one can use a *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 hold no 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, which 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. Example:

```
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>>() // Breaks type safety!
```

Unchecked casts

As said above, type erasure makes checking actual type arguments of a generic type instance impossible at runtime, and 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>
```

The compiler produces a warning for the cast in the last line. The cast cannot be fully checked 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 could be 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 the 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.

This Expression

To denote the current *receiver*, we 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](#), or [extension function](#), or labeled [function literal with receiver](#)) we 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
            }
        }
    }
}
```

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)
```

I.e. if `a` is not `null`, it calls the `equals(Any?)` function, otherwise (i.e. `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.

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](#).

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 which are represented as primitive types at runtime (for example, `Int`), the `===` equality check is equivalent to the `==` check.

Operator overloading

Kotlin allows us to provide implementations for a predefined set of operators on our types. These operators have fixed symbolic representation (like `+` or `*`) and fixed [precedence](#). To implement an operator, we provide a [member function](#) or an [extension function](#) with a fixed name, for the corresponding type, i.e. left-hand side type for binary operations and argument type for unary ones. Functions that overload operators need to be marked with the `operator` modifier.

Further we describe the conventions that regulate operator overloading for different operators.

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`, i.e. 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`;

Note that 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:

```kotlin
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, e.g. `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 `a.inc()` to `a`;
— Return `a0` as a result of the expression.

For `a--` the steps are completely analogous.

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

— Assign the result of `a.inc()` to `a`;
— Return the new value of `a` as a result of the expression.

Binary operations

Arithmetic operators

Expression	Translated to
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b), a.mod(b) (deprecated)
a..b	a.rangeTo(b)

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

Note that the `rem` operator is supported since Kotlin 1.1. Kotlin 1.0 uses the `mod` operator, which is deprecated in Kotlin 1.1.

Example

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

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

'In' operator

Expression	Translated to
a in b	b.contains(a)
a !in b	!b.contains(a)

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

Indexed access operator

Expression	Translated to
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, ..., i_n] = b	a.set(i_1, ..., i_n, b)

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

Invoke operator

Expression	Translated to
a()	a.invoke()
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ..., i_n)	a.invoke(i_1, ..., i_n)

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

Augmented assignments

Expression	Translated to
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.remAssign(b), a.modAssign(b) (deprecated)

For the assignment operations, e.g. `a += b`, the compiler performs the following steps:

— If the function from the right column is available

  — If the corresponding binary function (i.e. `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`).

Note: 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.

Note: `===` 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

We can simulate custom infix operations by using [infix function calls](#).

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, e.g. 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 can not (non-null references). For example, a regular variable of type `String` can not hold `null`:

```
var a: String = "abc"
a = null // compilation error
```

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

```
var b: String? = "abc"
b = null // ok
print(b)
```

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 we 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:

```
val b: String? = "Kotlin"
if (b != null && b.length > 0) {
    print("String of length ${b.length}")
} else {
    print("Empty string")
}
```

Note that this only works where `b` is immutable (i.e. 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 `?.`:

```
val a = "Kotlin"
val b: String? = null
println(b?.length)
println(a?.length) // Unnecessary safe call
```

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), we 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`:

```
val listWithNulls: List<String?> = listOf("Kotlin", null)
for (item in listWithNulls) {
    item?.let { println(it) } // prints Kotlin and ignores null
}
```

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 we have a nullable reference `r` , we can say "if `r` is not null, use it, otherwise use some non-null value `x` ":

```
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.

Note that, 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. We can write `b!!`, and this will return a non-null value of `b` (e.g., 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 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()
```

Exceptions

Exception Classes

All exception classes in Kotlin are descendants of the class `Throwable`. Every exception has a message, stack trace and an optional cause.

To throw an exception object, use the `throw`-expression:

```
throw Exception("Hi There!")
```

To catch an exception, use the `try`-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, i.e. it may 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 into 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 in [Does Java need 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)

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.

Annotations

Annotation Declaration

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 (classes, functions, properties, expressions etc.);

— @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 may 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 (since Kotlin 1.2) or `arrayOf(...)`:

```
// Java
public @interface AnnWithArrayMethod {
    String[] names();
}
```

```
// Kotlin 1.2+:
@AnnWithArrayMethod(names = ["abc", "foo", "bar"])
class C

// Older Kotlin versions:
@AnnWithArrayMethod(names = arrayOf("abc", "foo", "bar"))
class D
```

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
}
```

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 (i.e. learning a name or a type of a property or function at runtime) is closely intertwined with simply using a functional or reactive style.

> On the Java platform, the runtime component required for using the reflection features is distributed as a separate JAR file (`kotlin-reflect.jar`). This is done to reduce the required size of the runtime library for applications that do not use reflection features. If you do use reflection, please make sure that the .jar file is added to the classpath of your project.

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](#).

Note that 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 (since 1.1)

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 we have a named function declared like this:

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

We can easily call it directly (`isOdd(5)`), but we can also use it as a function type value, e.g. pass it to another function. To do this, we use the `::` operator:

```kotlin
val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd))
```

Here `::isOdd` is a value of function type `(Int) -> Boolean`.

Function references belong to one of the [KFunction<out R>](#) subtypes, depending on the parameter count, e.g. `KFunction3<T1, T2, T3, R>`.

`::` can be used with overloaded functions when the expected type is known from the context. For example:

```kotlin
fun isOdd(x: Int) = x % 2 != 0
fun isOdd(s: String) = s == "brillig" || s == "slithy" || s == "tove"

val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd)) // refers to isOdd(x: Int)
```

Alternatively, you can provide the necessary context by storing the method reference in a variable with an explicitly specified type:

```kotlin
val predicate: (String) -> Boolean = ::isOdd   // refers to isOdd(x: String)
```

If we need to use a member of a class, or an extension function, it needs to be qualified, e.g. `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:

```kotlin
val isEmptyStringList: List<String>.() -> Boolean = List<String>::isEmpty
```

Example: Function Composition

Consider the following function:

```kotlin
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 length(s: String) = s.length

val oddLength = compose(::isOdd, ::length)
val strings = listOf("a", "ab", "abc")

println(strings.filter(oddLength))
```

Property References

To access properties as first-class objects in Kotlin, we can also 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>`, which allows us to read its value using `get()` or retrieve the property name using the `name` property. For more information, please refer to the docs on the KProperty class.

For a mutable property, e.g. `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:

```
val strs = listOf("a", "bc", "def")
println(strs.map(String::length))
```

To access a property that is a member of a class, we qualify it:

```
class A(val p: Int)
val prop = A::p
println(prop.get(A(1)))
```

For an extension property:

```
val String.lastChar: Char
    get() = this[length - 1]

fun main() {
    println(String::lastChar.get("abc"))
}
```

Interoperability With Java Reflection

On the Java 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 say 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. They can be used 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, we can simply 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 (since 1.1)

You can refer to an instance method of a particular object:

```
val numberRegex = "\\d+".toRegex()
println(numberRegex.matches("29"))

val isNumber = numberRegex::matches
println(isNumber("29"))
```

Instead of calling the method `matches` directly we are storing 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:

```
val numberRegex = "\\d+".toRegex()
val strings = listOf("abc", "124", "a70")
println(strings.filter(numberRegex::matches))
```

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:

```
val prop = "abc"::length
println(prop.get())
```

Since Kotlin 1.2, explicitly specifying `this` as the receiver is not necessary: `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
```

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:

```
Person("Alice", 20, "Amsterdam").let {
    println(it)
    it.moveTo("London")
    it.incrementAge()
    println(it)
}
```

If you write the same without `let`, you'll have to introduce a new variable and repeat its name whenever you use it.

```
val alice = Person("Alice", 20, "Amsterdam")
println(alice)
alice.moveTo("London")
alice.incrementAge()
println(alice)
```

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.

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.

```kotlin
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.

```kotlin
val adam = Person("Adam").apply {
    age = 20                          // same as this.age = 20 or adam.age = 20
    city = "London"
}
```

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.

```kotlin
fun getRandomInt(): Int {
    return Random.nextInt(100).also {
        writeToLog("getRandomInt() generated value $it")
    }
}

val i = getRandomInt()
```

Additionally, when you pass the context object as an argument, you can provide a custom name for the context object inside the scope.

```kotlin
fun getRandomInt(): Int {
    return Random.nextInt(100).also { value ->
        writeToLog("getRandomInt() generated value $value")
    }
}

val i = getRandomInt()
```

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.

```kotlin
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()
```

They also can be used in return statements of functions returning the context object.

```kotlin
fun getRandomInt(): Int {
    return Random.nextInt(100).also {
        writeToLog("getRandomInt() generated value $it")
    }
}

val i = getRandomInt()
```

Lambda result

`let`, `run`, and `with` return the lambda result. So, you can use them when assigning the result to a variable, chaining operations on the result, and so on.

```
val numbers = mutableListOf("one", "two", "three")
val countEndsWithE = numbers.run {
    add("four")
    add("five")
    count { it.endsWith("e") }
}
println("There are $countEndsWithE elements that end with e.")
```

Additionally, you can ignore the return value and use a scope function to create a temporary scope for variables.

```
val numbers = mutableListOf("one", "two", "three")
with(numbers) {
    val firstItem = first()
    val lastItem = last()
    println("First item: $firstItem, last item: $lastItem")
}
```

Functions

To help you choose the right scope function for your 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:

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
val resultList = numbers.map { it.length }.filter { it > 3 }
println(resultList)
```

With `let`, you can rewrite it:

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.map { it.length }.filter { it > 3 }.let {
    println(it)
    // and more function calls if needed
}
```

If the code block contains a single function with `it` as an argument, you can use the method reference (`::`) instead of the lambda:

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.map { it.length }.filter { it > 3 }.let(::println)
```

`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.

```
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
}
```

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`.

```
val numbers = listOf("one", "two", "three", "four")
val modifiedFirstItem = numbers.first().let { firstItem ->
    println("The first item of the list is '$firstItem'")
    if (firstItem.length >= 5) firstItem else "!" + firstItem + "!"
}.toUpperCase()
println("First item after modifications: '$modifiedFirstItem'")
```

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."*

```
val numbers = mutableListOf("one", "two", "three")
with(numbers) {
    println("'with' is called with argument $this")
    println("It contains $size elements")
}
```

Another use case for `with` is introducing a helper object whose properties or functions will be used for calculating a value.

```
val numbers = mutableListOf("one", "two", "three")
val firstAndLast = with(numbers) {
    "The first element is ${first()}," +
    " the last element is ${last()}"
}
println(firstAndLast)
```

run

The context object is available as a receiver (`this`). **The return value** is the lambda result.

`run` does the same as `with` but 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.

```
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}")
}
```

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.

```
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)
}
```

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.*"

```
val adam = Person("Adam").apply {
    age = 32
    city = "London"
}
```

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 additional actions that don't alter the object, such as logging or printing debug information. Usually, you can remove the calls of `also` from the call chain without breaking the program logic.

When you see `also` in the code, you can read it as "*and also do the following*".

```
val numbers = mutableListOf("one", "two", "three")
numbers
    .also { println("The list elements before adding new one: $it") }
    .add("four")
```

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

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` .

`takeIf` and `takeUnless`

In addition to scope functions, the standard library contains the functions `takeIf` and `takeUnless`. These functions let you embed checks of the object state in call chains.

When called on an object with a predicate provided, `takeIf` returns this object if it matches the predicate. Otherwise, it returns `null`. So, `takeIf` 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`).

```
val number = Random.nextInt(100)

val evenOrNull = number.takeIf { it % 2 == 0 }
val oddOrNull = number.takeUnless { it % 2 == 0 }
println("even: $evenOrNull, odd: $oddOrNull")
```

When chaining other functions after `takeIf` and `takeUnless`, don't forget to perform the null check or the safe call (`?.`) because their return value is nullable.

```
val str = "Hello"
val caps = str.takeIf { it.isNotEmpty() }?.toUpperCase()
/val caps = str.takeIf { it.isNotEmpty() }.toUpperCase() //compilation error
println(caps)
```

`takeIf` 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 `takeIf` on the object and then call `let` with a safe call (`?`). For objects that don't match the predicate, `takeIf` returns `null` and `let` isn't invoked.

```
fun 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")
```

This is how the same function looks without the standard library functions:

```
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")
```

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. Some of the example 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](#).

A type-safe builder example

Consider the following code:

```kotlin
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 = "http://kotlinlang.org") {+"Kotlin"}

            // mixed content
            p {
                +"This is some"
                b {+"mixed"}
```

```
                +"text. For more see the"
                a(href = "http://kotlinlang.org") {+"Kotlin"}
                +"project"
            }
            p {+"some text"}

            // 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

Let's walk through the mechanisms of implementing type-safe builders in Kotlin. First of all, we need to define the model we want to build, in this case we 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, i.e. it defines children like `<head>` and `<body>`. (See its declaration below.)

Now, let's recall why we 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 we need to pass an instance of type `HTML` (a *receiver*) to the function, and we 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 we 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 our 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 we 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 our functions are very simple:

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)

fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

And we can use them to build `<head>` and `<body>` tags.

One other thing to be discussed here is how we add text to tag bodies. In the example above we say something like:

```
html {
    head {
        title {+"XML encoding with Kotlin"}
    }
    // ...
}
```

So basically, we just put a string inside a tag body, but there is this little `+` in front of it, so it is a function call that invokes a prefix `unaryPlus()` operation. That operation is actually defined by an extension function `unaryPlus()` that is a member of the `TagWithText` abstract class (a parent of `Title`):

```
operator fun String.unaryPlus() {
    children.add(TextElement(this))
}
```

So, what the prefix `+` does here is wrapping a string into an instance of `TextElement` and adding it to the `children` collection, so that it becomes a proper part of the tag tree.

All this is defined in a package `com.example.html` that is imported at the top of the builder example above. In the last section you can read through the full definition of this package.

Scope control: @DslMarker (since 1.1)

When using DSLs, one might have come across the problem that too many functions can be called in the context. We 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, in Kotlin 1.1 a special mechanism to control receiver scope was introduced.

To make the compiler start controlling scopes we only have to annotate the types of all receivers used in the DSL with the same marker annotation. For instance, for HTML Builders we 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) { ... }
```

We 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 we'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](#).

Note that the `@DslMarker` annotation is available only since Kotlin 1.1.

```
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>()
```

```kotlin
    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")
```

```kotlin
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
}
```

Experimental API Markers

The Kotlin standard library provides developers with a mechanism for creating and using *experimental* APIs. This mechanism lets library authors inform users that certain components of their API, such as classes or functions, are unstable and are likely to change in the future. Such changes may require rewriting and recompiling the client code. To prevent potential compatibility issues, the compiler warns users of the experimental status of such APIs and may require them to give their explicit consent to use the API.

Using experimental APIs

If a class or a function from a library is marked by its author as experimental, using it in your code will produce warnings or compilation errors unless you explicitly accept their experimental status. There are several ways to accept the experimental status of API elements; all of them are applicable without technical limitations. You are free to choose the way that you find best for your situation.

Propagating use

When you use an experimental API in the code intended for third-party use (a library), you can mark your API as experimental as well. To do this, annotate your declaration with the *experimental marker annotation* of the API used in its body. This enables you to use the API elements annotated with this marker.

```
// library code
@Experimental
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime            // Experimental API marker

@ExperimentalDateTime
class DateProvider                               // Experimental class
```

```
// client code
fun getYear(): Int {
    val dateProvider: DateProvider // error: DateProvider is experimental
    // ...
}

@ExperimentalDateTime
fun getDate(): Date {
    val dateProvider: DateProvider // OK: the function is marked as experimental
    // ...
}

fun displayDate() {
    println(getDate()) // error: getDate() is experimental, acceptance is required
}
```

As you can see in this example, the annotated function appears to be a part of the
`@ExperimentalDateTime` experimental API. So, the described way of acceptance propagates
the experimental status to the code that uses an experimental API; its clients will be required to
accept it as well. To use multiple experimental APIs, annotate the declaration with all their
markers.

Non-propagating use

In modules that don't provide their own API, such as application modules, you can use
experimental APIs without propagating the experimental status to your code. In this case, mark
your code with the @UseExperimental(Marker::class) annotation specifying the marker
annotation of the experimental API:

```
// library code
@Experimental
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime          // Experimental API marker

@ExperimentalDateTime
class DateProvider                              // Experimental class
```

```
//client code
@UseExperimental(ExperimentalDateTime::class)
fun getDate(): Date {              // uses DateProvider; doesn't expose the experimental
status
    val dateProvider: DateProvider
    // ...
}

fun displayDate() {
    println(getDate())                          // OK: getDate() is not experimental
}
```

When somebody calls the function `getDate()`, they won't be informed about the experimental
API used in its body.

To use an experimental API in all functions and classes in a file, add the file-level annotation `@file:UseExperimental` to the top of the file before the package specification and imports.

```
//client code
@file:UseExperimental(ExperimentalDateTime::class)
```

Module-wide use

If you don't want to annotate every usage of experimental APIs in your code, you can accept the experimental status for your whole module. Module-wide use of experimental APIs can be propagating and non-propagating as well:

— To accept the experimental status without propagation, compile the module with the argument `-Xuse-experimental`, specifying the fully qualified name of the experimental API marker you use: `-Xuse-experimental=org.mylibrary.ExperimentalMarker`. Compiling with this argument has the same effect as if every declaration in the module had the annotation `@UseExperimental(ExperimentalMarker::class)`.

— To accept and propagate the experimental status to your whole module, compile the module with the argument `-Xexperimental=org.mylibrary.ExperimentalMarker`. In this case, *every declaration* in the module becomes experimental. The use of the module requires the acceptance of its experimental status as well.

If you build your module with Gradle, you can add arguments like this:

```
compileKotlin {
    kotlinOptions {
        freeCompilerArgs += "-Xuse-experimental=org.mylibrary.ExperimentalMarker"
    }
}
```

```
tasks.withType<KotlinCompile>().all {
    kotlinOptions.freeCompilerArgs += "-Xuse-
experimental=org.mylibrary.ExperimentalMarker"
}
```

If your Gradle module is a multiplatform module, use the `useExperimentalAnnotation` method:

```
sourceSets {
    all {
        languageSettings {
            useExperimentalAnnotation('kotlin.Experimental')
        }
    }
}
```

```
sourceSets {
    all {
        languageSettings.useExperimentalAnnotation("kotlin.Experimental")
    }
}
```

For Maven, it would be:

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <executions>...</executions>
            <configuration>
                <args>
                    <arg>-Xuse-experimental=org.mylibrary.ExperimentalMarker</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

To accept the usage of multiple experimental APIs on the module level, add one of the described arguments for each experimental API marker used in your module.

Marking experimental API

Creating marker annotations

If you want to declare your module's API as experimental, create an annotation class to use as its *experimental marker*. This class must be annotated with @Experimental:

```
@Experimental
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime
```

Experimental marker annotations must meet several requirements:

— `BINARY` retention

— No `EXPRESSION` and `FILE` among targets

— No parameters.

A marker annotation can have one of two severity levels of informing about experimental API usage:

— `Experimental.Level.ERROR`. Acceptance is mandatory. Otherwise, the code that uses marked API won't compile. This level is used by default.

— `Experimental.Level.WARNING`. Acceptance is not mandatory, but advisable. Without it,

the compiler raises a warning. To set the desired level, specify the `level` parameter of the `@Experimental` annotation.

```
@Experimental(level = Experimental.Level.WARNING)
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime
```

If you publish several features in the experimental state, declare a marker for each. Separate markers make the use of experimental features safer for your clients: they can use only the features that they explicitly accept. This also lets you graduate the features to stable independently.

Marking API elements

To mark an API element as experimental, annotate its declaration with your experimental marker annotation:

```
@ExperimentalDateTime
class DateProvider

@ExperimentalDateTime
fun getTime(): Time {}
```

Module-wide markers

If you consider all the APIs of your module experimental, you can mark the entire module as such with the compiler argument `-Xexperimental` as described in Module-wide use.

