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.

— Micronaut is a modern, JVM-based, full-stack framework for building modular, easily testable microservice and serverless applications. It comes with a lot of built-in, handy features.

— Javalin is a very lightweight web framework for Kotlin and Java which supports WebSockets, HTTP2 and async requests.

— The available options for persistence include direct JDBC access, JPA, as well as using NoSQL databases through their Java drivers. For JPA, the kotlin-jpa compiler plugin adapts Kotlin-compiled classes to the requirements of the framework.

Deploying Kotlin Server-side Applications

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

To deploy Kotlin applications on Heroku, you can follow the official Heroku tutorial.

AWS Labs provides a sample project showing the use of Kotlin for writing AWS Lambda functions.

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

Users of Kotlin on the Server Side

Corda is an open-source distributed ledger platform, supported by major banks, and built entirely in Kotlin.

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

Next Steps

— 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.

— Micronaut also has a lot of well-detailed guides, showing how you can build microservices in Kotlin.

Using Kotlin for Android Development

Android mobile development has been Kotlin-first since Google I/O in 2019.

Using Kotlin for Android development, you can benefit from:

— **Less code combined with greater readability**. Spend less time writing your code and working to understand the code of others.

— **Mature language and environment**. Since its creation in 2011, Kotlin has developed continuously, not only as a language but as a whole ecosystem with robust tooling. Now it's seamlessly integrated in Android Studio and is actively used by many companies for developing Android applications.

— **Kotlin support in Android Jetpack and other libraries**. KTX extensions add Kotlin language features, such as coroutines, extension functions, lambdas, and named parameters, to existing Android libraries.

— **Interoperability with Java**. You can use Kotlin along with the Java programming language in your applications without needing to migrate all your code to Kotlin.

— **Support for multiplatform development**. You can use Kotlin for developing not only Android but also iOS, backend, and web applications. Enjoy the benefits of sharing the common code among the platforms.

— **Code safety**. Less code and better readability lead to fewer errors. The Kotlin compiler detects these remaining errors, making the code safe.

— **Easy learning**. Kotlin is very easy to learn, especially for Java developers.

— **Big community**. Kotlin has great support and many contributions from the community, which is growing all over the world. According to Google, over 60% of the top 1000 apps on the Play Store use Kotlin.

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

If you want to start using Kotlin for Android development, read Google's recommendation for getting started with Kotlin on Android.

If you're new to Android and want to learn to create applications with Kotlin, check out this Udacity course.

Kotlin/JS Overview

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

The recommended way to use Kotlin/JS is via the `kotlin.js` and `kotlin.multiplatform` Gradle plugins. They provide a central and convenient way to set up and control Kotlin projects targeting JavaScript. This includes essential functionality such as controlling the bundling of your application, adding JavaScript dependencies directly from npm, and more. To get an overview of the available options, check out the Kotlin/JS project setup documentation.

Some use cases for Kotlin/JS

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

- **Write frontend web applications using Kotlin/JS**
  - Kotlin/JS allows you to **leverage powerful browser and web APIs** in a type-safe fashion. Create, modify and interact with elements in the Document Object Model (DOM), use Kotlin code to control the rendering of `canvas` or WebGL components, and enjoy access to many more of the features supported in modern browsers.
  - Write **full, type-safe React applications with Kotlin/JS** using the `kotlin-wrappers` provided by JetBrains, which provide convenient abstractions and deep integrations for one of the most popular JavaScript frameworks. `kotlin-wrappers` also provides support for a select number of adjacent technologies like `react-redux`, `react-router`, or `styled-components`. Interoperability with the JavaScript ecosystem also means that you can also use third-party React components and component libraries.
  - Or, use **community-maintained Kotlin/JS frameworks** that take full advantage of Kotlin concepts, its expressive power and conciseness – like kvision or fritz2.
- **Write server-side and serverless applications using Kotlin/JS**
  - The Node.js target provided by Kotlin/JS enables you to create applications that **run on a server** or get **executed on serverless infrastructure**. You benefit from the same advantages as other applications executing in a JavaScript runtime, such as **faster startup speed** and a **reduced memory footprint**. With `kotlinx-nodejs`, you have typesafe access to the Node.js API directly from your Kotlin code.
- **Use Kotlin's multiplatform projects to share code with other Kotlin targets**
  - All Kotlin/JS functionality can also be accessed when using the Kotlin `multiplatform` Gradle plugin.
  - If you have a backend written in Kotlin, you can **share common code** such as data models or validation logic with a frontend written in Kotlin/JS, allowing you to **write and maintain full-stack web applications**.
  - You could also **share business logic between your web interface and mobile apps** for Android and iOS, and avoid duplicating commonly used functionality like providing abstractions around REST API endpoints, user authentication, or your domain models.
- **Create libraries for use with JavaScript and TypeScript**
  - You don't have to write your whole application in Kotlin/JS, either – you can also **generate libraries**

from your Kotlin code that can be consumed as modules from any code base written in JavaScript or TypeScript, regardless of other frameworks or technologies used. This approach of **creating hybrid applications** allows you to leverage the competencies that you and your team might already have around web development, while helping you **reduce the amount of duplicated work**, and making it easier to keep your web target consistent with other targets of your application.

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

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

Kotlin/JS is also compatible with the most common module systems: UMD, CommonJS, and AMD. Being able to produce and consume modules means that you can interact with the JavaScript ecosystem in a structured manner.

Kotlin/JS, Today and Tomorrow

Want to know more about Kotlin/JS?

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

Getting Started with Kotlin/JS

If you're new to Kotlin, a good first step would be to familiarise yourself with the Basic Syntax of the language.

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

Hands-on labs for Kotlin/JS

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

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

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

— Building Web Applications with React and Kotlin/JS guides you through the process of building a simple web application using the React framework, shows how a typesafe Kotlin DSL for HTML makes it convenient to build reactive DOM elements, and illustrates how to use third-party React components, and how to obtain information from APIs, while writing the whole application logic in pure Kotlin/JS.

— Building a Full Stack Web App with Kotlin Multiplatform teaches the concepts behind building an application that targets Kotlin/JVM and Kotlin/JS by building a client-server application that makes use of common code, serialization, and other multiplatform paradigms. It also provides a brief introduction into working with Ktor both as a server- and client-side framework.

New Kotlin/JS IR compiler

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

To learn more about the available features in the new Kotlin/JS IR compiler and how to try it for your project, visit the documentation.

Join the Kotlin/JS community

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

Kotlin/Native 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)

— watchOS (arm32, arm64, x86)

— tvOS (arm64, x86_64)

— Android (arm32, arm64, x86, x86_64)

— Windows (mingw x86_64, x86)

— Linux (x86_64, arm32, arm64, MIPS, MIPS little endian)

— 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 allow sharing common Kotlin code between multiple platforms, including Android, iOS, JVM, JavaScript, and native. Multiplatform libraries provide required APIs for the common Kotlin code and help develop shared parts of a project in Kotlin code in one place and share it with all or several target platforms.

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

What's next?

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

Documentation

— Kotlin Multiplatform Mobile documentation
— Multiplatform documentation
— C interop
— Swift/Objective-C interop

Tutorials

— Hello Kotlin/Native
— Types mapping between C and Kotlin/Native
— Kotlin/Native as a Dynamic Library
— Kotlin/Native as an Apple Framework

Sample projects

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

Kotlin for Data Science

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

— Kotlin is concise, readable and easy to learn.

— Static typing and null safety help create reliable, maintainable code that is easy to troubleshoot.

— Being a JVM language, Kotlin gives you great performance and an ability to leverage an entire ecosystem of tried and true Java libraries.

Interactive editors

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

Jupyter Kotlin kernel

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

Check out Kotlin kernel's GitHub repo for installation instructions, documentation, and examples.

Zeppelin Kotlin interpreter

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

Libraries

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

Kotlin libraries

— kotlin-statistics is a library providing extension functions for exploratory and production statistics. It supports basic numeric list/sequence/array functions (from `sum` to `skewness`), slicing operators (such as `countBy`, `simpleRegressionBy`), binning operations, discrete PDF sampling, naive bayes classifier, clustering, linear regression, and much more.

— kmath is a library inspired by NumPy. This library supports algebraic structures and operations, array-like structures, math expressions, histograms, streaming operations, a wrapper around commons-math and koma, and more.

— krangl is a library inspired by R's dplyr and Python's pandas. This library provides functionality for data manipulation using a functional-style API; it also includes functions for filtering, transforming, aggregating, and reshaping tabular data.

— lets-plot is a plotting library for statistical data written in Kotlin. Lets-Plot is multiplatform and can be used not only with JVM, but also with JS and Python.

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

Java libraries

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

— DeepLearning4J - a deep learning library for Java

— ND4J - an efficient matrix math library for JVM

— Dex - a Java-based data visualization tool

— Smile - a comprehensive machine learning, natural language processing, linear algebra, graph, interpolation, and visualization system. Besides Java API, Smile also provides a functional Kotlin API along with Scala and Clojure API.

  — Smile-NLP-kt - a Kotlin rewrite of the Scala implicits for the natural language processing part of Smile in the format of extension functions and interfaces.

— Apache Commons Math - a general math, statistics, and machine learning library for Java

— OptaPlanner - a solver utility for optimization planning problems

— Charts - a scientific JavaFX charting library in development

— CoreNLP - a natural language processing toolkit

— Apache Mahout - a distributed framework for regression, clustering and recommendation

— Weka - a collection of machine learning algorithms for data mining tasks

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

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 in Alpha. Language features and tooling may change in future Kotlin versions.

Support for multiplatform programming is one of Kotlin's key benefits. It reduces time spent writing and maintaining the same code for different platforms while retaining the flexibility and benefits of native programming.

This is how Kotlin Multiplatform works.

— **Common Kotlin** includes the language, core libraries, and basic tools. Code written in common Kotlin works everywhere on all platforms.

— With Kotlin Multiplatform libraries, you can reuse the multiplatform logic in common and platform-specific code. Common code can rely on a set of libraries that cover everyday tasks such as HTTP, serialization, and managing coroutines.

— To interop with platforms, use platform-specific versions of Kotlin. **Platform-specific versions of Kotlin** (Kotlin/JVM, Kotlin/JS, Kotlin/Native) include extensions to the Kotlin language, and platform-specific libraries and tools.

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

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

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

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

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

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

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

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

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

Use cases

Android — iOS

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

See KMM features, case studies and examples

Client — Server

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

The Ktor framework is suitable for building asynchronous servers and clients in connected systems.

What's next?

New to Kotlin? Visit Getting started with Kotlin.

Documentation

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

Tutorials

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

Sample projects

- Kotlin Multiplatform Mobile samples
- KotlinConf app
- KotlinConf Spinner app

What's New

What's New in Kotlin 1.4.0

In Kotlin 1.4.0, we ship a number of improvements in all of its components, with the focus on quality and performance. Below you will find the list of the most important changes in Kotlin 1.4.0.

Language features and improvements

- SAM conversions for Kotlin interfaces
- Explicit API mode for library authors
- Mixing named and positional arguments
- Trailing comma
- Callable reference improvements
- `break` and `continue` inside `when` included in loops

New tools in the IDE

- New flexible Project Wizard
- Coroutine Debugger

New compiler

- New, more powerful type inference algorithm
- New JVM and JS IR backends

Kotlin/JVM

- New JVM IR backend
- New modes for generating default methods in interfaces
- Unified exception type for null checks
- Type annotations in the JVM bytecode

Kotlin/JS

- New Gradle DSL
- New JS IR backend

Kotlin/Native

- Support for suspending functions in Swift and Objective-C
- Objective-C generics support by default
- Exception handling in Objective-C/Swift interop
- Generate release `.dSYM`s on Apple targets by default

Language features and improvements

Kotlin 1.4.0 comes with a variety of different language features and improvements. They include:

— `break` and `continue` inside `when` included in loops

— Explicit API mode for library authors

SAM conversions for Kotlin interfaces

Before Kotlin 1.4.0, you could apply SAM (Single Abstract Method) conversions only when working with Java methods and Java interfaces from Kotlin. From now on, you can use SAM conversions for Kotlin interfaces as well. To do so, mark a Kotlin interface explicitly as functional with the `fun` modifier.