Graduation of experimental API

Once your experimental API graduates and is released in its final state, remove its marker annotation from declarations so that the clients can use it without restriction. However, you should leave the marker classes in modules so that the existing client code remains compatible. To let the API users update their modules accordingly (remove the markers from their code and recompile), mark the annotations as @Deprecated and provide the explanation in its message.

```
@Deprecated("This experimental API marker is not used anymore. Remove its usages from
your code.")
@Experimental
annotation class ExperimentalDateTime
```

Experimental status of experimental API markers

The described mechanism for marking and using experimental APIs is itself experimental in Kotlin 1.3. This means that in future releases it may be changed in ways that make it incompatible. To make the users of annotations `@Experimental` and `UseExperimental` 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 '-Xuse-
experimental=kotlin.Experimental'
```

To remove the warnings, add the compiler argument `-Xuse-experimental=kotlin.Experimental`.

Reference

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

320

— `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](#)

— `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](#)

Grammar

Description

Notation

The notation used on this page corresponds to the ANTLR 4 notation with a few exceptions for better readability:

— omitted lexer rule actions and commands,

— omitted lexical modes.

Short description:

— operator `|` denotes *alternative*,

— operator `*` denotes *iteration* (zero or more),

— operator `+` denotes *iteration* (one or more),

— operator `?` denotes *option* (zero or one),

— operator `..` denotes *range* (from left to right),

— operator `~` denotes *negation*.

Grammar source files

Kotlin grammar source files (in ANTLR format) are located in the Kotlin specification repository:

— **KotlinLexer.g4** describes lexical structure;

— **UnicodeClasses.g4** describes the characters that can be used in identifiers (these rules are omitted on this page for better readability);

— **KotlinParser.g4** describes syntax.

The grammar on this page corresponds to the grammar files above.

Symbols and naming

Terminal symbol names start with an uppercase letter, e.g. Identifier.
Non-terminal symbol names start with a lowercase letter, e.g. kotlinFile.

Symbol definitions may be documented with *attributes*:

— `start` attribute denotes a symbol that represents the whole source file (see kotlinFile and script),

— `helper` attribute denotes a lexer fragment rule (used only inside other terminal symbols).

Also for better readability some simplifications are made:

— lexer rules consisting of one string literal element are inlined to the use site,

— new line tokens are excluded (new lines are not allowed in some places, see source grammar files for details).

Scope

The grammar corresponds to the latest stable version of the Kotlin compiler excluding lexer and parser rules for experimental features that are disabled by default.

Syntax grammar

General

Relevant pages: Packages

```
start
kotlinFile
  : shebangLine? fileAnnotation* packageHeader importList topLevelObject* EOF
  ;
start
script
  : shebangLine? fileAnnotation* packageHeader importList (statement semi)* EOF
  ;
shebangLine
(used by kotlinFile, script)
  : ShebangLine
  ;
fileAnnotation
(used by kotlinFile, script)
  : ('@' | AT_PRE_WS) 'file' ':' (('[' unescapedAnnotation+ ']') | unescapedAnnotation)
  ;
```

See Packages

```
packageHeader
(used by kotlinFile, script)
  : ('package' identifier semi?)?
  ;
```

See Imports

```
importList
(used by kotlinFile, script)
  : importHeader*
  ;
importHeader
(used by importList)
  : 'import' identifier (('.' '*') | importAlias)? semi?
  ;
importAlias
(used by importHeader)
  : 'as' simpleIdentifier
  ;
topLevelObject
(used by kotlinFile)
  : declaration semis?
  ;
typeAlias
(used by declaration)
  : modifiers? 'typealias' simpleIdentifier typeParameters? '=' type
  ;
declaration
(used by topLevelObject, classMemberDeclaration, statement)
  : classDeclaration
  | objectDeclaration
  | functionDeclaration
  | propertyDeclaration
  | typeAlias
  ;
```

Classes

See Classes and Inheritance

classDeclaration
(used by declaration)
  : modifiers? ('class' | 'interface')
    simpleIdentifier typeParameters?
    primaryConstructor?
    (':' delegationSpecifiers)?
    typeConstraints?
    (classBody | enumClassBody)?
  ;
primaryConstructor
(used by classDeclaration)
  : (modifiers? 'constructor')? classParameters
  ;
classBody
(used by classDeclaration, companionObject, objectDeclaration, enumEntry, objectLiteral)
  : '{' classMemberDeclarations '}'
  ;
classParameters
(used by primaryConstructor)
  : '(' (classParameter (',' classParameter)*)? ')'
  ;
classParameter
(used by classParameters)
  : modifiers? ('val' | 'var')? simpleIdentifier ':' type ('=' expression)?
  ;
delegationSpecifiers
(used by classDeclaration, companionObject, objectDeclaration, objectLiteral)
  : annotatedDelegationSpecifier (',' annotatedDelegationSpecifier)*
  ;
delegationSpecifier
(used by annotatedDelegationSpecifier)
  : constructorInvocation
  | explicitDelegation
  | userType
  | functionType
  ;
constructorInvocation
(used by delegationSpecifier, unescapedAnnotation)
  : userType valueArguments
  ;
annotatedDelegationSpecifier
(used by delegationSpecifiers)
  : annotation* delegationSpecifier
  ;
explicitDelegation
(used by delegationSpecifier)
  : (userType | functionType) 'by' expression
  ;

See Generic classes

typeParameters
(used by typeAlias, classDeclaration, functionDeclaration, propertyDeclaration)
  : '<' typeParameter (',' typeParameter)* '>'
  ;
typeParameter
(used by typeParameters)
  : typeParameterModifiers? simpleIdentifier (':' type)?
  ;

See Generic constraints

typeConstraints
(used by classDeclaration, functionDeclaration, propertyDeclaration, anonymousFunction)
  : 'where' typeConstraint (',' typeConstraint)*
  ;
typeConstraint
(used by typeConstraints)
  : annotation* simpleIdentifier ':' type
  ;

Class members

classMemberDeclarations
(used by classBody, enumClassBody)
  : (classMemberDeclaration semis?)*
  ;
classMemberDeclaration
(used by classMemberDeclarations)
  : declaration
  | companionObject
  | anonymousInitializer
  | secondaryConstructor
  ;
anonymousInitializer
(used by classMemberDeclaration)
  : 'init' block
  ;
companionObject
(used by classMemberDeclaration)
  : modifiers? 'companion' 'object' simpleIdentifier?
    (':' delegationSpecifiers)?
    classBody?
  ;
functionValueParameters
(used by functionDeclaration, secondaryConstructor)
  : '(' (functionValueParameter (',' functionValueParameter)*)? ')'
  ;
functionValueParameter
(used by functionValueParameters)
  : parameterModifiers? parameter ('=' expression)?
  ;
functionDeclaration
(used by declaration)
  : modifiers? 'fun' typeParameters?
    (receiverType '.')?
    simpleIdentifier functionValueParameters
    (':' type)? typeConstraints?
    functionBody?
  ;
functionBody
(used by functionDeclaration, getter, setter, anonymousFunction)
  : block
  | '=' expression
  ;
variableDeclaration
(used by multiVariableDeclaration, propertyDeclaration, forStatement, lambdaParameter, whenSubject)
  : annotation* simpleIdentifier (':' type)?
  ;
multiVariableDeclaration
(used by propertyDeclaration, forStatement, lambdaParameter)
  : '(' variableDeclaration (',' variableDeclaration)* ')'
  ;

See Properties and Fields

propertyDeclaration
(used by declaration)
  : modifiers? ('val' | 'var') typeParameters?
    (receiverType '.')?
    (multiVariableDeclaration | variableDeclaration)
    typeConstraints?
    (('=' expression) | propertyDelegate)? ';'?
    ((getter? (semi? setter)?) | (setter? (semi? getter)?))
  ;
propertyDelegate
(used by propertyDeclaration)
  : 'by' expression
  ;
getter
(used by propertyDeclaration)
  : modifiers? 'get'
  | modifiers? 'get' '(' ')'
    (':' type)?
    functionBody
  ;
setter
(used by propertyDeclaration)
  : modifiers? 'set'
  | modifiers? 'set' '(' parameterWithOptionalType ')' (':' type)?
    functionBody
  ;
parametersWithOptionalType
(used by anonymousFunction)

```
  : '(' (parameterWithOptionalType (',' parameterWithOptionalType)*)? ')'
  ;
parameterWithOptionalType
(used by setter, parametersWithOptionalType)
  : parameterModifiers? simpleIdentifier (':' type)?
  ;
parameter
(used by functionValueParameter, functionTypeParameters)
  : simpleIdentifier ':' type
  ;
```

See Object expressions and Declarations

```
objectDeclaration
(used by declaration)
  : modifiers? 'object' simpleIdentifier (':' delegationSpecifiers)? classBody?
  ;
secondaryConstructor
(used by classMemberDeclaration)
  : modifiers? 'constructor' functionValueParameters
    (':' constructorDelegationCall)? block?
  ;
constructorDelegationCall
(used by secondaryConstructor)
  : 'this' valueArguments
  | 'super' valueArguments
  ;
```

Enum classes

See Enum classes

```
enumClassBody
(used by classDeclaration)
  : '{' enumEntries? (';' classMemberDeclarations)? '}'
  ;
enumEntries
(used by enumClassBody)
  : enumEntry (',' enumEntry)* ','?
  ;
enumEntry
(used by enumEntries)
  : modifiers? simpleIdentifier valueArguments? classBody?
  ;
```

Types

See Types

```
type
(used by typeAlias, classParameter, typeParameter, typeConstraint, functionDeclaration, variableDeclaration,
getter, setter, parameterWithOptionalType, parameter, typeProjection, functionType, functionTypeParameters,
parenthesizedType, infixOperation, asExpression, lambdaParameter, anonymousFunction, superExpression,
typeTest, catchBlock)
  : typeModifiers? (parenthesizedType | nullableType | typeReference | functionType)
  ;
typeReference
(used by type, nullableType, receiverType)
  : userType
  | 'dynamic'
  ;
nullableType
(used by type, receiverType)
  : (typeReference | parenthesizedType) quest+
  ;
quest
(used by nullableType)
  : '?'
  | QUEST_WS
  ;
userType
(used by delegationSpecifier, constructorInvocation, explicitDelegation, typeReference,
parenthesizedUserType, unescapedAnnotation)
  : simpleUserType ('.' simpleUserType)*
  ;
```

simpleUserType
(used by userType)
  : simpleIdentifier typeArguments?
  ;
typeProjection
(used by typeArguments)
  : typeProjectionModifiers? type
  | '*'
  ;
typeProjectionModifiers
(used by typeProjection)
  : typeProjectionModifier+
  ;
typeProjectionModifier
(used by typeProjectionModifiers)
  : varianceModifier
  | annotation
  ;
functionType
(used by delegationSpecifier, explicitDelegation, type)
  : (receiverType '.')? functionTypeParameters '->' type
  ;
functionTypeParameters
(used by functionType)
  : '(' (parameter | type)? (',' (parameter | type))* ')'
  ;
parenthesizedType
(used by type, nullableType, receiverType)
  : '(' type ')'
  ;
receiverType
(used by functionDeclaration, propertyDeclaration, functionType, callableReference)
  : typeModifiers? (parenthesizedType | nullableType | typeReference)
  ;
parenthesizedUserType
(used by parenthesizedUserType)
  : '(' userType ')'
  | '(' parenthesizedUserType ')'
  ;

Statements

statements
(used by block, lambdaLiteral)
  : (statement (semis statement)* semis?)?
  ;
statement
(used by script, statements, controlStructureBody)
  : (label | annotation)* (declaration | assignment | loopStatement | expression)
  ;
See Returns and jumps

label
(used by statement, unaryPrefix, annotatedLambda)
  : simpleIdentifier ('@' | AT_POST_WS)
  ;
controlStructureBody
(used by forStatement, whileStatement, doWhileStatement, ifExpression, whenEntry)
  : block
  | statement
  ;
block
(used by anonymousInitializer, functionBody, secondaryConstructor, controlStructureBody, tryExpression, catchBlock, finallyBlock)
  : '{' statements '}'
  ;
loopStatement
(used by statement)
  : forStatement
  | whileStatement
  | doWhileStatement
  ;
forStatement
(used by loopStatement)
  : 'for'
    '(' annotation* (variableDeclaration | multiVariableDeclaration) 'in' expression ')'
    controlStructureBody?
  ;

whileStatement
(used by loopStatement)
  : 'while' '(' expression ')' controlStructureBody
  | 'while' '(' expression ')' ';'
  ;
doWhileStatement
(used by loopStatement)
  : 'do' controlStructureBody? 'while' '(' expression ')'
  ;
assignment
(used by statement)
  : directlyAssignableExpression '=' expression
  | assignableExpression assignmentAndOperator expression
  ;
semi
(used by script, packageHeader, importHeader, propertyDeclaration, whenEntry)
  : EOF
  ;
semis
(used by topLevelObject, classMemberDeclarations, statements)
  : EOF
  ;

Expressions

Precedence	Title	Symbols
Highest	Postfix	`++, --, ., ?., ?`
	Prefix	`-, +, ++, --, !,` label
	Type RHS	`:, as, as?`
	Multiplicative	`*, /, %`
	Additive	`+, -`
	Range	`..`
	Infix function	`simpleIdentifier`
	Elvis	`?:`
	Named checks	`in, !in, is, !is`
	Comparison	`<, >, <=, >=`
	Equality	`==, !==`
	Conjunction	`&&`
	Disjunction	`||`
	Spread operator	`*`
Lowest	Assignment	`=, +=, -=, *=, /=, %=`

expression
(used by classParameter, explicitDelegation, functionValueParameter, functionBody, propertyDeclaration, propertyDelegate, statement, forStatement, whileStatement, doWhileStatement, assignment, indexingSuffix, valueArgument, parenthesizedExpression, collectionLiteral, lineStringExpression, multiLineStringExpression, ifExpression, whenSubject, whenCondition, rangeTest, jumpExpression)
  : disjunction
  ;
disjunction
(used by expression)
  : conjunction ('||' conjunction)*
  ;
conjunction
(used by disjunction)
  : equality ('&&' equality)*
  ;
equality
(used by conjunction)
  : comparison (equalityOperator comparison)*
  ;
comparison
(used by equality)
  : infixOperation (comparisonOperator infixOperation)?
  ;

infixOperation
(used by comparison)
  : elvisExpression ((inOperator elvisExpression) | (isOperator type))*
  ;
elvisExpression
(used by infixOperation)
  : infixFunctionCall (elvis infixFunctionCall)*
  ;
elvis
(used by elvisExpression)
  : '?' ':'
  ;
infixFunctionCall
(used by elvisExpression)
  : rangeExpression (simpleIdentifier rangeExpression)*
  ;
rangeExpression
(used by infixFunctionCall)
  : additiveExpression ('..' additiveExpression)*
  ;
additiveExpression
(used by rangeExpression)
  : multiplicativeExpression (additiveOperator multiplicativeExpression)*
  ;
multiplicativeExpression
(used by additiveExpression)
  : asExpression (multiplicativeOperator asExpression)*
  ;
asExpression
(used by multiplicativeExpression)
  : prefixUnaryExpression (asOperator type)?
  ;
prefixUnaryExpression
(used by asExpression, assignableExpression)
  : unaryPrefix* postfixUnaryExpression
  ;
unaryPrefix
(used by prefixUnaryExpression)
  : annotation
  | label
  | prefixUnaryOperator
  ;
postfixUnaryExpression
(used by prefixUnaryExpression, directlyAssignableExpression)
  : primaryExpression
  | primaryExpression postfixUnarySuffix+
  ;
postfixUnarySuffix
(used by postfixUnaryExpression)
  : postfixUnaryOperator
  | typeArguments
  | callSuffix
  | indexingSuffix
  | navigationSuffix
  ;
directlyAssignableExpression
(used by assignment, parenthesizedDirectlyAssignableExpression)
  : postfixUnaryExpression assignableSuffix
  | simpleIdentifier
  | parenthesizedDirectlyAssignableExpression
  ;
parenthesizedDirectlyAssignableExpression
(used by directlyAssignableExpression)
  : '(' directlyAssignableExpression ')'
  ;
assignableExpression
(used by assignment, parenthesizedAssignableExpression)
  : prefixUnaryExpression
  | parenthesizedAssignableExpression
  ;
parenthesizedAssignableExpression
(used by assignableExpression)
  : '(' assignableExpression ')'
  ;
assignableSuffix
(used by directlyAssignableExpression)
  : typeArguments
  | indexingSuffix
  | navigationSuffix
  ;
indexingSuffix

333

(used by postfixUnarySuffix, assignableSuffix)
  : '[' expression (',' expression)* ']'
  ;
navigationSuffix
(used by postfixUnarySuffix, assignableSuffix)
  : memberAccessOperator (simpleIdentifier | parenthesizedExpression | 'class')
  ;
callSuffix
(used by postfixUnarySuffix)
  : typeArguments? valueArguments? annotatedLambda
  | typeArguments? valueArguments
  ;
annotatedLambda
(used by callSuffix)
  : annotation* label? lambdaLiteral
  ;
typeArguments
(used by simpleUserType, postfixUnarySuffix, assignableSuffix, callSuffix)
  : '<' typeProjection (',' typeProjection)* '>'
  ;
valueArguments
(used by constructorInvocation, constructorDelegationCall, enumEntry, callSuffix)
  : '(' ')'
  | '(' valueArgument (',' valueArgument)* ')'
  ;
valueArgument
(used by valueArguments)
  : annotation? (simpleIdentifier '=')? '*'? expression
  ;
primaryExpression
(used by postfixUnaryExpression)
  : parenthesizedExpression
  | simpleIdentifier
  | literalConstant
  | stringLiteral
  | callableReference
  | functionLiteral
  | objectLiteral
  | collectionLiteral
  | thisExpression
  | superExpression
  | ifExpression
  | whenExpression
  | tryExpression
  | jumpExpression
  ;
parenthesizedExpression
(used by navigationSuffix, primaryExpression)
  : '(' expression ')'
  ;
collectionLiteral
(used by primaryExpression)
  : '[' expression (',' expression)* ']'
  | '[' ']'
  ;
literalConstant
(used by primaryExpression)
  : BooleanLiteral
  | IntegerLiteral
  | HexLiteral
  | BinLiteral
  | CharacterLiteral
  | RealLiteral
  | 'null'
  | LongLiteral
  | UnsignedLiteral
  ;
stringLiteral
(used by primaryExpression)
  : lineStringLiteral
  | multiLineStringLiteral
  ;
lineStringLiteral
(used by stringLiteral)
  : '"' (lineStringContent | lineStringExpression)* '"'
  ;
multiLineStringLiteral
(used by stringLiteral)
  : '"""' (multiLineStringContent | multiLineStringExpression | '"')*
    TRIPLE_QUOTE_CLOSE
  ;

lineStringContent
(used by lineStringLiteral)
 : LineStrText
 | LineStrEscapedChar
 | LineStrRef
 ;
lineStringExpression
(used by lineStringLiteral)
 : '${' expression '}'
 ;
multiLineStringContent
(used by multiLineStringLiteral)
 : MultiLineStrText
 | '""'
 | MultiLineStrRef
 ;
multiLineStringExpression
(used by multiLineStringLiteral)
 : '${' expression '}'
 ;
lambdaLiteral
(used by annotatedLambda, functionLiteral)
 : '{' statements '}'
 | '{' lambdaParameters? '->' statements '}'
 ;
lambdaParameters
(used by lambdaLiteral)
 : lambdaParameter (',' lambdaParameter)*
 ;
lambdaParameter
(used by lambdaParameters)
 : variableDeclaration
 | multiVariableDeclaration (':' type)?
 ;
anonymousFunction
(used by functionLiteral)
 : 'fun' (type '.')? parametersWithOptionalType
   (':' type)? typeConstraints?
   functionBody?
 ;
functionLiteral
(used by primaryExpression)
 : lambdaLiteral
 | anonymousFunction
 ;
objectLiteral
(used by primaryExpression)
 : 'object' ':' delegationSpecifiers classBody
 | 'object' classBody
 ;
thisExpression
(used by primaryExpression)
 : 'this'
 | THIS_AT
 ;
superExpression
(used by primaryExpression)
 : 'super' ('<' type '>')? ('@' simpleIdentifier)?
 | SUPER_AT
 ;
ifExpression
(used by primaryExpression)
 : 'if' '(' expression ')'
   (controlStructureBody | ';')
 | 'if' '(' expression ')'
   controlStructureBody? ';'? 'else' (controlStructureBody | ';')
 ;
whenSubject
(used by whenExpression)
 : '(' (annotation* 'val' variableDeclaration '=')? expression ')'
 ;
whenExpression
(used by primaryExpression)
 : 'when' whenSubject? '{' whenEntry* '}'
 ;
whenEntry
(used by whenExpression)
 : whenCondition (',' whenCondition)* '->' controlStructureBody semi?
 | 'else' '->' controlStructureBody semi?
 ;
whenCondition

(used by whenEntry)
: expression
| rangeTest
| typeTest
;
rangeTest
(used by whenCondition)
: inOperator expression
;
typeTest
(used by whenCondition)
: isOperator type
;
tryExpression
(used by primaryExpression)
: 'try' block ((catchBlock+ finallyBlock?) | finallyBlock)
;
catchBlock
(used by tryExpression)
: 'catch' '(' annotation* simpleIdentifier ':' type ')' block
;
finallyBlock
(used by tryExpression)
: 'finally' block
;
jumpExpression
(used by primaryExpression)
: 'throw' expression
| ('return' | RETURN_AT) expression?
| 'continue'
| CONTINUE_AT
| 'break'
| BREAK_AT
;
callableReference
(used by primaryExpression)
: (receiverType? '::' (simpleIdentifier | 'class'))
;
assignmentAndOperator
(used by assignment)
: '+='
| '-='
| '*='
| '/='
| '%='
;
equalityOperator
(used by equality)
: '!='
| '!=='
| '=='
| '==='
;
comparisonOperator
(used by comparison)
: '<'
| '>'
| '<='
| '>='
;
inOperator
(used by infixOperation, rangeTest)
: 'in'
| NOT_IN
;
isOperator
(used by infixOperation, typeTest)
: 'is'
| NOT_IS
;
additiveOperator
(used by additiveExpression)
: '+'
| '-'
;
multiplicativeOperator
(used by multiplicativeExpression)
: '*'
| '/'
| '%'
;

336

asOperator
(used by asExpression)
 : 'as'
 | 'as?'
 ;
prefixUnaryOperator
(used by unaryPrefix)
 : '++'
 | '--'
 | '-'
 | '+'
 | excl
 ;
postfixUnaryOperator
(used by postfixUnarySuffix)
 : '++'
 | '--'
 | '!' excl
 ;
excl
(used by prefixUnaryOperator, postfixUnaryOperator)
 : '!'
 | EXCL_WS
 ;
memberAccessOperator
(used by navigationSuffix)
 : '.'
 | safeNav
 | '::'
 ;
safeNav
(used by memberAccessOperator)
 : '?' '.'
 ;

Modifiers

modifiers
(used by typeAlias, classDeclaration, primaryConstructor, classParameter, companionObject,
functionDeclaration, propertyDeclaration, getter, setter, objectDeclaration, secondaryConstructor,
enumEntry)
 : annotation
 | modifier+
 ;
parameterModifiers
(used by functionValueParameter, parameterWithOptionalType)
 : annotation
 | parameterModifier+
 ;
modifier
(used by modifiers)
 : classModifier
 | memberModifier
 | visibilityModifier
 | functionModifier
 | propertyModifier
 | inheritanceModifier
 | parameterModifier
 | platformModifier
 ;
typeModifiers
(used by type, receiverType)
 : typeModifier+
 ;
typeModifier
(used by typeModifiers)
 : annotation
 | 'suspend'
 ;
classModifier
(used by modifier)
 : 'enum'
 | 'sealed'
 | 'annotation'
 | 'data'
 | 'inner'
 ;
memberModifier
(used by modifier)

337

```
  : 'override'
  | 'lateinit'
  ;
visibilityModifier
(used by modifier)
  : 'public'
  | 'private'
  | 'internal'
  | 'protected'
  ;
varianceModifier
(used by typeProjectionModifier, typeParameterModifier)
  : 'in'
  | 'out'
  ;
typeParameterModifiers
(used by typeParameter)
  : typeParameterModifier+
  ;
typeParameterModifier
(used by typeParameterModifiers)
  : reificationModifier
  | varianceModifier
  | annotation
  ;
functionModifier
(used by modifier)
  : 'tailrec'
  | 'operator'
  | 'infix'
  | 'inline'
  | 'external'
  | 'suspend'
  ;
propertyModifier
(used by modifier)
  : 'const'
  ;
inheritanceModifier
(used by modifier)
  : 'abstract'
  | 'final'
  | 'open'
  ;
parameterModifier
(used by parameterModifiers, modifier)
  : 'vararg'
  | 'noinline'
  | 'crossinline'
  ;
reificationModifier
(used by typeParameterModifier)
  : 'reified'
  ;
platformModifier
(used by modifier)
  : 'expect'
  | 'actual'
  ;
```

Annotations

```
annotation
(used by annotatedDelegationSpecifier, typeConstraint, variableDeclaration, typeProjectionModifier,
statement, forStatement, unaryPrefix, annotatedLambda, valueArgument, whenSubject, catchBlock,
modifiers, parameterModifiers, typeModifier, typeParameterModifier)
  : singleAnnotation
  | multiAnnotation
  ;
singleAnnotation
(used by annotation)
  : annotationUseSiteTarget unescapedAnnotation
  | ('@' | AT_PRE_WS) unescapedAnnotation
  ;
multiAnnotation
(used by annotation)
  : annotationUseSiteTarget '[' unescapedAnnotation+ ']'
  | ('@' | AT_PRE_WS) '[' unescapedAnnotation+ ']'
  ;
```

annotationUseSiteTarget
(used by singleAnnotation, multiAnnotation)
  : ('@' | AT_PRE_WS)
    ('field' | 'property' | 'get' | 'set' | 'receiver' | 'param' | 'setparam' | 'delegate') ':'
  ;
unescapedAnnotation
(used by fileAnnotation, singleAnnotation, multiAnnotation)
  : constructorInvocation
  | userType
  ;

Identifiers

simpleIdentifier
(used by importAlias, typeAlias, classDeclaration, classParameter, typeParameter, typeConstraint,
companionObject, functionDeclaration, variableDeclaration, parameterWithOptionalType, parameter,
objectDeclaration, enumEntry, simpleUserType, label, infixFunctionCall, directlyAssignableExpression,
navigationSuffix, valueArgument, primaryExpression, superExpression, catchBlock, callableReference,
identifier)
  : Identifier
  | 'abstract'
  | 'annotation'
  | 'by'
  | 'catch'
  | 'companion'
  | 'constructor'
  | 'crossinline'
  | 'data'
  | 'dynamic'
  | 'enum'
  | 'external'
  | 'final'
  | 'finally'
  | 'get'
  | 'import'
  | 'infix'
  | 'init'
  | 'inline'
  | 'inner'
  | 'internal'
  | 'lateinit'
  | 'noinline'
  | 'open'
  | 'operator'
  | 'out'
  | 'override'
  | 'private'
  | 'protected'
  | 'public'
  | 'reified'
  | 'sealed'
  | 'tailrec'
  | 'set'
  | 'vararg'
  | 'where'
  | 'field'
  | 'property'
  | 'receiver'
  | 'param'
  | 'setparam'
  | 'delegate'
  | 'file'
  | 'expect'
  | 'actual'
  | 'const'
  | 'suspend'
  ;
identifier
(used by packageHeader, importHeader)
  : simpleIdentifier ('.' simpleIdentifier)*
  ;

Lexical grammar

General

ShebangLine
(used by shebangLine)
 : '#!' ~[\r\n]*
 ;
DelimitedComment
(used by DelimitedComment, Hidden)
 : ('/*' (DelimitedComment | .)*? '*/')
 ;
LineComment
(used by Hidden)
 : ('//' ~[\r\n]*)
 ;
WS
(used by Hidden)
 : [\u0020\u0009\u000C]
 ;
helper
Hidden
(used by EXCL_WS, AT_POST_WS, AT_PRE_WS, AT_BOTH_WS, QUEST_WS, NOT_IS, NOT_IN)
 : DelimitedComment
 | LineComment
 | WS
 ;

Separators and operations

RESERVED
 : '...'
 ;
EXCL_WS
(used by excl)
 : '!' Hidden
 ;
DOUBLE_ARROW
 : '=>'
 ;
DOUBLE_SEMICOLON
 : ';;'
 ;
HASH
 : '#'
 ;
AT_POST_WS
(used by label)
 : '@' Hidden
 ;
AT_PRE_WS
(used by fileAnnotation, singleAnnotation, multiAnnotation, annotationUseSiteTarget)
 : Hidden '@'
 ;
AT_BOTH_WS
 : Hidden '@' Hidden
 ;
QUEST_WS
(used by quest)
 : '?' Hidden
 ;
SINGLE_QUOTE
 : '\''
 ;

Keywords

RETURN_AT
(used by jumpExpression)
 : 'return@' Identifier
 ;
CONTINUE_AT
(used by jumpExpression)
 : 'continue@' Identifier
 ;
BREAK_AT
(used by jumpExpression)
 : 'break@' Identifier
 ;
THIS_AT
(used by thisExpression)

```
  : 'this@' Identifier
  ;
SUPER_AT
(used by superExpression)
  : 'super@' Identifier
  ;
TYPEOF
  : 'typeof'
  ;
NOT_IS
(used by isOperator)
  : '!is' Hidden
  ;
NOT_IN
(used by inOperator)
  : '!in' Hidden
  ;
```

Literals

```
helper
DecDigit
(used by DecDigitOrSeparator, DecDigits, IntegerLiteral)
  : '0'..'9'
  ;
helper
DecDigitNoZero
(used by IntegerLiteral)
  : '1'..'9'
  ;
helper
DecDigitOrSeparator
(used by DecDigits, IntegerLiteral)
  : DecDigit
  | '_'
  ;
helper
DecDigits
(used by DoubleExponent, FloatLiteral, DoubleLiteral)
  : DecDigit DecDigitOrSeparator* DecDigit
  | DecDigit
  ;
helper
DoubleExponent
(used by DoubleLiteral)
  : [eE] [+-]? DecDigits
  ;
RealLiteral
(used by literalConstant)
  : FloatLiteral
  | DoubleLiteral
  ;
FloatLiteral
(used by RealLiteral)
  : DoubleLiteral [fF]
  | DecDigits [fF]
  ;
DoubleLiteral
(used by RealLiteral, FloatLiteral)
  : DecDigits? '.' DecDigits DoubleExponent?
  | DecDigits DoubleExponent
  ;
IntegerLiteral
(used by literalConstant, UnsignedLiteral, LongLiteral)
  : DecDigitNoZero DecDigitOrSeparator* DecDigit
  | DecDigit
  ;
helper
HexDigit
(used by HexDigitOrSeparator, HexLiteral, UniCharacterLiteral)
  : [0-9a-fA-F]
  ;
helper
HexDigitOrSeparator
(used by HexLiteral)
  : HexDigit
  | '_'
  ;
HexLiteral
```

(used by [literalConstant](#), [UnsignedLiteral](#), [LongLiteral](#))
  : '0' [xX] [HexDigit](#) [HexDigitOrSeparator](#)* [HexDigit](#)
  | '0' [xX] [HexDigit](#)
  ;
helper
BinDigit
(used by [BinDigitOrSeparator](#), [BinLiteral](#))
  : [01]
  ;
helper
BinDigitOrSeparator
(used by [BinLiteral](#))
  : [BinDigit](#)
  | '_'
  ;
BinLiteral
(used by [literalConstant](#), [UnsignedLiteral](#), [LongLiteral](#))
  : '0' [bB] [BinDigit](#) [BinDigitOrSeparator](#)* [BinDigit](#)
  | '0' [bB] [BinDigit](#)
  ;
UnsignedLiteral
(used by [literalConstant](#))
  : ([IntegerLiteral](#) | [HexLiteral](#) | [BinLiteral](#)) [uU] 'L'?
  ;
LongLiteral
(used by [literalConstant](#))
  : ([IntegerLiteral](#) | [HexLiteral](#) | [BinLiteral](#)) 'L'
  ;
BooleanLiteral
(used by [literalConstant](#))
  : 'true'
  | 'false'
  ;
CharacterLiteral
(used by [literalConstant](#))
  : '\'' ([EscapeSeq](#) | ~[\n\r'\\]) '\''
  ;

Identifiers

helper
UnicodeDigit
(used by [Identifier](#))
  : [UNICODE_CLASS_ND](#)
  ;
Identifier
(used by [simpleIdentifier](#), [RETURN_AT](#), [CONTINUE_AT](#), [BREAK_AT](#), [THIS_AT](#), [SUPER_AT](#), [IdentifierOrSoftKey](#))
  : ([Letter](#) | '_') ([Letter](#) | '_' | [UnicodeDigit](#))*
  | '`' ~([\r\n] | '`')+ '`'
  ;

Depending on the target and publicity of the declaration, the set of allowed symbols in identifiers is different. This rule contains the union of allowed symbols from all targets. Thus, the code for any target can be parsed using the grammar.

The allowed symbols in identifiers corresponding to the target and publicity of the declaration are given below.

Kotlin/JVM (any declaration publicity)
~ ([\r\n] | '`' | '.' | ';' | ':' | '\' | '/' | '[' | ']' | '<' | '>')

Kotlin/Android (any declaration publicity)

The allowed symbols are different from allowed symbols for Kotlin/JVM and correspond to the [Dalvik Executable format](#).

Kotlin/JS (private declarations)
~ ([\r\n] | '`')

Kotlin/JS (public declarations)

The allowed symbols for public declarations correspond to the [ECMA specification (section 7.6)](#) except that ECMA reserved words is allowed.

Kotlin/Native (any declaration publicity)
~ ([\r\n] | '`')

IdentifierOrSoftKey
(used by FieldIdentifier)
  : Identifier
  | 'abstract'
  | 'annotation'
  | 'by'
  | 'catch'
  | 'companion'
  | 'constructor'
  | 'crossinline'
  | 'data'
  | 'dynamic'
  | 'enum'
  | 'external'
  | 'final'
  | 'finally'
  | 'import'
  | 'infix'
  | 'init'
  | 'inline'
  | 'inner'
  | 'internal'
  | 'lateinit'
  | 'noinline'
  | 'open'
  | 'operator'
  | 'out'
  | 'override'
  | 'private'
  | 'protected'
  | 'public'
  | 'reified'
  | 'sealed'
  | 'tailrec'
  | 'vararg'
  | 'where'
  | 'get'
  | 'set'
  | 'field'
  | 'property'
  | 'receiver'
  | 'param'
  | 'setparam'
  | 'delegate'
  | 'file'
  | 'expect'
  | 'actual'
  | 'const'
  | 'suspend'
  ;
FieldIdentifier
(used by LineStrRef, MultiLineStrRef)
  : '$' IdentifierOrSoftKey
  ;
helper
UniCharacterLiteral
(used by EscapeSeq, LineStrEscapedChar)
  : '\\' 'u' HexDigit HexDigit HexDigit HexDigit
  ;
helper
EscapedIdentifier
(used by EscapeSeq, LineStrEscapedChar)
  : '\\' ('t' | 'b' | 'r' | 'n' | '\'' | '"' | '\\' | '$')
  ;
helper
EscapeSeq
(used by CharacterLiteral)
  : UniCharacterLiteral
  | EscapedIdentifier
  ;

Characters

helper
Letter
(used by Identifier)

343

```
    : UNICODE_CLASS_LL
    | UNICODE_CLASS_LM
    | UNICODE_CLASS_LO
    | UNICODE_CLASS_LT
    | UNICODE_CLASS_LU
    | UNICODE_CLASS_NL
    ;
```

Strings

LineStrRef
(used by lineStringContent)
 : FieldIdentifier
 ;

See String templates

LineStrText
(used by lineStringContent)
 : ~('\\' | '"' | '$')+
 | '$'
 ;

LineStrEscapedChar
(used by lineStringContent)
 : EscapedIdentifier
 | UniCharacterLiteral
 ;

TRIPLE_QUOTE_CLOSE
(used by multiLineStringLiteral)
 : ('"'? '"""')
 ;

MultiLineStrRef
(used by multiLineStringContent)
 : FieldIdentifier
 ;

MultiLineStrText
(used by multiLineStringContent)
 : ~('"' | '$')+
 | '$'
 ;

ErrorCharacter
 : .
 ;

Java Interop

Calling Java code 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:

```kotlin
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.

For example:

```kotlin
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 will not be visible as a property in Kotlin, because Kotlin does not support set-only properties at this time.

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`, etc. 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 specially in Kotlin 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 we 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 one can not write them down explicitly in the language. When a platform value is assigned to a Kotlin variable, we can rely on type inference (the variable will have an inferred platform type then, as `item` has in the example above), or we can choose the type that we 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 we 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 we pass platform values to Kotlin functions expecting non-null values etc. 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 cannot 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 (in error messages, parameter info etc), so we have 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 which 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)

— 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 parameters

It is possible to annotate type arguments of generic types to provide nullability information for them as well. For example, consider these annotations on a Java declaration:

```
@NotNull
Set<@NotNull String> toSet(@NotNull Collection<@NotNull String> elements) { ... }
```

It leads to the following signature seen in Kotlin:

```
fun toSet(elements: (Mutable)Collection<String>) : (Mutable)Set<String> { ... }
```

Note the `@NotNull` annotations on `String` type arguments. Without them, we get platform types in the type arguments:

```
fun toSet(elements: (Mutable)Collection<String!>) : (Mutable)Set<String!> { ... }
```

Annotating type arguments works with Java 8 target or higher and requires the nullability annotations to support the `TYPE_USE` target (`org.jetbrains.annotations` supports this in version 15 and above).

> Note: due to the current technical limitations, the IDE does not correctly recognize these annotations on type arguments in compiled Java libraries that are used as dependencies.

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.

Since Kotlin 1.1.50, [custom nullability qualifiers (KEEP-79)](#) are also supported (see below).

Type qualifier nicknames (since 1.1.50)

If an annotation type is annotated with both [@TypeQualifierNickname](#) and JSR-305 `@Nonnull` (or its another nickname, such as `@CheckForNull`), then the annotation type is itself used for retrieving precise nullability and has the same meaning as that nullability annotation:

```
@TypeQualifierNickname
@Nonnull(when = When.ALWAYS)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNonnull {
}

@TypeQualifierNickname
@CheckForNull // a nickname to another type qualifier nickname
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNullable {
}

interface A {
    @MyNullable String foo(@MyNonnull String x);
    // in Kotlin (strict mode): `fun foo(x: String): String?`

    String bar(List<@MyNonnull String> x);
    // in Kotlin (strict mode): `fun bar(x: List<String>!): String!`
}
```

Type qualifier defaults (since 1.1.50)

@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; and

— `ElementType.TYPE_USE` (since 1.1.60) 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 {
}

@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?
}
```

> Note: 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
@NonNullApi // declaring all types in package 'test' as non-nullable by default
package test;
```

@UnderMigration annotation (since 1.1.60)

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;

— with `MigrationStatus.WARN`, the inappropriate usages are reported as compilation warnings instead of errors, but the types in the annotated declarations remain platform; and

— `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 `@NonNullApi` is annotated `@UnderMigration(status = MigrationStatus.WARN)`
@NonNullApi
public class Test {}
```

Note: 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}` (since 1.1.60) 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}` (since 1.1.60) 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.

For kotlin versions 1.1.50+/1.2, 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 specially. 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.lang.Throwable	kotlin.Throwable!

Java's boxed primitive types are mapped to nullable Kotlin types:

Java type	Kotlin type
`java.lang.Byte`	`kotlin.Byte?`
`java.lang.Short`	`kotlin.Short?`
`java.lang.Integer`	`kotlin.Int?`
`java.lang.Long`	`kotlin.Long?`
`java.lang.Character`	`kotlin.Char?`
`java.lang.Float`	`kotlin.Float?`
`java.lang.Double`	`kotlin.Double?`
`java.lang.Boolean`	`kotlin.Boolean?`

Note that a boxed primitive type used as a type parameter is mapped to a platform type: for example, `List<java.lang.Integer>` becomes a `List<Int!>` in Kotlin.

Collection types may be read-only or mutable in Kotlin, so Java's collections are mapped as follows (all Kotlin types in this table reside in the package `kotlin.collections`):

Java type	Kotlin read-only type	Kotlin mutable type	Loaded platform type
`Iterator<T>`	`Iterator<T>`	`MutableIterator<T>`	`(Mutable)Iterator<T>!`
`Iterable<T>`	`Iterable<T>`	`MutableIterable<T>`	`(Mutable)Iterable<T>!`
`Collection<T>`	`Collection<T>`	`MutableCollection<T>`	`(Mutable)Collection<T>!`
`Set<T>`	`Set<T>`	`MutableSet<T>`	`(Mutable)Set<T>!`
`List<T>`	`List<T>`	`MutableList<T>`	`(Mutable)List<T>!`
`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>!`

Note: 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 we perform some conversions:

- 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<*>!` , i.e. `List<out Any?>!` .

Like Java's, Kotlin's generics are not retained at runtime, i.e. objects do not carry information about actual type arguments passed to their constructors, i.e. `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 does not let us 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 JVM byte codes, 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 we 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)
```

It's currently not possible to pass null to a method that is declared as varargs.

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, which is supported since Kotlin 1.1. 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 { ... }
}
```

Do not 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. We cannot 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`.

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

Just like Java 8, Kotlin supports SAM conversions. This 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") })
```

Note that SAM conversions only work for interfaces, not for abstract classes, even if those also have just a single abstract method.

Also note that this feature works only for Java interop; since Kotlin has proper function types, automatic conversion of functions into implementations of Kotlin interfaces is unnecessary and therefore unsupported.

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:

```kotlin
external fun foo(x: Int): Double
```

The rest of the procedure works in exactly the same way as in Java.

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` gets compiled to the following Java declarations:

```java
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`.

```kotlin
// app.kt
package org.example

class Util

fun getTime() { /*...*/ }
```

```
// Java
new org.example.Util();
org.example.AppKt.getTime();
```

The name of the generated Java class can be changed using 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 which have 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 has the ability to 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 of the 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, and is not a delegated property.

```kotlin
class User(id: String) {
    @JvmField val ID = id
}
```

```java
// 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.

```kotlin
class Key(val value: Int) {
    companion object {
        @JvmField
        val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value }
    }
}
```

```java
// 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.

```kotlin
object Singleton {
    lateinit var provider: Provider
}
```

```java
// 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.

```kotlin
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.

> The `@JvmDefault` annotation is experimental in Kotlin 1.3. Its name and behavior may change, leading to future incompatibility.

Starting from JDK 1.8, interfaces in Java can contain [default methods](). You can declare a non-abstract member of a Kotlin interface as default for the Java classes implementing it. To make a member default, mark it with the [@JvmDefault]() annotation. Here is an example of a Kotlin interface with a default method:

```kotlin
interface Robot {
    @JvmDefault fun move() { println("~walking~") }
    fun speak(): Unit
}
```

The default implementation is available for Java classes implementing the interface.

```java
//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");
    }
}
```

```java
C3PO c3po = new C3PO();
c3po.move(); // default implementation from the Robot interface
c3po.speak();
```

Implementations of the interface can override default methods.

```java
//Java
public class BB8 implements Robot {
    //own implementation of the default method
    @Override
    public void move() {
        System.out.println("~rolling~");
    }

    @Override
    public void speak() {
        System.out.println("Beep-beep");
    }
}
```

For the `@JvmDefault` annotation to take effect, the interface must be compiled with an `-Xjvm-default` argument. Depending on the case of adding the annotation, specify one of the argument values:

— `-Xjvm-default=enabled` should be used if you add only new methods with the `@JvmDefault` annotation. This includes adding the entire interface for your API.