SAM conversion applies if you pass a lambda as an argument when an interface with only one single abstract method is expected as a parameter. In this case, the compiler automatically converts the lambda to an instance of the class that implements the abstract member function.

```
fun interface IntPredicate {
    fun accept(i: Int): Boolean
}

val isEven = IntPredicate { it % 2 == 0 }

fun main() {
    println("Is 7 even? - ${isEven.accept(7)}")
}
```

Learn more about Kotlin functional interfaces and SAM conversions.

Explicit API mode for library authors

Kotlin compiler offers *explicit API mode* for library authors. In this mode, the compiler performs additional checks that help make the library's API clearer and more consistent. It adds the following requirements for declarations exposed to the library's public API:

— Visibility modifiers are required for declarations if the default visibility exposes them to the public API. This helps ensure that no declarations are exposed to the public API unintentionally.

— Explicit type specifications are required for properties and functions that are exposed to the public API. This guarantees that API users are aware of the types of API members they use.

Depending on your configuration, these explicit APIs can produce errors (*strict* mode) or warnings (*warning* mode). Certain kinds of declarations are excluded from such checks for the sake of readability and common sense:

— primary constructors

— properties of data classes

— property getters and setters

— `override` methods

Explicit API mode analyzes only the production sources of a module.

To compile your module in the explicit API mode, add the following lines to your Gradle build script:

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

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

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

    // for warning mode
    explicitApiWarning()
    // or
    explicitApi = ExplicitApiMode.Warning
}
```

When using the command-line compiler, switch to explicit API mode by adding the `-Xexplicit-api` compiler option with the value `strict` or `warning`.

```
-Xexplicit-api={strict|warning}
```

For more details about the explicit API mode, see the KEEP.

Mixing named and positional arguments

In Kotlin 1.3, when you called a function with named arguments, you had to place all the arguments without names (positional arguments) before the first named argument. For example, you could call `f(1, y = 2)`, but you couldn't call `f(x = 1, 2)`.

It was really annoying when all the arguments were in their correct positions but you wanted to specify a name for one argument in the middle. It was especially helpful for making absolutely clear which attribute a boolean or `null` value belongs to.

In Kotlin 1.4, there is no such limitation – you can now specify a name for an argument in the middle of a set of positional arguments. Moreover, you can mix positional and named arguments any way you like, as long as they remain in the correct order.

```
fun reformat(
    str: String,
    uppercaseFirstLetter: Boolean = true,
    wordSeparator: Char = ' '
) {
    // ...
}

//Function call with a named argument in the middle
reformat("This is a String!", uppercaseFirstLetter = false , '-')
```

Trailing comma

With Kotlin 1.4 you can now add a trailing comma in enumerations such as argument and parameter lists, `when` entries, and components of destructuring declarations. With a trailing comma, you can add new items and change their order without adding or removing commas.

This is especially helpful if you use multi-line syntax for parameters or values. After adding a trailing comma, you can then easily swap lines with parameters or values.

```
fun reformat(
    str: String,
    uppercaseFirstLetter: Boolean = true,
    wordSeparator: Character = ' ', //trailing comma
) {
    // ...
}
```

```
val colors = listOf(
    "red",
    "green",
    "blue", //trailing comma
)
```

Callable reference improvements

Kotlin 1.4 supports more cases for using callable references:

— References to functions with default argument values

— Function references in `Unit`-returning functions

— References that adapt based on the number of arguments in a function

— Suspend conversion on callable references

References to functions with default argument values

Now you can use callable references to functions with default argument values. If the callable reference to the function `foo` takes no arguments, the default value `0` is used.

```
fun foo(i: Int = 0): String = "$i!"

fun apply(func: () -> String): String = func()

fun main() {
    println(apply(::foo))
}
```

Previously, you had to write additional overloads for the function `apply` to use the default argument values.

```
// some new overload
fun applyInt(func: (Int) -> String): String = func(0)
```

Function references in `Unit`-returning functions

In Kotlin 1.4, you can use callable references to functions returning any type in `Unit`-returning functions. Before Kotlin 1.4, you could only use lambda arguments in this case. Now you can use both lambda arguments and callable references.

```
fun foo(f: () -> Unit) { }
fun returnsInt(): Int = 42

fun main() {
    foo { returnsInt() } // this was the only way to do it  before 1.4
    foo(::returnsInt) // starting from 1.4, this also works
}
```

References that adapt based on the number of arguments in a function

Now you can adapt callable references to functions when passing a variable number of arguments (`vararg`). You can pass any number of parameters of the same type at the end of the list of passed arguments.

```
fun foo(x: Int, vararg y: String) {}

fun use0(f: (Int) -> Unit) {}
fun use1(f: (Int, String) -> Unit) {}
fun use2(f: (Int, String, String) -> Unit) {}

fun test() {
    use0(::foo)
    use1(::foo)
    use2(::foo)
}
```

Suspend conversion on callable references

In addition to suspend conversion on lambdas, Kotlin now supports suspend conversion on callable references starting from version 1.4.0.

```
fun call() {}
fun takeSuspend(f: suspend () -> Unit) {}

fun test() {
    takeSuspend { call() } // OK before 1.4
    takeSuspend(::call) // In Kotlin 1.4, it also works
}
```

Using `break` and `continue` inside `when` expressions included in loops

In Kotlin 1.3, you could not use unqualified `break` and `continue` inside `when` expressions included in loops. The reason was that these keywords were reserved for possible fall-through behavior in `when` expressions.

That's why if you wanted to use `break` and `continue` inside `when` expressions in loops, you had to label them, which became rather cumbersome.