— `-Xjvm-default=compatibility` should be used if you are adding a `@JvmDefault` to the methods that were available in the API before. This mode helps avoid compatibility breaks: all the interface implementations written for the previous versions will be fully compatible with the new version. However, the compatibility mode may add some overhead to the resulting bytecode size and affect the performance.

For more details about compatibility issues, see the `@JvmDefault` reference page.

Note that if an interface with `@JvmDefault` methods is used as a delegate, the default method implementations are called even if the actual delegate type provides its own implementations.

```kotlin
interface Producer {
    @JvmDefault fun produce() {
        println("interface method")
    }
}

class ProducerImpl: Producer {
    override fun produce() {
        println("class method")
    }
}

class DelegatedProducer(val p: Producer): Producer by p {
}

fun main() {
    val prod = ProducerImpl()
    DelegatedProducer(prod).produce() // prints "interface method"
}
```

For more details about interface delegation in Kotlin, see Delegation.

Visibility

The Kotlin visibilities are mapped 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 byte code. The most prominent example happens due to *type erasure*:

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

These two functions can not be defined side-by-side, because their JVM signatures are the same: `filterValid(Ljava/util/List;)Ljava/util/List;`. If we really want them to have the same name in Kotlin, we can annotate one (or both) of them with [@JvmName](#) and specify a different name as an argument:

```
fun List<String>.filterValid(): List<String>

@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

From Kotlin they 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")
@set: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 no-argument constructor will be generated for it. This works even if the @JvmOverloads annotation is not specified.

Checked exceptions

As we mentioned above, Kotlin does not have checked exceptions. So, normally, the Java signatures of Kotlin functions do not declare exceptions thrown. Thus if we have a function in Kotlin like this:

```
// example.kt
package demo

fun writeToFile() {
    /*...*/
    throw IOException()
}
```

And we 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
  // ...
}
```

we 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. Let's say we 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) { ... }
```

367

The problem is that in Kotlin we can say `unboxBase(boxDerived("s"))`, 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 it work in Java we'd have to define `unboxBase` as follows:

```
Base unboxBase(Box<? extends Base> box) { ... }
```

Here we make use of 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 we generate `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, we don't generate wildcards, 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 we need wildcards where they are not generated by default, we can use the `@JvmWildcard` annotation:

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> = Box(value)
// is translated to
// Box<? extends Derived> boxDerived(Derived value) { ... }
```

On the other hand, if we don't need wildcards where they are generated, we can 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:

```kotlin
fun emptyList(): List<Nothing> = listOf()
// is translated to
// List emptyList() { ... }
```

JavaScript

Setting 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 builds a JavaScript bundle from a Kotlin project using [Webpack](#).

To create a Kotlin/JS project in IntelliJ IDEA, go to **File | New | Project** and select **Gradle | Kotlin/JS for browser** or **Kotlin/JS for Node.js**. Be sure to clear the **Java** checkbox.

Alternatively, you can apply the `org.jetbrains.kotlin.js` plugin to a Gradle project manually in the `build.gradle` file. If you use the Gradle Kotlin DSL, you can apply the plugin with `kotlin("js")`.

```
plugins {
    id 'org.jetbrains.kotlin.js' version '1.3.61'
}
```

```
plugins {
    kotlin("js") version "1.3.61"
}
```

The Kotlin/JS 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](#) for browser projects.

Choosing execution environment

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 `target` section with `browser {}` or `nodejs {}` inside.

```
kotlin {
    target {
        browser {
        }
    }
}
```

The Kotlin/JS plugin automatically configures its tasks for working with the selected environment. This includes downloading and installing dependencies required for running and testing the application, and therefore lets developers build, run, and test simple projects without additional configuration.

Managing dependencies

Like any other Gradle projects, Kotlin/JS projects support traditional Gradle dependency declarations in the `dependencies` section of the build script.

```
dependencies {
    implementation 'org.example.myproject:1.1.0'
}
```

```
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.

```
kotlin {
    sourceSets {
        main {
            dependencies {
                implementation 'org.example.myproject:1.1.0'
            }
        }
    }
}
```

```
kotlin {
  sourceSets["main"].dependencies {
    implementation("org.example.myproject", "1.1.0")
  }
}
```

Kotlin standard libraries

The dependency on the Kotlin/JS [standard library](#) is mandatory for all Kotlin/JS projects. If your project contains tests written in Kotlin, you should also add the dependency on the [kotlin.test](#) library.

```
dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-stdlib-js'
    testImplementation 'org.jetbrains.kotlin:kotlin-test-js'
}
```

```
dependencies {
    implementation(kotlin("stdlib-js"))
    testImplementation(kotlin("test-js"))
}
```

NPM dependencies

In the JavaScript world, the common way to manage dependencies is [NPM](#). It offers the biggest public [repository](#)of JavaScript modules and a tool for downloading them. The Kotlin/JS plugin lets you declare NPM dependencies in the Gradle build script among other dependencies and does everything else automatically. Particularly, it installs the [Yarn](#) package manager and uses it to download the dependencies from the NPM repository to the `node_modules` directory of your project - the common location for NPM dependencies of a JavaScript project.

To declare an NPM dependency, use the `npm()` function inside the `dependencies` section of a source set.

```
kotlin {
    sourceSets {
        main {
            dependencies {
                implementation 'org.example.myproject:1.1.0'
            }
        }
    }
}
```

```
kotlin {
  sourceSets["main"].dependencies {
    implementation(npm("react", "16.8.3"))
  }
}
```

Once an NPM dependency is installed, you can use its API in your code as described in [Calling JS from Kotlin](#).

Configuring run task

The Kotlin/JS plugin provides a run task that lets you run projects without additional configuration. For running Kotlin/JS projects, it uses Webpack DevServer. If you want to customize the DevServer configuration, for example, change its port, use the Webpack configuration file.

To run the project, execute the standard lifecycle `run` task:

```
./gradlew run
```

To see the source file changes in browser without restarting the DevServer, use the Gradle continuous build:

```
./gradlew run --continuous
```

or

```
./gradlew run -t
```

Configuring test task

The Kotin/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 NodeJS 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

By default, the plugin uses Headless Chrome for running browser tests. You can also run them in other browsers by adding the corresponding entries inside the `useKarma` section of the build script:

```
kotlin {
    target {
        browser {
            testTask {
                useKarma {
                    useIe()
                    useSafari()
                    useFirefox()
                    useChrome()
                    useChromeCanary()
                    useChromeHeadless()
                    usePhantomJS()
                    useOpera()
                }
            }
        }
    }
}
```

If you want to skip tests, add the line `enabled = false` to the `testTask`.

```kotlin
kotlin {
    target {
        browser {
            testTask {
                enabled = false
            }
        }
    }
}
```

To run tests, execute the standard lifecycle `check` task:

```
./gradlew check
```

Configuring Webpack bundling

For building executable JavaScript artifacts, the Kotlin/JS plugins contains the `webpackTask`.

To build a project artifact using Webpack, execute the `build` Gradle task:

```
./gradlew build
```

Dynamic Type

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 this type can be assigned to any variable or passed anywhere as a parameter;

— any value can be assigned to a variable of type `dynamic` or passed to a function that takes `dynamic` as a parameter;

— `null`-checks are disabled for such 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 we can chain such calls freely:

```
dyn.foo().bar.baz()
```

When we 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](#).

Calling JavaScript from Kotlin

Kotlin was designed for easy interoperation with 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 in compile-time. You can freely talk to JavaScript from Kotlin via dynamic types, but if you want the full power of Kotlin type system, you can create Kotlin headers for JavaScript libraries.

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")
}
```

The parameter of `js` 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"
```

external modifier

To tell Kotlin that a certain declaration is written in pure JavaScript, you should mark it with `external` modifier. When the compiler sees such a declaration, it assumes that the implementation for the corresponding class, function or property is provided by the developer, and therefore does not try to generate any JavaScript code from the declaration. This means that you should omit bodies of `external` declarations. 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 `external` modifier is inherited by nested declarations, i.e. in `Node` class we do not put `external` 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.

Declaring (static) members of a class

In JavaScript you can define members either on a prototype or a class itself. I.e.:

```
function MyClass() { ... }
MyClass.sharedMember = function() { /* implementation */ };
MyClass.prototype.ownMember = function() { /* implementation */ };
```

There's no such syntax in Kotlin. However, in Kotlin we have `companion` objects. Kotlin treats companion objects of `external` class in a special way: instead of expecting an object, it assumes members of companion objects to be members of the class itself. To describe `MyClass` from the example above, you can write:

```
external class MyClass {
    companion object {
        fun sharedMember()
    }

    fun ownMember()
}
```

Declaring optional parameters

An external function can have optional parameters. How the JavaScript implementation actually computes default values for these parameters, is unknown to Kotlin, thus it's impossible to use the usual syntax to declare such parameters in Kotlin. You should use the following syntax:

```
external fun myFunWithOptionalArgs(x: Int,
    y: String = definedExternally,
    z: Long = definedExternally)
```

This means you can call `myFunWithOptionalArgs` with one required argument and two optional arguments (their default values are calculated by some JavaScript code).

Extending JavaScript classes

You can easily extend JavaScript classes as they were Kotlin classes. Just define an `external` class and extend it by non-`external` class. For example:

```
external open class HTMLElement : Element() {
    /* members */
}

class CustomElement : HTMLElement() {
    fun foo() {
        alert("bar")
    }
}
```

There are some limitations:

1. When a function of external base class is overloaded by signature, you can't override it in a derived class.

2. You can't override a function with default arguments.

Note that you can't extend a non-external class by external classes.

external interfaces

JavaScript does not have the concept of interfaces. When a function expects its parameter to support `foo` and `bar` methods, you just pass objects that actually have these methods. You can use interfaces to express this for statically-typed Kotlin, for example:

```kotlin
external interface HasFooAndBar {
    fun foo()

    fun bar()
}

external fun myFunction(p: HasFooAndBar)
```

Another use case for external interfaces is to describe settings objects. For example:

```kotlin
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:

1. They can't be used on the right hand side of `is` checks.
2. `as` cast to external interface always succeeds (and produces a warning in compile-time).
3. They can't be passed as reified type arguments.
4. They can't be used in class literal expressions (i.e. `I::class`).

Calling Kotlin from JavaScript

Kotlin compiler generates normal JavaScript classes, functions and properties you can freely use from JavaScript code. Nevertheless, there are some subtle things you should remember.

Isolating declarations in a separate JavaScript object

To prevent spoiling the global object, Kotlin creates an object that contains all Kotlin declarations from the current module. So if you name your module as `myModule`, all declarations are available to JavaScript via `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 module (see JavaScript Modules for more information on this). In this case there won't be a wrapper object, instead, declarations will be exposed as a JavaScript module of a corresponding kind. For example, in case of CommonJS you should write:

```
alert(require('myModule').foo());
```

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"
```

Can be called from JavaScript like this:

```
alert(myModule.my.qualified.packagename.foo());
```

@JsName annotation

In some cases (for example, to support overloads), Kotlin compiler mangles 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:

```
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 Kotlin compiler does not apply such mangling to `external` declarations, so you don't have to use `@JsName` on them. Another case worth noticing is inheriting non-external classes from external classes. In this case any overridden functions won't be mangled as well.

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()
```

Representing 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`), i.e. 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`.

— There's no 64 bit integer number in JavaScript, so `kotlin.Long` is not mapped to any JavaScript object, it's emulated by a Kotlin class.

— `kotlin.String` is mapped to JavaScript String.

381

— `kotlin.Any` is mapped to JavaScript Object (i.e. `new Object()`, `{}`, etc).

— `kotlin.Array` is mapped to JavaScript Array.

— Kotlin collections (i.e. `List`, `Set`, `Map`, etc) 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.

Starting with version 1.1.50 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

Kotlin allows you to compile your Kotlin projects to JavaScript modules for popular module systems. Here is the list of available options:

1. Plain. Don't compile for any module system. As usual, you can access a module by its name in the global scope. This option is used by default.
2. Asynchronous Module Definition (AMD), which is in particular used by require.js library.
3. CommonJS convention, widely used by node.js/npm (`require` function and `module.exports` object)
4. Unified Module Definitions (UMD), which is compatible with both *AMD* and *CommonJS*, and works as "plain" when neither *AMD* nor *CommonJS* is available at runtime.

Choosing the Target Module System

Choosing the target module system depends on your build environment:

From IntelliJ IDEA

Setup per module: Open File -> Project Structure..., find your module in Modules and select "Kotlin" facet under it. Choose appropriate module system in "Module kind" field.

Setup for the whole project: Open File -> Settings, select "Build, Execution, Deployment" -> "Compiler" -> "Kotlin compiler". Choose appropriate module system in "Module kind" field.

From Maven

To select module system when compiling via Maven, you should set `moduleKind` configuration property, i.e. your `pom.xml` should look like this:

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <executions>
        <execution>
            <id>compile</id>
            <goals>
                <goal>js</goal>
            </goals>
        </execution>
    </executions>
    <!-- Insert these lines -->
    <configuration>
        <moduleKind>commonjs</moduleKind>
    </configuration>
    <!-- end of inserted text -->
</plugin>
```

Available values are: `plain`, `amd`, `commonjs`, `umd`.

From Gradle

To select module system when compiling via Gradle, you should set `moduleKind` property, i.e.

```
compileKotlin2Js.kotlinOptions.moduleKind = "commonjs"
```

Available values are similar to Maven.

@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)
```

Applying @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 allows to 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 */ }
}
```

Important: files marked with `@file:JsModule` annotation can't declare non-external members. The example below produces compile-time error:

```
@file:JsModule("extModule")
package ext.jspackage.name

external fun foo()

fun bar() = "!" + foo() + "!" // error here
```

Importing 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 our 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 has `@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 project's static resources and include via `<script>` element. To tell Kotlin that it's ok to use a `@JsModule` declaration from non-module environment, you should put `@JsNonModule` declaration. For example, given JavaScript code:

```
function topLevelSayHello(name) { alert("Hello, " + name); }
if (module && module.exports) {
    module.exports = topLevelSayHello;
}
```

can be described like this:

```kotlin
@JsModule("hello")
@JsNonModule
@JsName("topLevelSayHello")
external fun sayHello(name: String)
```

Notes

Kotlin is distributed with `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. Also it is available on NPM as [kotlin package](#)

JavaScript Reflection

At this time, JavaScript does not support the full Kotlin reflection API. The only supported part of the API is the `::class` syntax which allows you to refer to the class of an instance, or the class corresponding to the given type. The value of a `::class` expression is a stripped-down KClass implementation that only supports the simpleName and isInstance members.

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.

Examples:

```kotlin
class A
class B
class C

inline fun <reified T> foo() {
    println(T::class.simpleName)
}

val a = A()
println(a::class.simpleName)  // Obtains class for an instance; prints "A"
println(B::class.simpleName)  // Obtains class for a type; prints "B"
println(B::class.js.name)     // prints "B"
foo<C>()                      // prints "C"
```

JavaScript DCE

Since version 1.1.4, Kotlin/JS includes a dead code elimination (DCE) tool. This tool allows to strip out unused properties, functions and classes from the generated JS. There are several ways you get unused declarations:

— Functions can be inlined and never get called directly (which happens always except for few situations).

— You are using a shared library which provides much more functions than you actually need. For example, standard library (`kotlin.js`) contains functions for manipulating lists, arrays, char sequences, adapters for DOM, etc, which together gives about 1.3 mb file. A simple "Hello, world" application only requires console routines, which is only few kilobytes for the entire file.

Dead code elimination is often also called 'tree shaking'.

How to use

DCE tool is currently available from Gradle.

To activate DCE tool, add the following line to `build.gradle`:

```
apply plugin: 'kotlin-dce-js'
```

Note that if you are using multi-project build, you should apply plugin to the main project that is an entry point to your application.

By default, the resulting set of JavaScript files (your application together with all dependencies) can be found at path `$BUILD_DIR/min/`, where `$BUILD_DIR` is the path to generated JavaScript (usually, `build/classes/main`).

Configuring

To configure DCE on the main source set, you can use the `runDceKotlinJs` task (and corresponding `runDce<sourceSetName>KotlinJs` for other source sets).

Sometimes you are going to use a Kotlin declaration directly from JavaScript, and it's being stripped out by DCE. You may want to keep this declaration. To do so, you can use the following syntax in `build.gradle`:

```
runDceKotlinJs.keep "declarationToKeep"[, "declarationToKeep", ...]
```

Where `declarationToKeep` has the following syntax:

```
moduleName.dot.separated.package.name.declarationName
```

For example, consider a module is named `kotlin-js-example` and it contains a function named `toKeep` in package `org.jetbrains.kotlin.examples`. Use the following line:

```
runDceKotlinJs.keep "kotlin-js-example_main.org.jetbrains.kotlin.examples.toKeep"
```

Note that if your function has parameters, its name will be mangled, so the mangled name should be used in the keep directive.

Development mode

Running DCE takes a bit of extra time each build, and the output size does not matter during development. You can improve development builds time by making the DCE tool skip actual dead code elimination with the `dceOptions.devMode` flag of the DCE tasks.

For example, to disable DCE based on a custom condition for the `main` source set and always for the `test` code, add the following lines to the build script:

```
runDceKotlinJs.dceOptions.devMode = isDevMode
runDceTestKotlinJs.dceOptions.devMode = true
```

Example

A full example that shows how to integrate Kotlin with DCE and webpack to get a small bundle, can be found [here](#).

Notes

— As for 1.3.x versions, DCE tool is an experimental feature. This does not mean we are going to remove it, or that it's unusable for production. This means that we can change names of configuration parameters, default settings, etc.

— Currently you should not use DCE tool if your project is a shared library. It's only applicable when you are developing an application (which may use shared libraries). The reason is: DCE does not know which parts of the library are going to be used by the user's application.

— DCE does not perform minification (uglification) of your code by removing unnecessary whitespaces and shortening identifiers. You should use existing tools, like [UglifyJS](#) or [Google Closure Compiler](#) for this purpose.

Native

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

— 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.

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-freeze programming with code reuse across platforms in MPP projects.

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 foo.kt -p library -o bar
```

the above 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 qux.kt -l bar
```

the above 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 INTEROP.md

klib utility

The **klib** library management utility allows you to inspect and install the libraries.

The following commands are available.

To list library contents:

```
$ klib contents <name>
```

To inspect the bookkeeping details of the library

```
$ klib info <name>
```

To install the library to the default location use

```
$ klib install <name>
```

To 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 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
```

We 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 we have just created:

```
$ kotlinc 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.
```

The 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/
    - 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 *java property* format describing the library.
```

An example layout can be found in `klib/stdlib` directory of your installation.

Platform libraries

Overview

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 `iPhone`) we provide the `posix` platform lib. It contains bindings to platform's implementation of `POSIX` standard.

To use the library just

```
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

`Kotlin/Native` installation provides a wide spectrum of examples demonstrating the use of platform libraries. See [samples](#) for details.

Kotlin/Native interoperability

Introduction

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.

— create a `.def` file describing what to include into bindings

— use the `cinterop` tool to produce Kotlin bindings

— run *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 a separate document OBJC_INTEROP.md.

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/main/c_interop/libgit2.def \
 -compiler-option -I/usr/local/include -o libgit2
```

Compile the client:

```
../../dist/bin/kotlinc src/main/kotlin \
 -library libgit2 -o GitChurn
```

Run the client:

```
./GitChurn.kexe ../..
```

Creating bindings for a new library

To create bindings for a new library, start by 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.

Selecting 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.

Filtering 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 <SomeLbrary/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.

Filtering by module maps

Some libraries have proper `module.modulemap` or `module.map` files in its 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, such as

```
 compilerOpts = -DBAR=bar
 compilerOpts.linux_x64 = -DFOO=foo1
 compilerOpts.mac_x64 = -DFOO=foo2
```

and so, C headers on Linux will be analyzed with `-DBAR=bar -DFOO=foo1` and on macOS with `-DBAR=bar -DFOO=foo2`. Note that any definition file option can have both common and the platform-specific part.

Adding 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.

Including 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 `staticLibrary` 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.

Using 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 / 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
}
```

Passing 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:

```c
void foo(int* elements, int count);
...
int elements[] = {1, 2, 3};
foo(elements, 3);
```

Kotlin:

```kotlin
foo(cValuesOf(1, 2, 3), 3)
```

Working with the 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.

   ```kotlin
   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:

```kotlin
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.

Passing and receiving 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()`.

Passing 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: COpaquePointer, size: Int) {
    memset(buffer, 0, size.convert<size_t>())
}
```

Also, the type parameter can be inferred automatically and so may be omitted in some cases.

Object pinning

Kotlin objects could be pinned, i.e. their position in memory is guaranteed to be stable until unpinned, and pointers to such objects inner data could be passed to the C functions. For example

```kotlin
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.

Kotlin/Native 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 e.g. "Using cinterop" in [Gradle plugin documentation](). 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 "Targets and output kinds" section in [Gradle plugin documentation](). See [calculator sample]() for an example.

Mappings

The table below shows how Kotlin concepts are mapped to Swift/Objective-C and vice versa.

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]()
`@Throws`	`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`	
`MutableSet`	`NSMutableSet`	`NSMutableSet`	[note]()
`Map`	`Dictionary`	`NSDictionary`	
`MutableMap`	`NSMutableDictionary`	`NSMutableDictionary`	[note]()
Function type	Function type	Block pointer type	[note]()
Suspend functions	Unsupported	Unsupported	[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. In this case all Kotlin exceptions (except for instances of `Error`, `RuntimeException` and subclasses) are translated into a Swift error/ `NSError`.

Note that the opposite reversed translation is not implemented yet: Swift/Objective-C error-throwing methods aren't imported to Kotlin as exception-throwing.

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 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 Objc and Swift differ from Kotlin, so the translation will inevitably lose some information, but the features supported retain meaningful information.

To Use

Generics are currently not enabled by default. To have the framework header written with generics, add an experimental flag to the compiler config:

```
binaries.framework {
    freeCompilerArgs += "-Xobjc-generics"
}
```

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 Objc and Swift) or functions.

Nullability

Kotlin and Swift both define nullability as part of the type specification, while Objc 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 Objc 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 Objc 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 variOut = GenVarOut<SomeData>(arg: sd)
let variOutAny : GenVarOut<BaseData> = variOut as! GenVarOut<BaseData>
```

Constraints

In Kotlin you can provide upper bounds for a generic type. Objective-C also supports this, but that support is unavailable in more complex cases, and is currently not supported in the Kotlin - Objective-C interop. The exception here being a non-null upper bound will make Objective-C methods/properties non-null.

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 INTEROP.md for an example case where the library uses some plain C features (e.g. unsafe pointers, structs etc.).

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:

— suspend functions

— 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

Symbolicating iOS crash reports

Debugging an iOS application crash sometimes involves analyzing crash reports. More info about crash reports can be found [in the official 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.

Enable .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 doesn't produce `.dSYM` for release (i.e. optimized) binaries. This can be changed with `-Xg0` experimental compiler flag: it enables debug info and `.dSYM` bundle generation for produced release binaries. To enable it in Gradle, use

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.all {
            freeCompilerArgs += "-Xg0"
        }
    }
}
```

(in Kotlin DSL).

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. So in this case it may be required to make the framework static, e.g. with

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.withType<org.jetbrains.kotlin.gradle.plugin.mpp.Framework> {
            isStatic = true
        }
    }
}
```

(in Kotlin DSL).

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.

CocoaPods integration

Starting with 1.3.30, an experimental integration with [CocoaPods](#) is added to Kotlin/Native. This feature allows you to represent a Kotlin/Native Gradle-project as a CocoaPods dependency. Such a representation provides the following advantages:

— Such a dependency can be included in a Podfile of an Xcode project and automatically built (and rebuilt) along with this project. As a result, importing to Xcode is simplified since there is no need to write corresponding Gradle tasks and Xcode build steps manually.

— When building from Xcode, you can use CocoaPods libraries without writing .def files manually and setting cinterop tool parameters. In this case, all required parameters can be obtained from the Xcode project configured by CocoaPods.

For an example of CocoaPods integration, refer to the [cocoapods](#) sample.

CocoaPods Gradle plugin

The CocoaPods support is implemented in a separate Gradle plugin: `org.jetbrains.kotlin.native.cocoapods`.

> **Note:** The plugin is based on the multiplatform project model and requires applying the `org.jetbrains.kotlin.multiplatform` plugin. See details about the multiplatform plugin at the [corresponding page](#).

When applied, the CocoaPods plugin does the following:

1. Adds both `debug` and `release` frameworks as output binaries for all iOS and macOS targets.
2. Creates a `podspec` task which generates a [podspec](#) file for the given project.

The podspec generated includes a path to an output framework and script phases which automate building this framework during a build process of an Xcode project. Some fields of the podspec file can be configured using the `kotlin.cocoapods { ... }` code block.

```
// Apply plugins.
plugins {
    id("org.jetbrains.kotlin.multiplatform") version "1.3.30"
    id("org.jetbrains.kotlin.native.cocoapods") version "1.3.30"
}

// 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"
    }
}
```

The following podspec fields are required by CocoaPods:

— `version`

— `summary`

— `homepage`

A version of the Gradle project is used as a value for the `version` field. Fields `summary` and `homepage` can be configured using the `cocoapods` code block.

This podspec file can be referenced from a [Podfile](#) of an Xcode project. After that the framework built from the Kotlin/Native module can be used from this Xcode project. If necessary, this framework is automatically rebuilt during Xcode build process.

Workflow

To import a Kotlin/Native module in an existing Xcode project:

1. Make sure that you have CocoaPods [installed](#). We recommend using CocoaPods 1.6.1 or later.

2. Configure a Gradle project: apply the `org.jetbrains.kotlin.native.cocoapods` plugin, add and configure the targets, and specify the required podspec fields.

3. Run the `podspec` task. The podspec file described above will be generated.

   In order to avoid compatibility issues during an Xcode build, the plugin requires using a [Gradle wrapper](#). To generate the wrapper automatically during execution of the `podspec` task, run it with the parameter -Pkotlin.native.cocoapods.generate.wrapper=true.

4. Add a reference to the generated podspec in a Podfile of the Xcode project.

   ```
   target 'my-ios-app' do
       pod 'my_kotlin_library', :path => 'path/to/my-kotlin-library'
   end
   ```

5. Run `pod install` for the Xcode project.

After completing these steps, you can open the generated workspace (see [CocoaPods documentation](#)) and run an Xcode build.

Interoperability

The CocoaPods plugin also allows using CocoaPods libraries without manual configuring cinterop parameters (see the [corresponding section](#) of the multiplatform plugin documentation). The `cocoapods { ... }` code block allows you to add dependencies on CocoaPods libraries.

```
kotlin {
    cocoapods {
        // Configure a dependency on AFNetworking.
        // The CocoaPods version notation is supported.
        // The dependency will be added to all macOS and iOS targets.
        pod("AFNetworking", "~> 3.2.0")
    }
}
```

To use these dependencies from a Kotlin code, import a package `cocoapods.<library-name>`. In the example above, it's `cocoapods.AFNetworking`.

The dependencies declared in this way are added in the podspec file and downloaded during the execution of `pod install`.

> **Important:** To correctly import the dependencies into the Kotlin/Native module, the Podfile must contain either `use_modular_headers!` or `use_frameworks!` directive.

Search paths for libraries added in the Kotlin/Native module in this way are obtained from properties of the Xcode projects configured by CocoaPods. Thus if the module uses CocoaPods libraries, it can be build **only from Xcode**.

Current Limitations

— If a Kotlin/Native module uses a CocoaPods library, you can built this module only from an Xcode project. Otherwise the CocoaPods library cannot be resolved by the Kotlin/Native infrastructure.

— Subspecs are not supported.

Kotlin/Native Gradle plugin

Since 1.3.40, a separate Gradle plugin for Kotlin/Native is deprecated in favor of the `kotlin-multiplatform` plugin. This plugin provides an IDE support along with support of the new multiplatform project model introduced in Kotlin 1.3.0. Below you can find a short list of differences between `kotlin-platform-native` and `kotlin-muliplatform` plugins. For more information see the `kotlin-muliplatform` [documentation page](#). For `kotlin-platform-native` reference see the [corresponding section](#).

Applying the multiplatform plugin

To apply the `kotlin-multiplatform` plugin, just add the following snippet into your build script:

```
plugins {
    id("org.jetbrains.kotlin.multiplatform") version '1.3.40'
}
```

Managing targets

With the `kotlin-platform-native` plugin a set of target platforms is specified as a list in properties of the main component:

```
components.main {
    targets = ['macos_x64', 'linux_x64', 'mingw_x64']
}
```

With the `kotlin-multiplatform` plugin target platforms can be added into a project using special methods available in the `kotlin` extension. Each method adds into a project one **target** which can be accessed using the `targets` property. Each target can be configured independently including output kinds, additional compiler options etc. See details about targets at the [corresponding page](#).

```
import org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget

kotlin {
    // These targets are declared without any target-specific settings.
    macosX64()
    linuxX64()

    // You can specify a custom name used to access the target.
    mingwX64("windows")

    iosArm64 {
        // Additional settings for ios_arm64.
    }

    // You can access declared targets using the `targets` property.
    println(targets.macosX64)
    println(targets.windows)

    // You also can configure all native targets in a single block.
    targets.withType(KotlinNativeTarget) {
        // Native target configuration.
    }
}
```

Each target includes two **compilations**: `main` and `test` compiling product and test sources respectively. A compilation is an abstraction over a compiler invocation and described at the [corresponding page](#).

Managing sources

With the `kotlin-platform-native` plugin source sets are used to separate test and product sources. Also you can specify different sources for different platforms in the same source set:

```
sourceSets {
    // Adding target-independent sources.
    main.kotlin.srcDirs += 'src/main/mySources'

    // Adding Linux-specific code.
    main.target('linux_x64').srcDirs += 'src/main/linux'
}
```

With the `kotlin-multiplatform` plugin **source sets** are also used to group sources but source files for different platforms are located in different source sets. For each declared target two source sets are created: `<target-name>Main` and `<target-name>Test` containing product and test sources for this platform. Common for all platforms sources are located in `commonMain` and `commonTest` source sets created by default. More information about source sets can be found [here](#).

```
kotlin {
    sourceSets {
        // Adding target-independent sources.
        commonMain.kotlin.srcDirs += file("src/main/mySources")

        // Adding Linux-specific code.
        linuxX64Main.kotlin.srcDirs += file("src/main/linux")
    }
}
```

Managing dependencies

With the `kotlin-platform-native` plugin dependencies are configured in a traditional for Gradle way by grouping them into configurations using the project `dependencies` block:

```
dependencies {
    implementation 'org.sample.test:mylibrary:1.0'
    testImplementation 'org.sample.test:testlibrary:1.0'
}
```

The `kotlin-multiplatform` plugin also uses configurations under the hood but it also provides a `dependencies` block for each source set allowing configuring dependencies of this sources set:

```
kotlin.sourceSets {
    commonMain {
        dependencies {
            implementation("org.sample.test:mylibrary:1.0")
        }
    }

    commonTest {
        dependencies {
            implementation("org.sample.test:testlibrary:1.0")
        }
    }
}
```

Note that a module referenced by a dependency declared for `commonMain` or `commonTest` source set must be published using the `kotlin-multiplatform` plugin. If you want to use libraries published by the `kotlin-platform-native` plugin, you need to declare a separate source set for common native sources.

```
kotlin.sourceSets {
    // Create a common source set used by native targets only.
    nativeMain {
        dependsOn(commonMain)
        dependencies {
            // Depend on a library published by the kotlin-platform-naive plugin.
            implementation("org.sample.test:mylibrary:1.0")
        }
    }

    // Configure all native platform sources sets to use it as a common one.
    linuxX64Main.dependsOn(nativeMain)
    macosX64Main.dependsOn(nativeMain)
    //...
}
```

See more info about dependencies at the [corresponding page](#).

Output kinds

With the `kotlin-platform-native` plugin output kinds are specified as a list in properties of a component:

```
components.main {
    // Compile the component into an executable and a Kotlin/Native library.
    outputKinds = [EXECUTABLE, KLIBRARY]
}
```

With the `kotlin-multiplatform` plugin a compilation always produces a `*.klib` file. A separate `binaries` block is used to configure what final native binaries should be produced by each target. Each binary can be configured independently including linker options, executable entry point etc.

```
kotlin {
    macosX64 {
        binaries {
            executable {
                // Binary configuration: linker options, name, etc.
            }
            framework {
                // ...
            }

        }
    }
}
```

See more about native binaries declaration at the [corresponding page](#).

Publishing
```

Both `kotlin-platform-native` and `kotlin-multiplatform` plugins automatically set up artifact publication when the `maven-publish` plugin is applied. See details about publication at the [corresponding page](#). Note that currently only Kotlin/Native libraries ( `*.klib` ) can be published for native targets.

## Cinterop support

With the `kotlin-platform-native` plugin interop with a native library can be declared in component dependencies:

```
components.main {
    dependencies {
        cinterop('mystdio') {
            // Cinterop configuration.
        }
    }
}
```

With the `kotlin-multiplatform` plugin interops are configured as a part of a compilation (see details [here](#)). The rest of an interop configuration is the same as for the `kotlin-platform-native` plugin.

```
kotlin {
    macosX64 {
        compilations.main.cinterops {
            mystdio {
                // Cinterop configuration.
            }
        }
    }
}
```

`kotlin-platform-native` reference

Overview

You may use the Gradle plugin to build *Kotlin/Native* projects. Builds of the plugin are [available](#) at the Gradle plugin portal, so you can apply it using Gradle plugin DSL:

```
plugins {
    id "org.jetbrains.kotlin.platform.native" version "1.3.0-rc-146"
}
```

You also can get the plugin from a Bintray repository. In addition to releases, this repo contains old and development versions of the plugin which are not available at the plugin portal. To get the plugin from the Bintray repo, include the following snippet in your build script:

```
buildscript {
    repositories {
        mavenCentral()
        maven {
            url "https://dl.bintray.com/jetbrains/kotlin-native-dependencies"
        }
    }

    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-native-gradle-plugin:1.3.0-rc-146"
    }
}

apply plugin: 'org.jetbrains.kotlin.platform.native'
```

By default the plugin downloads the Kotlin/Native compiler during the first run. If you have already downloaded the compiler manually you can specify the path to its root directory using `org.jetbrains.kotlin.native.home` project property (e.g. in `gradle.properties`).

```
org.jetbrains.kotlin.native.home=/home/user/kotlin-native-0.8
```

In this case the compiler will not be downloaded by the plugin.

Source management

Source management in the `kotlin.platform.native` plugin is uniform with other Kotlin plugins and is based on source sets. A source set is a group of Kotlin/Native source which may contain both common and platform-specific code. The plugin provides a top-level script block `sourceSets` allowing you to configure source sets. Also it creates the default source sets `main` and `test` (for production and test code respectively).

By default the production sources are located in `src/main/kotlin` and the test sources - in `src/test/kotlin`.

```
sourceSets {
    // Adding target-independent sources.
    main.kotlin.srcDirs += 'src/main/mySources'

    // Adding Linux-specific code. It will be compiled in Linux binaries only.
    main.target('linux_x64').srcDirs += 'src/main/linux'
}
```

Targets and output kinds

By default the plugin creates software components for the main and test source sets. You can access them via the `components` container provided by Gradle or via the `component` property of a corresponding source set:

```
// Main component.
components.main
sourceSets.main.component

// Test component.
components.test
sourceSets.test.component
```

Components allow you to specify:

— Targets (e.g. Linux/x64 or iOS/arm64 etc)

— Output kinds (e.g. executable, library, framework etc)

— Dependencies (including interop ones)

Targets can be specified by setting a corresponding component property:

```
components.main {
    // Compile this component for 64-bit MacOS, Linux and Windows.
    targets = ['macos_x64', 'linux_x64', 'mingw_x64']
}
```

The plugin uses the same notation as the compiler. By default, test component uses the same targets as specified for the main one.

Output kinds can also be specified using a special property:

```
components.main {
    // Compile the component into an executable and a Kotlin/Native library.
    outputKinds = [EXECUTABLE, KLIBRARY]
}
```

All constants used here are available inside a component configuration script block. The plugin supports producing binaries of the following kinds:

— `EXECUTABLE` - an executable file;

— `KLIBRARY` - a Kotlin/Native library (*.klib);

— `FRAMEWORK` - an Objective-C framework;

— `DYNAMIC` - shared native library;

— `STATIC` - static native library.

Also each native binary is built in two variants (build types): `debug` (debuggable, not optimized) and `release` (not debuggable, optimized). Note that Kotlin/Native libraries have only `debug` variant because optimizations are preformed only during compilation of a final binary (executable, static lib etc) and affect all libraries used to build it.

Compile tasks

The plugin creates a compilation task for each combination of the target, output kind, and build type. The tasks have the following naming convention:

```
compile<ComponentName><BuildType><OutputKind><Target>KotlinNative
```

For example `compileDebugKlibraryMacos_x64KotlinNative`,
`compileTestDebugKotlinNative`.

The name contains the following parts (some of them may be empty):

— `<ComponentName>` - name of a component. Empty for the main component.

— `<BuildType>` - `Debug` or `Release`.

— `<OutputKind>` - output kind name, e.g. `Executabe` or `Dynamic`. Empty if the component
has only one output kind.

— `<Target>` - target the component is built for, e.g. `Macos_x64` or `Wasm32`. Empty if the
component is built only for one target.

Also the plugin creates a number of aggregate tasks allowing you to build all the binaries for a
build type (e.g. `assembleAllDebug`) or all the binaries for a particular target (e.g.
`assembleAllWasm32`).

Basic lifecycle tasks like `assemble`, `build`, and `clean` are also available.

Running tests

The plugin builds a test executable for all the targets specified for the `test` component. If the
current host platform is included in this list the test running tasks are also created. To run tests,
execute the standard lifecycle `check` task:

```
./gradlew check
```

Dependencies

The plugin allows you to declare dependencies on files and other projects using traditional
Gradle's mechanism of configurations. The plugin supports Kotlin multiplatform projects allowing
you to declare the `expectedBy` dependencies

```
dependencies {
    implementation files('path/to/file/dependencies')
    implementation project('library')
    testImplementation project('testLibrary')
    expectedBy project('common')
}
```

It's possible to depend on a Kotlin/Native library published earlier in a maven repo. The plugin
relies on Gradle's metadata support so the corresponding feature must be enabled. Add the
following line in your `settings.gradle`:

```
enableFeaturePreview('GRADLE_METADATA')
```

Now you can declare a dependency on a Kotlin/Native library in the traditional `group:artifact:version` notation:

```
dependencies {
    implementation 'org.sample.test:mylibrary:1.0'
    testImplementation 'org.sample.test:testlibrary:1.0'
}
```

Dependency declaration is also possible in the component block:

```
components.main {
    dependencies {
        implementation 'org.sample.test:mylibrary:1.0'
    }
}

components.test {
    dependencies {
        implementation 'org.sample.test:testlibrary:1.0'
    }
}
```

Using cinterop

It's possible to declare a cinterop dependency for a component:

```
components.main {
    dependencies {
        cinterop('mystdio') {
            // src/main/c_interop/mystdio.def is used as a def file.

            // Set up compiler options
            compilerOpts '-I/my/include/path'

            // It's possible to set up different options for different targets
            target('linux') {
                compilerOpts '-I/linux/include/path'
            }
        }
    }
}
```

Here an interop library will be built and added in the component dependencies.

Often it's necessary to specify target-specific linker options for a Kotlin/Native binary using an interop. It can be done using the `target` script block:

```
components.main {
    target('linux') {
        linkerOpts '-L/path/to/linux/libs'
    }
}
```

Also the `allTargets` block is available.

```
components.main {
    // Configure all targets.
    allTargets {
        linkerOpts '-L/path/to/libs'
    }
}
```

Publishing

In the presence of `maven-publish` plugin the publications for all the binaries built are created. The plugin uses Gradle metadata to publish the artifacts so this feature must be enabled (see the dependencies section).

Now you can publish the artifacts with the standard Gradle `publish` task:

```
./gradlew publish
```

Only `EXECUTABLE` and `KLIBRARY` binaries are published currently.

The plugin allows you to customize the pom generated for the publication with the `pom` code block available for every component:

```
components.main {
    pom {
        withXml {
            def root = asNode()
            root.appendNode('name', 'My library')
            root.appendNode('description', 'A Kotlin/Native library')
        }
    }
}
```

Serialization plugin

The plugin is shipped with a customized version of the `kotlinx.serialization` plugin. To use it you don't have to add new buildscript dependencies, just apply the plugins and add a dependency on the serialization library:

```
apply plugin: 'org.jetbrains.kotlin.platform.native'
apply plugin: 'kotlinx-serialization-native'

dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-serialization-runtime-native'
}
```

The the example project for details.

DSL example

In this section a commented DSL is shown. See also the example projects that use this plugin, e.g. Kotlinx.coroutines, MPP http client

```
plugins {
    id "org.jetbrains.kotlin.platform.native" version "1.3.0-rc-146"
}

sourceSets.main {
    // Plugin uses Gradle's source directory sets here,
    // so all the DSL methods available in SourceDirectorySet can be called here.
    // Platform independent sources.
    kotlin.srcDirs += 'src/main/customDir'

    // Linux-specific sources
    target('linux').srcDirs += 'src/main/linux'
}

components.main {

    // Set up targets
    targets = ['linux_x64', 'macos_x64', 'mingw_x64']

    // Set up output kinds
    outputKinds = [EXECUTABLE, KLIBRARY, FRAMEWORK, DYNAMIC, STATIC]

    // Specify custom entry point for executables
    entryPoint = "org.test.myMain"

    // Target-specific options
    target('linux_x64') {
        linkerOpts '-L/linux/lib/path'
    }

    // Targets independent options
    allTargets {
        linkerOpts '-L/common/lib/path'
    }

    dependencies {

        // Dependency on a published Kotlin/Native library.
        implementation 'org.test:mylib:1.0'

        // Dependency on a project
        implementation project('library')

        // Cinterop dependency
        cinterop('interop-name') {
            // Def-file describing the native API.
            // The default path is src/main/c_interop/<interop-name>.def
            defFile project.file("deffile.def")

            // Package to place the Kotlin API generated.
            packageName 'org.sample'

            // Options to be passed to compiler and linker by cinterop tool.
            compilerOpts 'Options for native stubs compilation'
            linkerOpts 'Options for native stubs'

            // Additional headers to parse.
            headers project.files('header1.h', 'header2.h')

            // Directories to look for headers.
            includeDirs {
```

```
                // All objects accepted by the Project.file method may be used with both
options.