```
fun test(xs: List<Int>) {
    LOOP@for (x in xs) {
        when (x) {
            2 -> continue@LOOP
            17 -> break@LOOP
            else -> println(x)
        }
    }
}
```

In Kotlin 1.4, you can use `break` and `continue` without labels inside `when` expressions included in loops. They behave as expected by terminating the nearest enclosing loop or proceeding to its next step.

```
fun test(xs: List<Int>) {
    for (x in xs) {
        when (x) {
            2 -> continue
            17 -> break
            else -> println(x)
        }
    }
}
```

The fall-through behavior inside `when` is subject to further design.

Back to top

New tools in the IDE

With Kotlin 1.4, you can use the new tools in IntelliJ IDEA to simplify Kotlin development:

— New flexible Project Wizard

— Coroutine Debugger

New flexible Project Wizard

With the flexible new Kotlin Project Wizard, you have a place to easily create and configure different types of Kotlin projects, including multiplatform projects, which can be difficult to configure without a UI.

The new Kotlin Project Wizard is both simple and flexible:

1. *Select the project template*, depending on what you're trying to do. More templates will be added in the future.
2. *Select the build system* – Gradle (Kotlin or Groovy DSL), Maven, or IntelliJ IDEA.
   The Kotlin Project Wizard will only show the build systems supported on the selected project template.
3. *Preview the project structure* directly on the main screen.

Then you can finish creating your project or, optionally, *configure the project* on the next screen:

4. *Add/remove modules and targets* supported for this project template.
5. *Configure module and target settings*, for example, the target JVM version, target template, and test framework.

In the future, we are going to make the Kotlin Project Wizard even more flexible by adding more configuration options and templates.

You can try out the new Kotlin Project Wizard by working through these tutorials:

— Create a console application based on Kotlin/JVM

— Create a Kotlin/JS application for React

— Create a Kotlin/Native application

Coroutine Debugger

Many people already use coroutines for asynchronous programming. But when it came to debugging, working with coroutines before Kotlin 1.4, could be a real pain. Since coroutines jumped between threads, it was difficult to understand what a specific coroutine was doing and check its context. In some cases, tracking steps over breakpoints simply didn't work. As a result, you had to rely on logging or mental effort to debug code that used coroutines.

In Kotlin 1.4, debugging coroutines is now much more convenient with the new functionality shipped with the Kotlin plugin.

> Debugging works for versions 1.3.8 or later of `kotlinx-coroutines-core`.

The **Debug Tool Window** now contains a new **Coroutines** tab. In this tab, you can find information about both currently running and suspended coroutines. The coroutines are grouped by the dispatcher they are running on.

Now you can:

— Easily check the state of each coroutine.
— See the values of local and captured variables for both running and suspended coroutines.
— See a full coroutine creation stack, as well as a call stack inside the coroutine. The stack includes all frames with variable values, even those that would be lost during standard debugging.

If you need a full report containing the state of each coroutine and its stack, right-click inside the **Coroutines** tab, and then click **Get Coroutines Dump**. Currently, the coroutines dump is rather simple, but we're going to make it more readable and helpful in future versions of Kotlin.

Learn more about debugging coroutines in this blog post and IntelliJ IDEA documentation.

Back to top

New compiler

The new Kotlin compiler is going to be really fast; it will unify all the supported platforms and provide an API for compiler extensions. It's a long-term project, and we've already completed several steps in Kotlin 1.4.0:

— New, more powerful type inference algorithm is enabled by default.
— New JVM and JS IR backends are now in Alpha. They will become the default once we stabilize them.

New more powerful type inference algorithm

Kotlin 1.4 uses a new, more powerful type inference algorithm. This new algorithm was already available to try in Kotlin 1.3 by specifying a compiler option, and now it's used by default. You can find the full list of issues fixed in the new algorithm in YouTrack. Here you can find some of the most noticeable improvements:

— More cases where type is inferred automatically
— Smart casts for a lambda's last expression
— Smart casts for callable references
— Better inference for delegated properties

— SAM conversion for Java interfaces with different arguments

— Java SAM interfaces in Kotlin

More cases where type is inferred automatically

The new inference algorithm infers types for many cases where the old algorithm required you to specify them explicitly. For instance, in the following example the type of the lambda parameter `it` is correctly inferred to `String?`:

```
val rulesMap: Map<String, (String?) -> Boolean> = mapOf(
    "weak" to { it != null },
    "medium" to { !it.isNullOrBlank() },
    "strong" to { it != null && "^[a-zA-Z0-9]+$".toRegex().matches(it) }
)
```

In Kotlin 1.3, you needed to introduce an explicit lambda parameter or replace `to` with a `Pair` constructor with explicit generic arguments to make it work.

Smart casts for a lambda's last expression

In Kotlin 1.3, the last expression inside a lambda wasn't smart cast unless you specified the expected type. Thus, in the following example, Kotlin 1.3 infers `String?` as the type of the `result` variable:

```
val result = run {
    var str = currentValue()
    if (str == null) {
        str = "test"
    }
    str // the Kotlin compiler knows that str is not null here
}
// The type of 'result' is String? in Kotlin 1.3 and String in Kotlin 1.4
```

In Kotlin 1.4, thanks to the new inference algorithm, the last expression inside a lambda gets smart cast, and this new, more precise type is used to infer the resulting lambda type. Thus, the type of the `result` variable becomes `String`.

In Kotlin 1.3, you often needed to add explicit casts (either `!!` or type casts like `as String`) to make such cases work, and now these casts have become unnecessary.

Smart casts for callable references

In Kotlin 1.3, you couldn't access a member reference of a smart cast type. Now in Kotlin 1.4 you can:

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

You can use different member references `animal::meow` and `animal::woof` after the animal variable has been smart cast to specific types `Cat` and `Dog`. After type checks, you can access member references corresponding to subtypes.

Better inference for delegated properties

The type of a delegated property wasn't taken into account while analyzing the delegate expression which follows the `by` keyword. For instance, the following code didn't compile before, but now the compiler correctly infers the types of the `old` and `new` parameters as `String?`:

```
import kotlin.properties.Delegates

fun main() {
    var prop: String? by Delegates.observable(null) { p, old, new ->
        println("$old → $new")
    }
    prop = "abc"
    prop = "xyz"
}
```

SAM conversion for Java interfaces with different arguments

Kotlin has supported SAM conversions for Java interfaces from the beginning, but there was one case that wasn't supported, which was sometimes annoying when working with existing Java libraries. If you called a Java method that took two SAM interfaces as parameters, both arguments needed to be either lambdas or regular objects. You couldn't pass one argument as a lambda and another as an object.

The new algorithm fixes this issue, and you can pass a lambda instead of a SAM interface in any case, which is the way you'd naturally expect it to work.