                // Directories for header search (an analogue 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 analogue.
                headerFilterOnly 'path1', 'path2'
            }
            // A shortcut for includeDirs.allHeaders.
            includeDirs "include/directory" "another/directory"

            // Pass additional command line options to the cinterop tool.
            extraOpts '-verbose'

            // Additional configuration for Linux.
            target('linux') {
                compilerOpts 'Linux-specific options'
            }
        }
    }

    // Additional pom settings for publication.
    pom {
        withXml {
            def root = asNode()
            root.appendNode('name', 'My library')
            root.appendNode('description', 'A Kotlin/Native library')
        }
    }

    // Additional options passed to the compiler.
    extraOpts '--time'
}
```

Debugging

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

Producing binaries with debug info with Kotlin/Native compiler

To produce binaries with the Kotlin/Native compiler it's sufficient to use the `-g` option on the command line.

Example:

```
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 motocycle")
}
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 = Point(x=1, y=2)
(lldb) p p
(ObjHeader *) $2 = Point(x=1, y=2)
(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 -- Konan_DebugPrint(a_variable)
(a_variable) one is 1(KInt) $0 = 0
(lldb)
```

Known issues

— performance of Python bindings.

Note: 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.

Q: How do I run my program?

A: 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.

Q: What is Kotlin/Native memory management model?

A: 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.

Q: How do I create a shared library?

A: Use the `-produce dynamic` compiler switch, or `binaries.sharedLib()` in Gradle, i.e.

```kotlin
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 `samples/python_extension` for an example of using such a shared object to provide a bridge between Python and Kotlin/Native.

Q: How do I create a static library or an object file?

A: Use the `-produce static` compiler switch, or `binaries.staticLib()` in Gradle, i.e.

```kotlin
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.

Q: How do I run Kotlin/Native behind a corporate proxy?

A: 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.

Q: How do I specify a custom Objective-C prefix/name for my Kotlin framework?

A: Use the `-module-name` compiler option or matching Gradle DSL statement, i.e.

```
kotlin {
    iosArm64("myapp") {
        binaries.framework {
            freeCompilerArgs += listOf("-module-name", "TheName")
        }
    }
}
```

```
kotlin {
    iosArm64("myapp") {
        binaries.framework {
            freeCompilerArgs += ["-module-name", "TheName"]
        }
    }
}
```

Q: How do I rename the iOS framework? (default name is *\<project name\>*.framework)

A: Use the `baseName` option. This will also set the module name.

```
kotlin {
    iosArm64("myapp") {
        binaries {
            framework {
                baseName = "TheName"
            }
        }
    }
}
```

Q: How do I enable bitcode for my Kotlin framework?

A: 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`.

Q: Why do I see `InvalidMutabilityException?`

A: 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).

Q: How do I make a singleton object mutable?

A: 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.

Q: How can I compile my project against the Kotlin/Native master?

A: One of the following should be done:

▸ For the CLI, you can compile using gradle as stated in the README (and if you get errors, you can try to do a `./gradlew clean`):
▸ For Gradle, you can use [Gradle composite builds](#) like this:

Coroutines

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 `kotlinx-coroutines-core` module as explained [in the project README](#).

Table of contents

Additional references

Table of contents

Coroutine Basics

This section covers basic coroutine concepts.

Your first coroutine

Run the following code:

```kotlin
import kotlinx.coroutines.*

fun main() {
    GlobalScope.launch { // launch a new coroutine in background and continue
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello,") // main thread continues while coroutine is delayed
    Thread.sleep(2000L) // block main thread for 2 seconds to keep JVM alive
}
```

> You can get full code here.

You will see the following result:

```
Hello,
World!
```

Essentially, coroutines are light-weight threads. They are launched with launch *coroutine builder* in a context of some CoroutineScope. Here we are launching a new coroutine in the GlobalScope, meaning that the lifetime of the new coroutine is limited only by the lifetime of the whole application.

You can achieve the same result replacing `GlobalScope.launch { ... }` with `thread { ... }` and `delay(...)` with `Thread.sleep(...)`. Try it (don't forget to import `kotlin.concurrent.thread`).

If you start by replacing `GlobalScope.launch` by `thread`, the compiler produces the following error:

```
Error: Kotlin: Suspend functions are only allowed to be called from a coroutine or another
suspend function
```

That is because delay is a special *suspending function* that does not block a thread, but *suspends* coroutine and it can be only used from a coroutine.

Bridging blocking and non-blocking worlds

The first example mixes *non-blocking* `delay(...)` and *blocking* `Thread.sleep(...)` in the same code. It is easy to lose track of which one is blocking and which one is not. Let's be explicit about blocking using runBlocking coroutine builder:

```
import kotlinx.coroutines.*

fun main() {
    GlobalScope.launch { // launch a new coroutine in background and continue
        delay(1000L)
        println("World!")
    }
    println("Hello,") // main thread continues here immediately
    runBlocking {     // but this expression blocks the main thread
        delay(2000L)  // ... while we delay for 2 seconds to keep JVM alive
    }
}
```

> You can get full code here.

The result is the same, but this code uses only non-blocking delay. The main thread invoking `runBlocking` *blocks* until the coroutine inside `runBlocking` completes.

This example can be also rewritten in a more idiomatic way, using `runBlocking` to wrap the execution of the main function:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> { // start main coroutine
    GlobalScope.launch { // launch a new coroutine in background and continue
        delay(1000L)
        println("World!")
    }
    println("Hello,") // main coroutine continues here immediately
    delay(2000L)      // delaying for 2 seconds to keep JVM alive
}
```

> You can get full code here.

Here `runBlocking<Unit> { ... }` works as an adaptor that is used to start the top-level main coroutine. We explicitly specify its `Unit` return type, because a well-formed `main` function in Kotlin has to return `Unit`.

This is also a way to write unit tests for suspending functions:

```
class MyTest {
    @Test
    fun testMySuspendingFunction() = runBlocking<Unit> {
        // here we can use suspending functions using any assertion style that we like
    }
}
```

Waiting for a job

Delaying for a time while another coroutine is working is not a good approach. Let's explicitly wait (in a non-blocking way) until the background [Job](#) that we have launched is complete:

```
val job = GlobalScope.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
```

> You can get full code [here](#).

Now the result is still the same, but the code of the main coroutine is not tied to the duration of the background job in any way. Much better.

Structured concurrency

There is still something to be desired for practical usage of coroutines. When we use `GlobalScope.launch`, we create a top-level coroutine. Even though it is light-weight, it still consumes some memory resources while it runs. If we forget to keep a reference to the newly launched coroutine it still runs. What if the code in the coroutine hangs (for example, we erroneously delay for too long), what if we launched too many coroutines and ran out of memory? Having to manually keep references to all the launched coroutines and [join](#) them is error-prone.

There is a better solution. We can use structured concurrency in our code. Instead of launching coroutines in the [GlobalScope](#), just like we usually do with threads (threads are always global), we can launch coroutines in the specific scope of the operation we are performing.

In our example, we have `main` function that is turned into a coroutine using [runBlocking](#)
coroutine builder. Every coroutine builder, including `runBlocking`, adds an instance of
[CoroutineScope](#) to the scope of its code block. We can launch coroutines in this scope without
having to `join` them explicitly, because an outer coroutine (`runBlocking` in our example)
does not complete until all the coroutines launched in its scope complete. Thus, we can make our
example simpler:

```
import kotlinx.coroutines.*

fun main() = runBlocking { // this: CoroutineScope
    launch { // launch a new coroutine in the scope of runBlocking
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
```

You can get full code [here](#).

Scope builder

In addition to the coroutine scope provided by different builders, it is possible to declare your
own scope using [coroutineScope](#) builder. It creates a coroutine scope and does not complete
until all launched children complete.

[runBlocking](#) and [coroutineScope](#) may look similar because they both wait for its body and all its
children to complete. The main difference between these two 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.

It can be demonstrated by the following example:

```
import kotlinx.coroutines.*

fun main() = runBlocking { // this: CoroutineScope
    launch {
        delay(200L)
        println("Task from runBlocking")
    }

    coroutineScope { // Creates a coroutine scope
        launch {
            delay(500L)
            println("Task from nested launch")
        }

        delay(100L)
        println("Task from coroutine scope") // This line will be printed before the
nested launch
    }

    println("Coroutine scope is over") // This line is not printed until the nested
launch completes
}
```

You can get full code here.

Note that right after "Task from coroutine scope" message, while waiting for nested launch, "Task from runBlocking" is executed and printed, though coroutineScope is not completed yet.

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 `suspend` modifier. That 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.*

fun main() = runBlocking {
    launch { doWorld() }
    println("Hello,")
}

// this is your first suspending function
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

You can get full code here.

446

But what if the extracted function contains a coroutine builder which is invoked on the current scope? In this case `suspend` modifier on the extracted function is not enough. Making `doWorld` an extension method on `CoroutineScope` is one of the solutions, but it may not always be applicable as it does not make API clearer. The idiomatic solution is to have either an explicit `CoroutineScope` as a field in a class containing the target function or an implicit one when the outer class implements `CoroutineScope`. As a last resort, [CoroutineScope(coroutineContext)](#) can be used, but such approach is structurally unsafe because you no longer have control on the scope of execution of this method. Only private APIs can use this builder.

Coroutines ARE light-weight

Run the following code:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    repeat(100_000) { // launch a lot of coroutines
        launch {
            delay(1000L)
            print(".")
        }
    }
}
```

> You can get full code [here](#).

It launches 100K coroutines and, after a second, each coroutine prints a dot. Now, try that with threads. What would happen? (Most likely your code will produce some sort of out-of-memory error)

Global coroutines are like daemon threads

The following code launches a long-running coroutine in [GlobalScope](#) that prints "I'm sleeping" twice a second and then returns from the main function after some delay:

```
GlobalScope.launch {
    repeat(1000) { i ->
        println("I'm sleeping $i ...")
        delay(500L)
    }
}
delay(1300L) // just quit after delay
```

> You can get full code [here](#).

You can run and see that it prints three lines and terminates:

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
```

Active coroutines that were launched in GlobalScope do not keep the process alive. They are like daemon threads.

Table of contents

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:

```kotlin
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.")
```

You can get 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:

```kotlin
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.")
```

> You can get 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.

```
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.")
```

You can get 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:

```
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.")
```

You can get 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:

```kotlin
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.")
```

You can get 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:

```kotlin
withTimeout(1300L) {
    repeat(1000) { i ->
        println("I'm sleeping $i ...")
        delay(500L)
    }
}
```

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:

```kotlin
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")
```

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
```

Table of contents

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:

```
val time = measureTimeMillis {
    val one = doSomethingUsefulOne()
    val two = doSomethingUsefulTwo()
    println("The answer is ${one + two}")
}
println("Completed in $time ms")
```

> You can get 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.

```
val time = measureTimeMillis {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    println("The answer is ${one.await() + two.await()}")
}
println("Completed in $time ms")
```

> You can get 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:

```
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")
```

> You can get 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 with an explicit [GlobalScope](#) reference. 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.

```
// The result type of somethingUsefulOneAsync is Deferred<Int>
fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
}

// The result type of somethingUsefulTwoAsync is Deferred<Int>
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:

```
// 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")
}
```

You can get 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. But 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.

```kotlin
val time = measureTimeMillis {
    println("The answer is ${concurrentSum()}")
}
println("Completed in $time ms")
```

You can get 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:

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    try {
        failedConcurrentSum()
    } catch(e: ArithmeticException) {
        println("Computation failed with ArithmeticException")
    }
}

suspend fun failedConcurrentSum(): Int = coroutineScope {
    val one = async<Int> {
        try {
            delay(Long.MAX_VALUE) // Emulates very long computation
            42
        } finally {
            println("First child was cancelled")
        }
```

```
    }
    val two = async<Int> {
        println("Second child throws an exception")
        throw ArithmeticException()
    }
    one.await() + two.await()
}
```

You can get 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
```

Table of contents

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:

```
launch { // context of the parent, main runBlocking coroutine
    println("main runBlocking      : I'm working in thread
${Thread.currentThread().name}")
}
launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
    println("Unconfined            : I'm working in thread
${Thread.currentThread().name}")
}
launch(Dispatchers.Default) { // will get dispatched to DefaultDispatcher
    println("Default               : I'm working in thread
${Thread.currentThread().name}")
}
launch(newSingleThreadContext("MyOwnThread")) { // will get its own new thread
    println("newSingleThreadContext: I'm working in thread
${Thread.currentThread().name}")
}
```

You can get 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 coroutines are launched in GlobalScope is represented by Dispatchers.Default and uses a shared background pool of threads, so `launch(Dispatchers.Default) { ... }` uses the same dispatcher as `GlobalScope.launch { ... }`.

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.

```
launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
    println("Unconfined      : I'm working in thread ${Thread.currentThread().name}")
    delay(500)
    println("Unconfined      : After delay in thread ${Thread.currentThread().name}")
}
launch { // context of the parent, main runBlocking coroutine
    println("main runBlocking: I'm working in thread ${Thread.currentThread().name}")
    delay(1000)
    println("main runBlocking: After delay in thread ${Thread.currentThread().name}")
}
```

You can get 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. The common approach to debugging applications with threads 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:

```
val a = async {
    log("I'm computing a piece of the answer")
    6
}
val b = async {
    log("I'm computing another piece of the answer")
    7
}
log("The answer is ${a.await() * b.await()}")
```

You can get 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](#)):

```
newSingleThreadContext("Ctx1").use { ctx1 ->
    newSingleThreadContext("Ctx2").use { ctx2 ->
        runBlocking(ctx1) {
            log("Started in ctx1")
            withContext(ctx2) {
                log("Working in ctx2")
            }
            log("Back to ctx1")
        }
    }
}
```

You can get 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 the `coroutineContext[Job]` expression:

```
println("My job is ${coroutineContext[Job]}")
```

You can get 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, when GlobalScope is used to launch a coroutine, there is no parent for the job of the new coroutine. It is therefore not tied to the scope it was launched from and operates independently.

```
// launch a coroutine to process some kind of incoming request
val request = launch {
    // it spawns two other jobs, one with GlobalScope
    GlobalScope.launch {
        println("job1: I run in GlobalScope 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?")
```

You can get full code here.

The output of this code is:

```
job1: I run in GlobalScope 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:

```
// launch a coroutine to process some kind of incoming request
val request = launch {
    repeat(3) { i -> // launch a few children jobs
        launch  {
            delay((i + 1) * 200L) // variable delay 200ms, 400ms, 600ms
            println("Coroutine $i is done")
        }
    }
    println("request: I'm done and I don't explicitly join my children that are still active")
}
request.join() // wait for completion of the request, including all its children
println("Now processing of the request is complete")
```

You can get 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:

```
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()}")
```

> You can get 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:

```
launch(Dispatchers.Default + CoroutineName("test")) {
    println("I'm working in thread ${Thread.currentThread().name}")
}
```

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:

```kotlin
class Activity {
    private val mainScope = MainScope()

    fun destroy() {
        mainScope.cancel()
    }
    // to be continued ...
```

Alternatively, we can implement the [CoroutineScope](#) interface in this `Activity` class. The best way to do it is to use delegation with default factory functions. We also can combine the desired dispatcher (we used [Dispatchers.Default](#) in this example) with the scope:

```kotlin
    class Activity : CoroutineScope by CoroutineScope(Dispatchers.Default) {
    // to be continued ...
```

Now, we can launch coroutines in the scope of this `Activity` without having to explicitly specify their context. 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 ->
            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.

```
val activity = Activity()
activity.doSomething() // run test function
println("Launched coroutines")
delay(500L) // delay for half a second
println("Destroying activity!")
activity.destroy() // cancels all coroutines
delay(1000) // visually confirm that they don't work
```

You can get 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()`.

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:

```
threadLocal.set("main")
println("Pre-main, current thread: ${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
val job = launch(Dispatchers.Default + threadLocal.asContextElement(value = "launch")) {
    println("Launch start, current thread: ${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
    yield()
    println("After yield, current thread: ${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
}
job.join()
println("Post-main, current thread: ${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
```

You can get 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 on what thread the
coroutine is executed. 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 documentation of the
[ThreadContextElement](#) interface that should be implemented.

Table of contents

Asynchronous Flow

Suspending functions 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 function `foo()` that returns a List of three numbers and then print them all using forEach:

```
fun foo(): List<Int> = listOf(1, 2, 3)

fun main() {
    foo().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 foo(): 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() {
    foo().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 function `foo` with a `suspend` modifier, so that it can perform its work without blocking and return the result as a list:

```
suspend fun foo(): List<Int> {
    delay(1000) // pretend we are doing something asynchronous here
    return listOf(1, 2, 3)
}

fun main() = runBlocking<Unit> {
    foo().forEach { value -> println(value) }
}
```

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 the `Sequence<Int>` type for synchronously computed values:

```
foo(): Flow<Int> = flow { // flow builder
for (i in 1..3) {
    delay(100) // pretend we are doing something useful here
    emit(i) // emit next value
}


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
foo().collect { value -> println(value) }
```

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 function `foo()` 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:

```
foo(): Flow<Int> = flow {
println("Flow started")
for (i in 1..3) {
    delay(100)
    emit(i)
}
}


main() = runBlocking<Unit> {
println("Calling foo...")
val flow = foo()
println("Calling collect...")
flow.collect { value -> println(value) }
println("Calling collect again...")
flow.collect { value -> println(value) }
}
```

You can get the full code from here.

Which prints:

```
Calling foo...
Calling collect...
Flow started
1
2
3
Calling collect again...
Flow started
1
2
3
```

This is a key reason the `foo()` function (which returns a flow) is not marked with `suspend` modifier. By itself, `foo()` 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

Flow adheres to the general cooperative cancellation of coroutines. However, flow infrastructure does not introduce additional cancellation points. It is fully transparent for cancellation. As usual, flow collection can be cancelled when the flow is suspended in a cancellable suspending function (like delay), and cannot be cancelled otherwise.

The following example shows how the flow gets cancelled on a timeout when running in a withTimeoutOrNull block and stops executing its code:

```
foo(): Flow<Int> = flow {
for (i in 1..3) {
    delay(100)
    println("Emitting $i")
    emit(i)
}


main() = runBlocking<Unit> {
withTimeoutOrNull(250) { // Timeout after 250ms
    foo().collect { value -> println(value) }
}
println("Done")
```

You can get the full code from [here](#).

Notice how only two numbers get emitted by the flow in `foo()` function, producing the following output:

```
Emitting 1
1
Emitting 2
2
Done
```

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:

```
// Convert an integer range to a flow
(1..3).asFlow().collect { value -> println(value) }
```

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:

```
end fun performRequest(request: Int): String {
delay(1000) // imitate long-running asynchronous work
return "response $request"


main() = runBlocking<Unit> {
(1..3).asFlow() // a flow of requests
    .map { request -> performRequest(request) }
    .collect { response -> println(response) }
```

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:

```
(1..3).asFlow() // a flow of requests
    .transform { request ->
        emit("Making request $request")
        emit(performRequest(request))
    }
    .collect { response -> println(response) }
```

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:

```kotlin
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) }
}
```

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:

```
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)
```

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:

```
(1..5).asFlow()
    .filter {
        println("Filter $it")
        it % 2 == 0
    }
    .map {
        println("Map $it")
        "string $it"
    }.collect {
        println("Collect $it")
    }
```

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 `foo` flow, then the following code runs in the context specified by the author of this code, regardless of the implementation details of the `foo` flow:

```
withContext(context) {
    foo.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 `foo` that prints the thread it is called on and emits three numbers:

```
fun foo(): Flow<Int> = flow {
    log("Started foo flow")
    for (i in 1..3) {
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    foo().collect { value -> log("Collected $value") }
}
```

> You can get the full code from here.

Running this code produces:

```
[main @coroutine#1] Started foo flow
[main @coroutine#1] Collected 1
[main @coroutine#1] Collected 2
[main @coroutine#1] Collected 3
```

Since `foo().collect` is called from the main thread, the body of `foo`'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:

```
fun foo(): 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> {
    foo().collect { value -> println(value) }
}
```

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}@5511c7f8, BlockingEventLoop@2eac3323],
  but emission happened in [CoroutineId(1),
"coroutine#1":DispatchedCoroutine{Active}@2dae0000, DefaultDispatcher].
  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:

```
fun foo(): 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> {
    foo().collect { value ->
        log("Collected $value")
    }
}
```

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 `foo()` 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:

```
fun foo(): 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 {
        foo().collect { value ->
            delay(300) // pretend we are processing it for 300 ms
            println(value)
        }
    }
    println("Collected in $time ms")
}
```

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 `foo()` concurrently with collecting code, as opposed to running them sequentially:

```
val time = measureTimeMillis {
    foo()
        .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")
```

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:

```
val time = measureTimeMillis {
    foo()
        .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")
```

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
```

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:

```kotlin
val time = measureTimeMillis {
    foo()
        .collectLatest { value -> // cancel & restart on the latest value
            println("Collecting $value")
            delay(300) // pretend we are processing it for 300 ms
            println("Done $value")
        }
}
println("Collected in $time ms")
```

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:

```kotlin
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
```

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.

```
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")
    }
```

You can get the full code from here.

However, when using a combine operator here instead of a zip:

```
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")
    }
```

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 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:

```
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() - startTime} ms from start")
    }
```

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).

```
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")
    }
```

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.

```
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")
    }
```

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:

```kotlin
fun foo(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
    try {
        foo().collect { value ->
            println(value)
            check(value <= 1) { "Collected $value" }
        }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
```

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:

```
fun foo(): 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 {
        foo().collect { value -> println(value) }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
```

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:

```
foo()
    .catch { e -> emit("Caught $e") } // emit on exception
    .collect { value -> println(value) }
```

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:

```
fun foo(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    foo()
        .catch { e -> println("Caught $e") } // does not catch downstream exceptions
        .collect { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
}
```

A "Caught …" message is not printed despite there being a `catch` operator:

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:

```
foo()
    .onEach { value ->
        check(value <= 1) { "Collected $value" }
        println(value)
    }
    .catch { e -> println("Caught $e") }
    .collect()
```

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:

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.

```
fun foo(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
    try {
        foo().collect { value -> println(value) }
    } finally {
        println("Done")
    }
}
```

> You can get the full code from here.

This code prints three numbers produced by the `foo()` 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:

```
foo()
    .onCompletion { println("Done") }
    .collect { value -> println(value) }
```

> 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 `foo()` flow throws an exception after emitting the number 1:

```
fun foo(): Flow<Int> = flow {
    emit(1)
    throw RuntimeException()
}

fun main() = runBlocking<Unit> {
    foo()
        .onCompletion { cause -> if (cause != null) println("Flow completed
exceptionally") }
        .catch { cause -> println("Caught exception") }
        .collect { value -> println(value) }
}
```

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.

Upstream exceptions only

Just like the catch operator, onCompletion only sees exceptions coming from upstream and does not see downstream exceptions. For example, run the following code:

```
fun foo(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
    foo()
        .onCompletion { cause -> println("Flow completed with $cause") }
        .collect { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
}
```

You can get the full code from here.

We can see the completion cause is null, yet collection failed with exception:

```
1
Flow completed with null
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:

```
// 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")
}
```

> 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:

```
fun main() = runBlocking<Unit> {
    events()
        .onEach { event -> println("Event: $event") }
        .launchIn(this) // <--- Launching the flow in a separate coroutine
    println("Done")
}
```

> 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 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` for RxJava2). Integration modules include conversions from and to `Flow`, integration with Reactor's `Context` and suspension-friendly ways to work with various reactive entities.

Table of contents

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](#).

```
val channel = Channel<Int>()
launch {
    // this might be heavy CPU-consuming computation or async logic, we'll just send five squares
    for (x in 1..5) channel.send(x * x)
}
// here we print five received integers:
repeat(5) { println(channel.receive()) }
println("Done!")
```

You can get 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:

```
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x * x)
    channel.close() // we're done sending
}
// here we print received values using `for` loop (until the channel is closed)
for (y in channel) println(y)
println("Done!")
```

You can get 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:

```
val squares = produceSquares()
squares.consumeEach { println(it) }
println("Done!")
```

You can get 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:

```kotlin
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:

```kotlin
val numbers = produceNumbers() // produces integers from 1 and on
val squares = square(numbers) // squares integers
for (i in 1..5) println(squares.receive()) // print first five
println("Done!") // we are done
coroutineContext.cancelChildren() // cancel children coroutines
```

> You can get 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.

```kotlin
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:

```kotlin
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.

```
var cur = numbersFrom(2)
for (i in 1..10) {
    val prime = cur.receive()
    println(prime)
    cur = filter(cur, prime)
}
coroutineContext.cancelChildren() // cancel all children to let main finish
```

You can get 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:

```
val producer = produceNumbers()
repeat(5) { launchProcessor(it, producer) }
delay(950)
producer.cancel() // cancel producer coroutine and thus kill them all
```

You can get 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):

```
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
```

> You can get 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:

```
val channel = Channel<Int>(4) // create buffered channel
val sender = launch { // launch sender coroutine
    repeat(10) {
        println("Sending $it") // print before sending each element
        channel.send(it) // will suspend when buffer is full
    }
}
// don't receive anything... just wait....
delay(1000)
sender.cancel() // cancel sender coroutine
```

> You can get 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.

```kotlin
data class Ball(var hits: Int)

fun main() = runBlocking {
    val table = Channel<Ball>() // a shared table
    launch { player("ping", table) }
    launch { player("pong", table) }
    table.send(Ball(0)) // serve the ball
    delay(1000) // delay 1 second
    coroutineContext.cancelChildren() // game over, cancel them
}

suspend fun player(name: String, table: Channel<Ball>) {
    for (ball in table) { // receive the ball in a loop
        ball.hits++
        println("$name $ball")
        delay(300) // wait a bit
        table.send(ball) // send the ball back
    }
}
```

> You can get 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:

```kotlin
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") // initial delay
hasn't passed yet

    nextElement = withTimeoutOrNull(50) { tickerChannel.receive() } // all subsequent
elements has 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 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.

Table of contents

Exception Handling

This section covers exception handling and cancellation on exceptions. We already know that cancelled coroutine throws CancellationException in suspension points and that it is ignored by coroutines machinery. But 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). The former treat exceptions as unhandled, 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 coroutines in the GlobalScope:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job = GlobalScope.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 {
        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 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

But what if one does not want to print all exceptions to the console? CoroutineExceptionHandler context element is used as generic `catch` block of coroutine where custom logging or exception handling may take place. It is similar to using `Thread.uncaughtExceptionHandler`.

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 exceptions which are not expected to be handled by the user, so registering it in async builder and the like of it has no effect.

```
val handler = CoroutineExceptionHandler { _, exception ->
    println("Caught $exception")
}
val job = GlobalScope.launch(handler) {
    throw AssertionError()
}
val deferred = GlobalScope.async(handler) {
    throw ArithmeticException() // Nothing will be printed, relying on user to call
deferred.await()
}
joinAll(job, deferred)
```

You can get full code [here](#).

The output of this code is:

```
Caught java.lang.AssertionError
```

Cancellation and exceptions

Cancellation is tightly bound with 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.

```
val job = launch {
    val child = launch {
        try {
            delay(Long.MAX_VALUE)
        } finally {
            println("Child is cancelled")
        }
    }
    yield()
    println("Cancelling child")
    child.cancel()
    child.join()
    yield()
    println("Parent is not cancelled")
}
job.join()
```

You can get 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](#) which do not depend on [CoroutineExceptionHandler](#) implementation. The original exception is handled by the parent when all its children terminate.

> This also a reason why, 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.

```kotlin
val handler = CoroutineExceptionHandler { _, exception ->
    println("Caught $exception")
}
val job = GlobalScope.launch(handler) {
    launch { // the first child
        try {
            delay(Long.MAX_VALUE)
        } finally {
            withContext(NonCancellable) {
                println("Children are cancelled, but exception is not handled until all
children terminate")
                delay(100)
                println("The first child finished its non cancellable block")
            }
        }
    }
    launch { // the second child
        delay(10)
        println("Second child throws an exception")
        throw ArithmeticException()
    }
}
job.join()
```

> You can get 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
Caught java.lang.ArithmeticException
```

Exceptions aggregation

What happens if multiple children of a coroutine throw an exception? The general rule is "the first exception wins", so the first thrown exception is exposed to the handler. But that may cause lost exceptions, for example if coroutine throws an exception in its `finally` block. So, additional exceptions are suppressed.

> One of the solutions would have been to report each exception separately, but then [Deferred.await](#) should have had the same mechanism to avoid behavioural inconsistency and this would cause implementation details of a coroutines (whether it had delegated parts of its work to its children or not) to leak to its exception handler.

```
import kotlinx.coroutines.*
import java.io.*

fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("Caught $exception with suppressed
${exception.suppressed.contentToString()}")
    }
    val job = GlobalScope.launch(handler) {
        launch {
            try {
                delay(Long.MAX_VALUE)
            } finally {
                throw ArithmeticException()
            }
        }
        launch {
            delay(100)
            throw IOException()
        }
        delay(Long.MAX_VALUE)
    }
    job.join()
}
```

> You can get full code [here](#).

> Note: This above code will work properly only on JDK7+ that supports `suppressed` exceptions

The output of this code is:

```
Caught java.io.IOException with suppressed [java.lang.ArithmeticException]
```

> Note, this mechanism currently works only on Java version 1.7+. Limitation on JS and Native is temporary and will be fixed in the future.

Cancellation exceptions are transparent and unwrapped by default:

```
val handler = CoroutineExceptionHandler { _, exception ->
    println("Caught original $exception")
}
val job = GlobalScope.launch(handler) {
    val inner = launch {
        launch {
            launch {
                throw IOException()
            }
        }
    }
    try {
        inner.join()
    } catch (e: CancellationException) {
        println("Rethrowing CancellationException with original cause")
        throw e
    }
}
job.join()
```

You can get full code here.

The output of this code is:

```
Rethrowing CancellationException with original cause
Caught original java.io.IOException
```

Supervision

As we have studied before, cancellation is a bidirectional relationship propagating through the whole coroutines hierarchy. But what if 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 required.

Another example is a server process that spawns several children jobs and needs to *supervise* their execution, tracking their failures and restarting just those children jobs that had failed.

Supervision job

For these purposes SupervisorJob can be used. It is similar to a regular [Job](Job) with the only exception that cancellation is propagated only downwards. It is easy to demonstrate with an 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("First child is failing")
            throw AssertionError("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("First child is cancelled: ${firstChild.isCancelled}, but second one
 is still active")
            try {
                delay(Long.MAX_VALUE)
            } finally {
                // But cancellation of the supervisor is propagated
                println("Second child is cancelled because supervisor is cancelled")
            }
        }
        // wait until the first child fails & completes
        firstChild.join()
        println("Cancelling supervisor")
        supervisor.cancel()
        secondChild.join()
    }
}
```

You can get full code [here](#).

The output of this code is:

```
First child is failing
First child is cancelled: true, but second one is still active
Cancelling supervisor
Second child is cancelled because supervisor is cancelled
```

Supervision scope

For *scoped* concurrency supervisorScope can be used instead of coroutineScope for the same purpose. It propagates cancellation only in one direction and cancels all children only if it has failed itself. It also waits for all children before completion just like coroutineScope does.

```kotlin
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
    try {
        supervisorScope {
            val child = launch {
                try {
                    println("Child is sleeping")
                    delay(Long.MAX_VALUE)
                } finally {
                    println("Child is cancelled")
                }
            }
            // Give our child a chance to execute and print using yield
            yield()
            println("Throwing exception from scope")
            throw AssertionError()
        }
    } catch(e: AssertionError) {
        println("Caught assertion error")
    }
}
```

You can get full code [here](#).

The output of this code is:

```
Child is sleeping
Throwing exception from scope
Child is cancelled
Caught 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 exception handling mechanisms. This difference comes from the fact that child's failure is not propagated to the parent.

```kotlin
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("Caught $exception")
    }
    supervisorScope {
        val child = launch(handler) {
            println("Child throws an exception")
            throw AssertionError()
        }
        println("Scope is completing")
    }
    println("Scope is completed")
}
```

The output of this code is:

```
Scope is completing
Child throws an exception
Caught java.lang.AssertionError
Scope is completed
```

Table of contents

Shared mutable state and concurrency

Coroutines can be executed concurrently using a multi-threaded dispatcher like the
Dispatchers.Default. It presents all the usual concurrency 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.

```
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
```

You can get 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 common misconception that making a variable `volatile` solves concurrency problem. Let us try it:

```
@Volatile // in Kotlin `volatile` is an annotation
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
```

You can get 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 necessarily 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:

```
var counter = AtomicInteger()

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter.incrementAndGet()
        }
    }
    println("Counter = $counter")
}
```

> You can get 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.

```
val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            // confine each increment to a single-threaded context
            withContext(counterContext) {
                counter++
            }
        }
    }
    println("Counter = $counter")
}
```

> You can get 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.

```
val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking {
    // confine everything to a single-threaded context
    withContext(counterContext) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
```

> You can get 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:

```
val mutex = Mutex()
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            // protect each increment with lock
            mutex.withLock {
                counter++
            }
        }
    }
    println("Counter = $counter")
}
```

> You can get 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:

```kotlin
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
}
```

You can get 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.

Table of contents

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:

```kotlin
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:

```kotlin
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:

```
val fizz = fizz()
val buzz = buzz()
repeat(7) {
    selectFizzBuzz(fizz, buzz)
}
coroutineContext.cancelChildren() // cancel fizz & buzz coroutines
```

You can get 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 onReceiveOrNull 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.onReceiveOrNull { value ->
            if (value == null)
                "Channel 'a' is closed"
            else
                "a -> '$value'"
        }
        b.onReceiveOrNull { value ->
            if (value == null)
                "Channel 'b' is closed"
            else
                "b -> '$value'"
        }
    }
```

Note that [onReceiveOrNull](#) is an extension function defined only for channels with non-nullable elements so that there is no accidental confusion between a closed channel and a null value.

Let's use it with channel `a` that produces "Hello" string four times and channel `b` that produces "World" four times:

```
val a = produce<String> {
    repeat(4) { send("Hello $it") }
}
val b = produce<String> {
    repeat(4) { send("World $it") }
}
repeat(8) { // print first eight results
    println(selectAorB(a, b))
}
coroutineContext.cancelChildren()
```

> You can get full code [here](#).

The result of this code is quite interesting, so we'll analyze it in mode 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 onReceiveOrNull 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:

```
val side = Channel<Int>() // allocate side channel
launch { // this is a very fast consumer for the side channel
    side.consumeEach { println("Side channel has $it") }
}
produceNumbers(side).consumeEach {
    println("Consuming $it")
    delay(250) // let us digest the consumed number properly, do not hurry
}
println("Done consuming")
coroutineContext.cancelChildren()
```

> You can get 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.

```
val list = asyncStringsList()
val result = select<String> {
    list.withIndex().forEach { (index, deferred) ->
        deferred.onAwait { answer ->
            "Deferred $index produced answer '$answer'"
        }
    }
}
println(result)
val countActive = list.count { it.isActive }
println("$countActive coroutines are still active")
```

You can get 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 onReceiveOrNull 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.onReceiveOrNull { update ->
                update // replaces next value to wait
            }
            current.onAwait { value ->
                send(value) // send value that current deferred has produced
                input.receiveOrNull() // 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:

```
val chan = Channel<Deferred<String>>() // the channel for test
launch { // launch printing coroutine
    for (s in switchMapDeferreds(chan))
        println(s) // print each received string
}
chan.send(asyncString("BEGIN", 100))
delay(200) // enough time for "BEGIN" to be produced
chan.send(asyncString("Slow", 500))
delay(100) // not enough time to produce slow
chan.send(asyncString("Replace", 100))
delay(500) // give it time before the last one
chan.send(asyncString("END", 500))
delay(1000) // give it time to process
chan.close() // close the channel ...
delay(500) // and wait some time to let it finish
```

You can get full code here.

The result of this code:

```
BEGIN
Replace
END
Channel was closed
```

Tools

Documenting Kotlin Code

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.

Generating 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.

Linking 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.
```

Annotation Processing with Kotlin

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:

```
plugins {
    id "org.jetbrains.kotlin.kapt" version "1.3.61"
}
```

```
plugins {
    kotlin("kapt") version "1.3.61"
}
```

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:

```
dependencies {
    kapt 'groupId:artifactId:version'
}
```

```
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 (since 1.2.20)

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
}
```

Running kapt tasks in parallel (since 1.2.60)

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
```

Compile avoidance for kapt (since 1.3.20)

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 (since 1.3.30)

Starting from version 1.3.30, kapt supports incremental annotation processing as an experimental feature. Currently, annotation processing can be incremental only if all annotation processors being used are incremental.

Incremental annotation processing is enabled by default starting from version 1.3.50. 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 additional flag 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())
}
```

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. For example:

```
$ kotlinc hello.kt -include-runtime -d hello.jar
```

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 Using Gradle.

— If you're using Maven, specify the compiler arguments in the `<configuration>` element of the Maven plugin node. For details, see Using Maven.

— If you run a command-line compiler, add the compiler arguments directly to the utility call or write them into an argfile.

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.

-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 [Compiler plugins](#).

-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 `kotlinc` and `kotlinc-jvm`. You can also use them for executing Kotlin script files.

In addition to the [common options](#), Kotlin/JVM compiler has the options listed below.

-classpath <path>(-cp <path>)

Search for class files in the specified paths. Separate elements of the classpath with system path separators (**;** on Windows, **:** on macOS/Linux). The classpath can contain file and directory paths, ZIP, or JAR files.

-d <path>

Place the generated class files into the specified location. The location can be a directory, a ZIP, or a JAR file.

-include-runtime

Include the Kotlin runtime into the resulting JAR file. Makes the resulting archive runnable on any Java-enabled environment.

-jdk-home <path>

Use a custom JDK home directory to include into the classpath if it differs from the default `JAVA_HOME`.

-jvm-target <version>

Specify the target version of the generated JVM bytecode. Possible values are `1.6`, `1.8`, `9`, `10`, `11`, and `12`. The default value is `1.6`.

`-java-parameters`

Generate metadata for Java 1.8 reflection on method parameters.

`-module-name <name>`

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`

Don't automatically include the Kotlin/JVM stdlib (`kotlin-stdlib.jar`) and Kotlin reflection (`kotlin-reflect.jar`) into the classpath.

`-script`

Evaluate a Kotlin script file. When called with this option, the compiler executes the first Kotlin script (`*.kts`) file among the given arguments.

`-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 {plain|amd|commonjs|umd}

The kind of JS module generated by the compiler:

— `plain` - a plain JS module;

— `commonjs` - a CommonJS module;

— `amd` - an Asynchronous Module Definition module;

— `umd` - a Universal Module Definition module.

To learn more about the different kinds of JS module and the distinctions between them, see this article.

-no-stdlib

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)`

Set the library version.

`-list-targets`

List the available hardware targets.

`-manifest <path>`

Provide a manifest addend file.

`-module-name <name>`

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)

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.

Using Gradle

In order to build a Kotlin project with Gradle, you should set up the *kotlin-gradle* plugin, apply it to your project and add *kotlin-stdlib* dependencies. Those actions may also be performed automatically in IntelliJ IDEA by invoking **Tools | Kotlin | Configure Kotlin** in **Project** action.

Plugin and Versions

Apply the Kotlin Gradle plugin by using the Gradle plugins DSL. The Kotlin Gradle plugin 1.3.61 works with Gradle 4.9 and later.

```
plugins {
    id 'org.jetbrains.kotlin.<...>' version '1.3.61'
}
```

```
plugins {
    kotlin("<...>") version "1.3.61"
}
```

The placeholder `<...>` should be replaced with one of the plugin names that can be found in further sections.

Alternatively, apply plugin by adding the `kotlin-gradle-plugin` dependency to the build script classpath:

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.3.61"
    }
}

plugins {
    id "org.jetbrains.kotlin.<...>" version "1.3.61"
}
```

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath(kotlin("gradle-plugin", version = "1.3.61"))
    }
}
plugins {
    kotlin("<...>")
}
```

This is not required when using Kotlin Gradle plugin 1.1.1 and above with the Gradle plugins DSL, and with Gradle Kotlin DSL.

Building Kotlin Multiplatform Projects

Using the `kotlin-multiplatform` plugin for building [multiplatform projects](#) is described in [Building Multiplatform Projects with Gradle](#).