```
// FILE: A.java
public class A {
    public static void foo(Runnable r1, Runnable r2) {}
}
```

```
// FILE: test.kt
fun test(r1: Runnable) {
    A.foo(r1) {}  // Works in Kotlin 1.4
}
```

Java SAM interfaces in Kotlin

In Kotlin 1.4, you can use Java SAM interfaces in Kotlin and apply SAM conversions to them.

```
import java.lang.Runnable

fun foo(r: Runnable) {}

fun test() {
    foo { } // OK
}
```

In Kotlin 1.3, you would have had to declare the function `foo` above in Java code to perform a SAM conversion.

Unified backends and extensibility

In Kotlin, we have three backends that generate executables: Kotlin/JVM, Kotlin/JS, and Kotlin/Native. Kotlin/JVM and Kotlin/JS don't share much code since they were developed independently of each other. Kotlin/Native is based on a new infrastructure built around an intermediate representation (IR) for Kotlin code.

We are now migrating Kotlin/JVM and Kotlin/JS to the same IR. As a result, all three backends share a lot of logic and have a unified pipeline. This allows us to implement most features, optimizations, and bug fixes only once for all platforms. Both new IR-based back-ends are in Alpha.

A common backend infrastructure also opens the door for multiplatform compiler extensions. You will be able to plug into the pipeline and add custom processing and transformations that will automatically work for all platforms.

We encourage you to use our new JVM IR and JS IR backends, which are currently in Alpha, and share your feedback with us.

Back to top

Kotlin/JVM

Kotlin 1.4.0 includes a number of JVM-specific improvements, such as:

— New JVM IR backend
— New modes for generating default methods in interfaces
— Unified exception type for null checks
— Type annotations in the JVM bytecode

New JVM IR backend

Along with Kotlin/JS, we are migrating Kotlin/JVM to the unified IR backend, which allows us to implement most features and bug fixes once for all platforms. You will also be able to benefit from this by creating multiplatform extensions that will work for all platforms.

Kotlin 1.4.0 does not provide a public API for such extensions yet, but we are working closely with our partners, including Jetpack Compose, who are already building their compiler plugins using our new backend.

We encourage you to try out the new Kotlin/JVM backend, which is currently in Alpha, and to file any issues and feature requests to our issue tracker. This will help us to unify the compiler pipelines and bring compiler extensions like Jetpack Compose to the Kotlin community more quickly.

To enable the new JVM IR backend, specify an additional compiler option in your Gradle build script:

```
kotlinOptions.useIR = true
```

> If you enable Jetpack Compose, you will automatically be opted in to the new JVM backend without needing to specify the compiler option in `kotlinOptions`.

When using the command-line compiler, add the compiler option `-Xuse-ir`.

> You can use code compiled by the new JVM IR backend only if you've enabled the new backend. Otherwise, you will get an error. Considering this, we don't recommend that library authors switch to the new backend in production.

New modes for generating default methods

When compiling Kotlin code to targets JVM 1.8 and above, you could compile non-abstract methods of Kotlin interfaces into Java's `default` methods. For this purpose, there was a mechanism that includes the `@JvmDefault` annotation for marking such methods and the `-Xjvm-default` compiler option that enables processing of this annotation.

In 1.4.0, we've added a new mode for generating default methods: `-Xjvm-default=all` compiles *all* non-abstract methods of Kotlin interfaces to `default` Java methods. For compatibility with the code that uses the interfaces compiled without `default`, we also added `all-compatibility` mode.

For more information about default methods in the Java interop, see the documentation and this blog post.

Unified exception type for null checks

Starting from Kotlin 1.4.0, all runtime null checks will throw a `java.lang.NullPointerException` instead of `KotlinNullPointerException`, `IllegalStateException`, `IllegalArgumentException`, and `TypeCastException`. This applies to: the `!!` operator, parameter null checks in the method preamble, platform-typed expression null checks, and the `as` operator with a non-null type. This doesn't apply to `lateinit` null checks and explicit library function calls like `checkNotNull` or `requireNotNull`.

This change increases the number of possible null check optimizations that can be performed either by the Kotlin compiler or by various kinds of bytecode processing tools, such as the Android R8 optimizer.

Note that from a developer's perspective, things won't change that much: the Kotlin code will throw exceptions with the same error messages as before. The type of exception changes, but the information passed stays the same.

Type annotations in the JVM bytecode

Kotlin can now generate type annotations in the JVM bytecode (target version 1.8+), so that they become available in Java reflection at runtime. To emit the type annotation in the bytecode, follow these steps:

1. Make sure that your declared annotation has a proper annotation target (Java's `ElementType.TYPE_USE` or Kotlin's `AnnotationTarget.TYPE`) and retention (`AnnotationRetention.RUNTIME`).
2. Compile the annotation class declaration to JVM bytecode target version 1.8+. You can specify it with `-jvm-target=1.8` compiler option.
3. Compile the code that uses the annotation to JVM bytecode target version 1.8+ (`-jvm-target=1.8`) and add the `-Xemit-jvm-type-annotations` compiler option.

Note that the type annotations from the standard library aren't emitted in the bytecode for now because the standard library is compiled with the target version 1.6.

So far, only the basic cases are supported:

— Type annotations on method parameters, method return types and property types;

— Invariant projections of type arguments, such as `Smth<@Ann Foo>`, `Array<@Ann Foo>`.

In the following example, the `@Foo` annotation on the `String` type can be emitted to the bytecode and then used by the library code:

```
@Target(AnnotationTarget.TYPE)
annotation class Foo

class A {
    fun foo(): @Foo String = "OK"
}
```

Back to top

Kotlin/JS

On the JS platform, Kotlin 1.4.0 provides the following improvements:

— New Gradle DSL

— New JS IR backend

New Gradle DSL

The `kotlin.js` Gradle plugin comes with an adjusted Gradle DSL, which provides a number of new configuration options and is more closely aligned to the DSL used by the `kotlin-multiplatform` plugin. Some of the most impactful changes include:

— Explicit toggles for the creation of executable files via `binaries.executable()`. Read more about the executing Kotlin/JS and its environment here.

— Configuration of webpack's CSS and style loaders from within the Gradle configuration via `cssSupport`. Read more about using them here.

— Improved management for npm dependencies, with mandatory version numbers or semver version ranges, as well as support for *development*, *peer*, and *optional* npm dependencies using `devNpm`, `optionalNpm` and `peerNpm`. Read more about dependency management for npm packages directly from Gradle here.

— Stronger integrations for Dukat, the generator for Kotlin external declarations. External declarations can now be generated at build time, or can be manually generated via a Gradle task. Read more about how to use the integration here.

New JS IR backend

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

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

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

For more detailed information about how to configure the Kotlin/JS IR compiler backend, check out the documentation.

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

Learn more about the available features in the Kotlin/JS IR compiler backend in the documentation.

Back to top

Kotlin/Native

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

— Support for suspending functions in Swift and Objective-C
— Objective-C generics support by default
— Exception handling in Objective-C/Swift interop
— Generate release `.dSYM`s on Apple targets by default
— Performance improvements
— Simplified management of CocoaPods dependencies

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

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

For example, if you write this Kotlin function:

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

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

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

For more information about using suspending functions in Swift and Objective-C, see the documentation.

Objective-C generics support by default

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

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

Please note that all specifics and limitations listed in the documentation are still valid.

Exception handling in Objective-C/Swift interop

In 1.4.0, we slightly change the Swift API generated from Kotlin with respect to the way exceptions are translated. There is a fundamental difference in error handling between Kotlin and Swift. All Kotlin exceptions are unchecked, while Swift has only checked errors. Thus, to make Swift code aware of expected exceptions, Kotlin functions should be marked with a `@Throws` annotation specifying a list of potential exception classes.

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

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

Generate release `.dSYM`s on Apple targets by default

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

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

For more information about crash report symbolication, see the documentation.

Performance improvements

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

— To improve the speed of object allocation, we now offer the mimalloc memory allocator as an alternative to the system allocator. mimalloc works up to two times faster on some benchmarks. Currently, the usage of mimalloc in Kotlin/Native is experimental; you can switch to it using the `-Xallocator=mimalloc` compiler option.

— We've reworked how C interop libraries are built. With the new tooling, Kotlin/Native produces interop libraries up to 4 times as fast as before, and artifacts are 25% to 30% the size they used to be.

— Overall runtime performance has improved because of optimizations in GC. This improvement will be especially apparent in projects with a large number of long-lived objects. `HashMap` and `HashSet` collections now work faster by escaping redundant boxing.

— In 1.3.70 we introduced two new features for improving the performance of Kotlin/Native compilation: caching project dependencies and running the compiler from the Gradle daemon. Since that time, we've managed to fix numerous issues and improve the overall stability of these features.

Simplified management of CocoaPods dependencies

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

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

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

Now you can also work with Pod libraries stored locally.

Depending on your needs, you can add dependencies between:

— A Kotlin project and Pod libraries stored remotely in the CocoaPods repository or stored locally on your machine.

— A Kotlin Pod (Kotlin project used as a CocoaPods dependency) and an Xcode project with one or more targets.

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

Learn how to add dependencies.

Back to top

Kotlin Multiplatform

> Multiplatform projects are in Alpha. Language features and tooling may change in future Kotlin versions.

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

— Sharing code in several targets with the hierarchical project structure

— Leveraging native libs in the hierarchical structure

— Specifying kotlinx dependencies only once

> Multiplatform projects require Gradle 6.0 or later.

Sharing code in several targets with the hierarchical project structure

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

Previously, any code added to a multiplatform project could be placed either in a platform-specific source set, which is limited to one target and can't be reused by any other platform, or in a common source set, like `commonMain` or `commonTest`, which is shared across all the platforms in the project. In the common source set, you could only call a platform-specific API by using an `expect` declaration that needs platform-specific `actual` implementations.

This made it easy to share code on all platforms, but it was not so easy to share between only some of the targets, especially similar ones that could potentially reuse a lot of the common logic and third-party APIs.

For example, in a typical multiplatform project targeting iOS, there are two iOS-related targets: one for iOS ARM64 devices, and the other for the x64 simulator. They have separate platform-specific source sets, but in practice, there is rarely a need for different code for the device and simulator, and their dependencies are much alike. So iOS-specific code could be shared between them.

Apparently, in this setup, it would be desirable to have a *shared source set for two iOS targets*, with Kotlin/Native code that could still directly call any of the APIs that are common to both the iOS device and the simulator.

Now you can do this with the hierarchical project structure support, which infers and adapts the API and language features available in each source set based on which targets consume them.

For common combinations of targets, you can create a hierarchical structure with target shortcuts.

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

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

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

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