Targeting the JVM

To target the JVM, apply the Kotlin JVM plugin. Starting with Kotlin 1.1.1, the plugin can be applied using the [Gradle plugins DSL](#):

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "1.3.61"
}
```

```
plugins {
    kotlin("jvm") version "1.3.61"
}
```

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 to apply Kotlin plugins with `apply` in Gradle Kotlin DSL. The details are provided [below](#).

Kotlin sources can be mixed with Java sources in the same folder, or in 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:

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

```
sourceSets["main"].java.srcDir("src/main/myJava")
sourceSets["main"].withConvention(KotlinSourceSet::class) {
    kotlin.srcDir("src/main/myKotlin")
}
```

With Gradle Kotlin DSL, configure source sets with `java.sourceSets { ... }` instead.

Targeting JavaScript

When targeting JavaScript, a different plugin should be applied:

```
plugins {
    id 'org.jetbrains.kotlin.js' version '1.3.61'
}
```

```
plugins {
    kotlin("js") version "1.3.61"
}
```

This plugin only works for Kotlin files so it is recommended to keep Kotlin and Java files separate (in case if the same project contains Java files). As with targeting the JVM, if not using the default convention, you should specify the source folder using *sourceSets*:

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
}
```

```
sourceSets["main"].withConvention(KotlinSourceSet::class) {
    kotlin.srcDir("src/main/myKotlin")
}
```

In addition to the output JavaScript file, the plugin by default creates an additional JS file with binary descriptors. This file is required if you're building a reusable library that other Kotlin modules can depend on, and should be distributed together with the result of translation. The generation is controlled by the `kotlinOptions.metaInfo` option:

```
compileKotlin2Js {
    kotlinOptions.metaInfo = true
}
```

```
tasks {
    "compileKotlin2Js"(Kotlin2JsCompile::class)  {
        kotlinOptions.metaInfo = true
    }
}
```

Targeting Android

Android's Gradle model is a little different from ordinary Gradle, so if we want to build an Android project written in Kotlin, we need *kotlin-android* plugin instead of *kotlin*:

```
buildscript {
    ext.kotlin_version = '1.3.61'

    ...

    dependencies {
        classpath 'com.android.tools.build:gradle:3.2.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

plugins {
    id 'com.android.application'
    id 'kotlin-android'
}
```

```
buildscript {
    dependencies {
        classpath("com.android.tools.build:gradle:3.2.1")
        classpath(kotlin("gradle-plugin", version = "1.3.61"))
    }
}
plugins {
    id("com.android.application")
    kotlin("android")
}
```

Kotlin Gradle plugin 1.3.61 works with Android Gradle Plugin 3.0 and later.

Don't forget to configure the standard library dependency.

Android Studio

If using Android Studio, the following needs to be added under android:

```
android {
  ...

  sourceSets {
    main.java.srcDirs += 'src/main/kotlin'
  }
}
```

```
android {
  ...

    sourceSets["main"].java.srcDir("src/main/kotlin")
}
```

This lets Android Studio know that the kotlin directory is a source root, so when the project model is loaded into the IDE it will be properly recognized. Alternatively, you can put Kotlin classes in the Java source directory, typically located in `src/main/java`.

Configuring Dependencies

In addition to the `kotlin-gradle-plugin` dependency shown above, you need to add a dependency on the Kotlin standard library:

```
repositories {
    mavenCentral()
}

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

```
repositories {
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib"))
}
```

The Kotlin standard library `kotlin-stdlib` targets Java 6 and above. There are extended versions of the standard library that add support for some of the features of JDK 7 and JDK 8. To use these versions, add one of the following dependencies instead of `kotlin-stdlib`:

```
implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7"
implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
```

```
implementation(kotlin("stdlib-jdk7"))
implementation(kotlin("stdlib-jdk8"))
```

In Kotlin 1.1.x, use `kotlin-stdlib-jre7` and `kotlin-stdlib-jre8` instead.

If you target JavaScript, use the `stdlib-js` dependency.

```
implementation "org.jetbrains.kotlin:kotlin-stdlib-js"
```

```
implementation(kotlin("stdlib-js"))
```

If your project uses [Kotlin reflection](#) or testing facilities, you need to add the corresponding dependencies as well:

```
implementation "org.jetbrains.kotlin:kotlin-reflect"
testImplementation "org.jetbrains.kotlin:kotlin-test"
testImplementation "org.jetbrains.kotlin:kotlin-test-junit"
```

```
implementation(kotlin("reflect"))
testImplementation(kotlin("test"))
testImplementation(kotlin("test-junit"))
```

Starting with Kotlin 1.1.2, the dependencies with group `org.jetbrains.kotlin` are by default resolved with the version taken from the applied plugin. You can provide the version manually using the full dependency notation:

```
implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
```

```
implementation(kotlin("stdlib", kotlinVersion))
```

Annotation Processing

Kotlin supports annonation processing via the *Kotlin annotation processing tool*(`kapt`). Usage of kapt with Gradle is described on the [kapt page](#).

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. It's enabled by default since Kotlin 1.1.1 for Kotlin/JVM and 1.3.20 for Kotlin/JS.

There are several ways to override the default setting:

— In Gradle configuration files: add the line `kotlin.incremental=<value>` for Kotlin/JVM or `kotlin.incremental.js=<value>` for Kotlin/JS projects either to `gradle.properties` or to `local.properties` file. `<value>` is a boolean value reflecting the usage of incremental compilation.

— In Gradle command line parameters: add the parameter `-Pkotlin.incremental` or `-Pkotlin.incremental.js` with the boolean value reflecting the usage of incremental compilation. 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 (since 1.2.20)

The Kotlin plugin supports [Gradle Build Cache](#) (Gradle version 4.3 and above is required; caching is disabled with lower versions).

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. See the [kapt page](#) for details.

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 `compileKotlin2Js` and `compileTestKotlin2Js` respectively, and `compile<Name>Kotlin2Js` for custom source sets.

To configure a single task, use its name. Examples:

```
compileKotlin {
    kotlinOptions.suppressWarnings = true
}

//or

compileKotlin {
    kotlinOptions {
        suppressWarnings = true
    }
}
```

```
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:

```
tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompile).configureEach {
    kotlinOptions { ... }
}
```

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

tasks.withType<KotlinCompile>().configureEach {
    kotlinOptions.suppressWarnings = true
}
```

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	"1.0", "1.1", "1.2", "1.3", "1.4 (EXPERIMENTAL)"	
languageVersion	Provide source compatibility with the specified version of Kotlin	"1.0", "1.1", "1.2", "1.3", "1.4 (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, 1.8, 9, 10, 11 or 12), default is 1.6	"1.6", "1.8", "9", "10", "11", "12"	"1.6"
noJdk	Don't automatically include the Java runtime into the classpath		false
noReflect	Don't automatically include Kotlin reflection into the classpath		true
noStdlib	Don't automatically include the Kotlin/JVM stdlib and Kotlin reflection into the classpath		true

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	"plain", "amd", "commonjs", "umd"	"plain"
`noStdlib`	Don't automatically include the default Kotlin/JS stdlib into compilation dependencies		true
`outputFile`	Destination *.js file for the compilation result		
`sourceMap`	Generate source map		false
`sourceMapEmbedSources`	Embed source files into source map	"never", "always", "inlining"	
`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.

Examples

The following examples show different possibilities of configuring the Gradle plugin:

— [Kotlin](#)

— [Mixed Java and Kotlin](#)

— [Android](#)

— [JavaScript](#)

Using 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>1.3.61</kotlin.version>
</properties>
```

Dependencies

Kotlin has an extensive standard library that can be used in your applications. Configure 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 which 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 1.1.x use `kotlin-stdlib-jre7` and `kotlin-stdlib-jre8` as the `jdk` counterparts were introduced in 1.2.0).

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.

Compiling Kotlin only source code

To compile source code, specify the source directories in the 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:

```xml
<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>
```

Compiling Kotlin and Java sources

To compile mixed code applications Kotlin compiler should be invoked before Java compiler. In maven terms that means kotlin-maven-plugin should be run before maven-compiler-plugin using the following method, making sure that the kotlin plugin is above the maven-compiler-plugin in your pom.xml file:

```xml
<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 (supported since Kotlin 1.1.2). In order to do that, define 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`).

Coroutines support

[Coroutines] support is an experimental feature in Kotlin 1.2, so the Kotlin compiler reports a warning when you use coroutines in your project. To turn off the warning, add the following block to your `pom.xml` file:

```
<configuration>
    <experimentalCoroutines>enable</experimentalCoroutines>
</configuration>
```

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:

```xml
<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
```

Targeting JavaScript

In order to compile JavaScript code, you need to use the `js` and `test-js` goals for the `compile` execution:

```xml
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <executions>
        <execution>
            <id>compile</id>
            <phase>compile</phase>
            <goals>
                <goal>js</goal>
            </goals>
        </execution>
        <execution>
            <id>test-compile</id>
            <phase>test-compile</phase>
            <goals>
                <goal>test-js</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

You also need to change the standard library dependency:

```
<groupId>org.jetbrains.kotlin</groupId>
<artifactId>kotlin-stdlib-js</artifactId>
<version>${kotlin.version}</version>
```

For unit testing support, you also need to add a dependency on the `kotlin-test-js` artifact.

See the [Getting Started with Kotlin and JavaScript with Maven](#) tutorial for more information.

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 specified language version	"1.0", "1.1", "1.2", "1.3", "1.4 (EXPERIMENTAL)"	
apiVersion	kotlin.compiler.apiVersion	Allow to use declarations only from the specified version of bundled libraries	"1.0", "1.1", "1.2", "1.3", "1.4 (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", "1.8", "9", "10", "11", "12"	"1.6"
jdkHome	kotlin.compiler.jdkHome	Path to JDK home directory to include into classpath, if differs from default JAVA_HOME		

Attributes specific for JS

Name	Property name	Description	Possible values	Default value
outputFile		Output file path		
metaInfo		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"
sourceMapPrefix		Prefix for paths in a source map		
moduleKind		Kind of a module generated by compiler	"plain", "amd", "commonjs", "umd"	"plain"

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 directly from the GitHub repository](#)

Using 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 for the [Kotlin Compiler](#) 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
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`.

Kotlin and OSGi

To enable Kotlin OSGi support you need to include `kotlin-osgi-bundle` instead of 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:

```xml
<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):

```xml
<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:

```groovy
compile "org.jetbrains.kotlin:kotlin-osgi-bundle:$kotlinVersion"
```

To exclude default Kotlin libraries that comes as transitive dependencies you can use the following approach:

```groovy
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.

Compiler Plugins

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.

Note: For Spring you can use the `kotlin-spring` compiler plugin ([see below](#)).

Using in Gradle

Add the plugin artifact to the buildscript 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 "1.3.61"
}
```

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.

Using in 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 "Using in Gradle" section above 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 buildscript 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 "1.3.61"
}
```

In Maven, enable the `spring` plugin:

```
<compilerPlugins>
    <plugin>spring</plugin>
</compilerPlugins>
```

The plugin specifies the following annotations: @Component, @Async, @Transactional, @Cacheable and @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.

Using in CLI

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](#)).

Using in Gradle

The usage is pretty similar to all-open.

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 "1.3.61"
}
```

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. Starting from Kotlin 1.1.3-2, it is disabled by default because of [KT-18667](#) and [KT-18668](#) which will be addressed in the future.

```
noArg {
    invokeInitializers = true
}
```

Using in Maven

```xml
<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, *kotlin-jpa* is a 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 "1.3.61"
}
```

In Maven, enable the `jpa` plugin:

```xml
<compilerPlugins>
    <plugin>jpa</plugin>
</compilerPlugins>
```

Using in CLI

As with all-open, 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 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")
    }
}
```

Using in 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")
}
```

Using in Maven

```xml
<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>
```

Using in CLI

Just 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
```

Code Style Migration Guide

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

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 Code 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>
```

Warning: 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.

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 please 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.

Evolution

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 with *experimental* status. Experimental 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 Andrey Breslav). 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 Andrey Breslav).

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. Experimental 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 experimental 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.

Experimental 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 the *experimental* status and *are supposed to change*. Experimental features can be added, changed or removed at any point and without warning. We make sure that experimental 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.

Experimental 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 @Experimental 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 experimental 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 behaviour 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 Different Components

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 1.3

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructions 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

One has to 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.

Evaluation order of constructor arguments regarding `<clinit>` call

Issue: [KT-19532](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](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 case code which was (erroneously) passing `null`s here fail faster.
  `-XXLanguage:-StrictJavaNullabilityAssertions` can be used to temporarily return to the pre-1.3 behavior. Support for this flag will be removed in the next major release.

Unsound smartcasts on enum members

Issue: [KT-20772](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: <u>KT-16681</u>

Components: Core language

Incompatible change type: Source

Short summary: reassignment of the backing field of `val`-property in its getter is now prohibited

Deprecation cycle:

— <1.2: Kotlin compiler allowed to modify backing field of `val` in its getter. Not only it violates Kotlin semantic, but also generates ill-behaved JVM bytecode which reassigns `final` field.

— 1.2.X: deprecation warning is reported on code which reassigns backing field of `val`

— >=1.3: deprecation warnings are elevated to errors

Array capturing before the `for`-loop where it is iterated

Issue: <u>KT-21354</u>

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

Data class overriding copy

Inner classes inheriting `Throwable` that capture generic parameters from the outer class

Issue: [KT-17981](#)

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, inner classes are not allowed to inherit `Throwable`

Deprecation cycle:

— <1.2: inner classes inheriting `Throwable` are compiled fine. If such inner classes happen to capture generic parameters, it could lead to strange code patterns which fail at runtime.

— 1.2.X: deprecation warnings reported on inner classes inheriting `Throwable`

— >=1.3: deprecation warnings elevated to errors

Visibility rules regarding complex class hierarchies with companion objects

Issues: [KT-21515](#), [KT-25333](#)

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, rules of visibility by short names are stricter for complex class hierarchies involving companion objects and nested classifiers.

Deprecation cycle:

— <1.2: old visibility rules (see Issue for details)

— 1.2.X: deprecation warnings reported on short names which are not going to be accessible anymore. Tooling suggests automated migration by adding full name.

— >=1.3: deprecation warnings elevated to errors. Offending code should add full qualifiers or explicit imports

Non-constant vararg annotation parameters

Issue: KT-23153

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, setting non-constant values as vararg annotation parameters is prohibited

Deprecation cycle:

— <1.2: the compiler allows to pass non-constant value for vararg annotation parameter, but actually drops that value during bytecode generation, leading to non-obvious behavior
— 1.2.X: deprecation warnings reported on such code patterns
— >=1.3: deprecation warnings elevated to errors

Local annotation classes

Issue: KT-23277

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 local annotation classes are not supported

Deprecation cycle:

— <1.2: the compiler compiled local annotation classes fine
— 1.2.X: deprecation warnings reported on local annotation classes
— >=1.3: deprecation warnings elevated to errors

Smartcasts on local delegated properties

Issue: KT-22517

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 smartcasts on local delegated properties are not allowed

Deprecation cycle:

— <1.2: the compiler allowed to smartcast local delegated property, which could lead to unsound smartcast in case of ill-behaved delegates
— 1.2.X: smartcasts on local delegated properries are reported as deprecated (the compiler issues warnings)
— >=1.3: deprecation warnings elevated to errors

mod operator convention

Issues: KT-24197

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 declaration of mod operator is prohibited, as well as calls which resolve to such declarations

Deprecation cycle:

— 1.1.X, 1.2.X: report warnings on declarations of `operator mod`, as well as on calls which resolve to it
— 1.3.X: elevate warnings to error, but still allow to resolve to `operator mod` declarations
— 1.4.X: do not resolve calls to `operator mod` anymore

Passing single element to vararg in named form

Issues: KT-20588, KT-20589. See also KT-20171

Component: Core language

Incompatible change type: Source

Short summary: in Kotlin 1.3, assigning single element to vararg is deprecated and should be replaced with consecutive spread and array construction.

Deprecation cycle:

— <1.2: assigning one value element to vararg in named form compiles fine and is treated as assigning *single* element to array, causing non-obvious behavior when assigning array to vararg
— 1.2.X: deprecation warnings are reported on such assignments, users are suggested to switch to consecutive spread and array construction.
— 1.3.X: warnings are elevated to errors
— >= 1.4: change semantic of assigning single element to vararg, making assignment of array equivalent to the assignment of a spread of an array

Retention of annotations with target EXPRESSION

Issue: KT-13762

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, only `SOURCE` retention is allowed for annotations with target `EXPRESSION`

Deprecation cycle:

— <1.2: annotations with target `EXPRESSION` and retention other than `SOURCE` are allowed, but silently ignored at use-sites
— 1.2.X: deprecation warnings are reported on declarations of such annotations
— >=1.3: warnings are elevated to errors

Annotations with target PARAMETER shouldn't be applicable to parameter's type

Issue: KT-9580

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, error about wrong annotation target will be properly reported when annotation with target `PARAMETER` is applied to parameter's type

Deprecation cycle:

— <1.2: aforementioned code patterns are compiled fine; annotations are silently ignored and not present in the bytecode
— 1.2.X: deprecation warnings are reported on such usages
— >=1.3: warnings are elevated to errors

`Array.copyOfRange` throws an exception when indices are out of bounds instead of enlarging the returned array

Issue: KT-19489

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, ensure that the `toIndex` argument of `Array.copyOfRange`, which represents the exclusive end of the range being copied, is not greater than the array size and throw `IllegalArgumentException` if it is.

Deprecation cycle:

— <1.3: in case `toIndex` in the invocation of `Array.copyOfRange` is greater than the array size, the missing elements in range fill be filled with `null`s, violating soundness of the Kotlin type system.
— >=1.3: check that `toIndex` is in the array bounds, and throw exception if it isn't

Progressions of ints and longs with a step of `Int.MIN_VALUE` and `Long.MIN_VALUE` are outlawed and won't be allowed to be instantiated

Issue: [KT-17176](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, prohibit step value for integer progressions being the minimum negative value of its integer type (`Long` or `Int`), so that calling `IntProgression.fromClosedRange(0, 1, step = Int.MIN_VALUE)` will throw `IllegalArgumentException`

Deprecation cycle:

— <1.3: it was possible to create an `IntProgression` with `Int.MIN_VALUE` step, which yields two values [`0, -2147483648`], which is non-obvious behavior

— >=1.3: throw `IllegalArgumentException` if the step is the minimum negative value of its integer type

Check for index overflow in operations on very long sequences

Issue: [KT-16097](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, make sure `index`, `count` and similar methods do not overflow for long sequences. See the Issue for the full list of affected methods.

Deprecation cycle:

— <1.3: calling such methods on very long sequences could produce negative results due to integer overflow

— >=1.3: detect overflow in such methods and throw exception immediately

Unify split by an empty match regex result across the platforms

Issue: KT-21049

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, unify behavior of `split` method by empty match regex across all platforms

Deprecation cycle:

— <1.3: behavior of described calls is different when comparing JS, JRE 6, JRE 7 versus JRE 8+
— >=1.3: unify behavior across the platforms

Discontinued deprecated artifacts in the compiler distribution

Issue: KT-23799

Component: other

Incompatible change type: Binary

Short summary: Kotlin 1.3 discontinues the following deprecated binary artifacts:

— `kotlin-runtime`: use `kotlin-stdlib` instead
— `kotlin-stdlib-jre7/8`: use `kotlin-stdlib-jdk7/8` instead
— `kotlin-jslib` in the compiler distribution: use `kotlin-stdlib-js` instead

Deprecation cycle:

— 1.2.X: the artifacts were marked as deprecated, the compiler reported warning on usage of those artifacts
— >=1.3: the artifacts are discontinued

Annotations in stdlib

Issue: [KT-21784](#)

Component: kotlin-stdlib (JVM)

Incompatible change type: Binary

Short summary: Kotlin 1.3 removes annotations from the package `org.jetbrains.annotations` from stdlib and moves them to the separate artifacts shipped with the compiler: `annotations-13.0.jar` and `mutability-annotations-compat.jar`

Deprecation cycle:

— <1.3: annotations were shipped with the stdlib artifact

— >=1.3: annotations ship in separate artifacts

FAQ

FAQ

What is Kotlin?

Kotlin is an OSS 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 1.3.61, published on November 27, 2019.

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-nullable types makes applications less prone to NPE's. Other features including smart casting, higher-order functions, extension functions and lambdas with receivers provide the ability to write expressive code as well as facilitating creation of DSL.

Is Kotlin compatible with the Java programming language?

Yes. Kotlin is 100% interoperable with the Java programming language and major emphasis has been placed on making sure that your existing codebase can interact properly with Kotlin. You can easily call Kotlin code from Java and Java code from Kotlin. This makes adoption much easier and lower-risk. There's also an automated Java-to-Kotlin converter built into the IDE that simplifies migration of existing code.

What can I use Kotlin for?

Kotlin can be used for any kind of development, be it 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?

Kotlin/Native is currently in the works. It compiles Kotlin to native code that can run without a VM. There is a Technology Preview released but it is not production-ready yet, and doesn't yet target all the platforms that we plan to support for 1.0. For more information check out the blog post announcing Kotlin/Native.

What IDEs support Kotlin?

Kotlin is supported by all major Java IDEs including IntelliJ IDEA, Android Studio, Eclipse and NetBeans. 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 6 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 8 to 12. 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 or Corda.

Who develops Kotlin?

Kotlin is primarily developed by a team of engineers at JetBrains (current team size is 90+). The lead language designer is Andrey Breslav. 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, [reference documentation]() and [tutorials]().

Are there any books on Kotlin?

There are already [a number of books]() available for Kotlin, including [Kotlin in Action]() which is by Kotlin team members Dmitry Jemerov and Svetlana Isakova, [Kotlin for Android Developers]() targeted at Android developers.

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 20000 members as of October 2018).

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 organised [Kotlin Nights]() events around the world.

Is there a Kotlin conference?

Yes. The official annual [KotlinConf]() is hosted by JetBrains. It has taken place in San-Francisco in [2017]() and in Amsterdam in 2018. 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](#).

Comparison to Java Programming Language

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

— [Static members](#)

— [Non-private fields](#)

— [Wildcard-types](#)

— [Ternary-operator a ? b : c](#)

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](#)