```
kotlin{
    sourceSets {
        val desktopMain by creating {
            dependsOn(commonMain)
        }
        val linuxX64Main by getting {
            dependsOn(desktopMain)
        }
        val mingwX64Main by getting {
            dependsOn(desktopMain)
        }
        val macosX64Main by getting {
            dependsOn(desktopMain)
        }
    }
}
```

Thanks to the hierarchical project structure, libraries can also provide common APIs for a subset of targets. Learn more about sharing code in libraries.

Leveraging native libs in the hierarchical structure

You can use platform-dependent libraries, such as `Foundation`, `UIKit`, and `posix`, in source sets shared among several native targets. This can help you share more native code without being limited by platform-specific dependencies.

No additional steps are required – everything is done automatically. IntelliJ IDEA will help you detect common declarations that you can use in the shared code.

Learn more about usage of platform-dependent libraries.

Specifying dependencies only once

From now on, instead of specifying dependencies on different variants of the same library in shared and platform-specific source sets where it is used, you should specify a dependency only once in the shared source set.

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.9'
            }
        }
    }
}
```

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.9")
            }
        }
    }
}
```

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

However, the change doesn't currently affect:

— The `stdlib` library – starting from Kotlin 1.4.0, the `stdlib` dependency is added automatically.

— The `kotlin.test` library – you should still use `test-common` and `test-annotations-common`. These dependencies will be addressed later.

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

Learn more about configuring dependencies.

Back to top

Gradle project improvements

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

— Dependency on the standard library is now added by default

— Kotlin projects require a recent version of Gradle

— Improved support for Kotlin Gradle DSL in the IDE

Dependency on the standard library added by default

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

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

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

Learn how to change the default behavior.

Minimum Gradle version for Kotlin projects

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

Improved *.gradle.kts support in the IDE

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

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

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

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

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

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

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

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

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

Back to top

Standard library

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

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

Common exception processing API

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

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

New functions for arrays and collections

Collections

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

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

```
val set = setOfNotNull(null, 1, 2, 0, null)
println(set)
```

— `shuffled()` for sequences.

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

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

```
  listOf("a", "b", "c", "d").onEachIndexed {
      index, item -> println(index.toString() + ":" + item)
  }

 val list = listOf("hello", "kot", "lin", "world")
        val kotlin = list.flatMapIndexed { index, item ->
            if (index in 1..2) item.toList() else emptyList()
        }
```

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

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

— `runningFold()`, its synonym `scan()`, and `runningReduce()` apply the given operation to the collection elements sequentially, similarly to `fold()` and `reduce()`; the difference is that these new functions return the whole sequence of intermediate results.

```
  val numbers = mutableListOf(0, 1, 2, 3, 4, 5)
  val runningReduceSum = numbers.runningReduce { sum, item -> sum + item }
  val runningFoldSum = numbers.runningFold(10) { sum, item -> sum + item }
```

— `sumOf()` takes a selector function and returns a sum of its values for all elements of a collection. `sumOf()` can produce sums of the types `Int`, `Long`, `Double`, `UInt`, and `ULong`. On the JVM, `BigInteger` and `BigDecimal` are also available.

```
  val order = listOf<OrderItem>(
      OrderItem("Cake", price = 10.0, count = 1),
      OrderItem("Coffee", price = 2.5, count = 3),
      OrderItem("Tea", price = 1.5, count = 2))

  val total = order.sumOf { it.price * it.count} // Double
  val count = order.sumOf { it.count } // Int
```

— The `min()` and `max()` functions have been renamed to `minOrNull()` and `maxOrNull()` to comply with the naming convention used across the Kotlin collections API. An `*OrNull` suffix in the function name means that it returns `null` if the receiver collection is empty. The same applies to `minBy()`, `maxBy()`, `minWith()`, `maxWith()` – in 1.4, they have `*OrNull()` synonyms.

— The new `minOf()` and `maxOf()` extension functions return the minimum and the maximum value of the given selector function on the collection items.

```
  val order = listOf<OrderItem>(
      OrderItem("Cake", price = 10.0, count = 1),
      OrderItem("Coffee", price = 2.5, count = 3),
      OrderItem("Tea", price = 1.5, count = 2))
  val highestPrice = order.maxOf { it.price }
```

There are also `minOfWith()` and `maxOfWith()`, which take a `Comparator` as an argument, and `*OrNull()` versions of all four functions that return `null` on empty collections.

— New overloads for `flatMap` and `flatMapTo` let you use transformations with return types that don't match the receiver type, namely:

— Transformations to `Sequence` on `Iterable`, `Array`, and `Map`

— Transformations to `Iterable` on `Sequence`

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

— `removeFirst()` and `removeLast()` shortcuts for removing elements from mutable lists, and `*orNull()` counterparts of these functions.

Arrays

To provide a consistent experience when working with different container types, we've also added new functions for **arrays**:

— `shuffle()` puts the array elements in a random order.

— `onEach()` performs the given action on each array element and returns the array itself.

— `associateWith()` and `associateWithTo()` build maps with the array elements as keys.

— `reverse()` for array subranges reverses the order of the elements in the subrange.

— `sortDescending()` for array subranges sorts the elements in the subrange in descending order.

— `sort()` and `sortWith()` for array subranges are now available in the common library.

```
var language = ""
val letters = arrayOf("k", "o", "t", "l", "i", "n")
val fileExt = letters.onEach { language += it }
   .filterNot { it in "aeuio" }.take(2)
   .joinToString(prefix = ".", separator = "")
println(language) // "kotlin"
println(fileExt) // ".kt"

letters.shuffle()
letters.reverse(0, 3)
letters.sortDescending(2, 5)
println(letters.contentToString()) // [k, o, t, l, i, n]
```

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

— `ByteArray.decodeToString()` and `String.encodeToByteArray()`

— `CharArray.concatToString()` and `String.toCharArray()`

```
 str = "kotlin"
val array = str.toCharArray()
println(array.concatToString())
```

ArrayDeque

We've also added the `ArrayDeque` class – an implementation of a double-ended queue. Double-ended queue lets you can add or remove elements both at the beginning and the end of the queue in an amortized constant time. You can use a double-ended queue by default when you need a queue or a stack in your code.

```
fun main() {
    val deque = ArrayDeque(listOf(1, 2, 3))

    deque.addFirst(0)
    deque.addLast(4)
    println(deque) // [0, 1, 2, 3, 4]

    println(deque.first()) // 0
    println(deque.last()) // 4

    deque.removeFirst()
    deque.removeLast()
    println(deque) // [1, 2, 3]
}
```

The `ArrayDeque` implementation uses a resizable array underneath: it stores the contents in a circular buffer, an `Array`, and resizes this `Array` only when it becomes full.

Functions for string manipulations

The standard library in 1.4.0 includes a number of improvements in the API for string manipulation:

— `StringBuilder` has useful new extension functions: `set()`, `setRange()`, `deleteAt()`, `deleteRange()`, `appendRange()`, and others.

```
val sb = StringBuilder("Bye Kotlin 1.3.72")
sb.deleteRange(0, 3)
sb.insertRange(0, "Hello", 0 ,5)
sb.set(15, '4')
sb.setRange(17, 19, "0")
print(sb.toString())
```

— Some existing functions of `StringBuilder` are available in the common library. Among them are `append()`, `insert()`, `substring()`, `setLength()`, and more.

— New functions `Appendable.appendLine()` and `StringBuilder.appendLine()` have been added to the common library. They replace the JVM-only `appendln()` functions of these classes.

```
println(buildString {
    appendLine("Hello,")
    appendLine("world")
})
```

Bit operations

New functions for bit manipulations:

— `countOneBits()`

— `countLeadingZeroBits()`

— `countTrailingZeroBits()`

— `takeHighestOneBit()`

— `takeLowestOneBit()`

— `rotateLeft()` and `rotateRight()` (experimental)

```
val number = "1010000".toInt(radix = 2)
println(number.countOneBits())
println(number.countTrailingZeroBits())
println(number.takeHighestOneBit().toString(2))
```

Delegated properties improvements

In 1.4.0, we have added new features to improve your experience with delegated properties in Kotlin:

— Now a property can be delegated to another property.

— A new interface `PropertyDelegateProvider` helps create delegate providers in a single declaration.

— `ReadWriteProperty` now extends `ReadOnlyProperty` so you can use both of them for read-only properties.

Aside from the new API, we've made some optimizations that reduce the resulting bytecode size. These optimizations are described in this blog post.

For more information about delegated properties, see the documentation.

Converting from KType to Java Type

A new extension property `KType.javaType` (currently experimental) in the stdlib helps you obtain a `java.lang.reflect.Type` from a Kotlin type without using the whole `kotlin-reflect` dependency.

```
import kotlin.reflect.javaType
import kotlin.reflect.typeOf

@OptIn(ExperimentalStdlibApi::class)
inline fun <reified T> accessReifiedTypeArg() {
   val kType = typeOf<T>()
   println("Kotlin type: $kType")
   println("Java type: ${kType.javaType}")
}

@OptIn(ExperimentalStdlibApi::class)
fun main() {
   accessReifiedTypeArg<String>()
   // Kotlin type: kotlin.String
   // Java type: class java.lang.String

   accessReifiedTypeArg<List<String>>()
   // Kotlin type: kotlin.collections.List<kotlin.String>
   // Java type: java.util.List<java.lang.String>
}
```

Proguard configurations for Kotlin reflection

Starting from 1.4.0, we have embedded Proguard/R8 configurations for Kotlin Reflection in `kotlin-reflect.jar`. With this in place, most Android projects using R8 or Proguard should work with kotlin-reflect without needing any additional configuration. You no longer need to copy-paste the Proguard rules for kotlin-reflect internals. But note that you still need to explicitly list all the APIs you're going to reflect on.

Improving the existing API

— Several functions now work on null receivers, for example:

  - `toBoolean()` on strings
  - `contentEquals()`, `contentHashcode()`, `contentToString()` on arrays
- `NaN`, `NEGATIVE_INFINITY`, and `POSITIVE_INFINITY` in `Double` and `Float` are now defined as `const`, so you can use them as annotation arguments.
- New constants `SIZE_BITS` and `SIZE_BYTES` in `Double` and `Float` contain the number of bits and bytes used to represent an instance of the type in binary form.
- The `maxOf()` and `minOf()` top-level functions can accept a variable number of arguments (`vararg`).

module-info descriptors for stdlib artifacts

Kotlin 1.4.0 adds `module-info.java` module information to default standard library artifacts. This lets you use them with **jlink** tool, which generates custom Java runtime images containing only the platform modules that are required for your app. You could already use jlink with Kotlin standard library artifacts, but you had to use separate artifacts to do so – the ones with the "modular" classifier – and the whole setup wasn't straightforward.
In Android, make sure you use the Android Gradle plugin version 3.2 or higher, which can correctly process jar files with module-info.

Deprecations

toShort() and toByte() of Double and Float

We've deprecated the functions `toShort()` and `toByte()` on `Double` and `Float` because they could lead to unexpected results because of the narrow value range and smaller variable size.

To convert floating-point numbers to `Byte` or `Short`, use the two-step conversion: first, convert them to `Int`, and then convert them again to the target type.

contains(), indexOf(), and lastIndexOf() on floating-point arrays

We've deprecated the `contains()`, `indexOf()`, and `lastIndexOf()` extension functions of `FloatArray` and `DoubleArray` because they use the IEEE 754 standard equality, which contradicts the total order equality in some corner cases. See this issue for details.

min() and max() collection functions

We've deprecated the `min()` and `max()` collection functions in favor of `minOrNull()` and `maxOrNull()`, which more properly reflect their behavior – returning `null` on empty collections. See this issue for details.

Exclusion of the deprecated experimental coroutines

The `kotlin.coroutines.experimental` API was deprecated in favor of kotlin.coroutines in 1.3.0. In 1.4.0, we're completing the deprecation cycle for `kotlin.coroutines.experimental` by removing it from the standard library. For those who still use it on the JVM, we've provided a compatibility artifact `kotlin-coroutines-experimental-compat.jar` with all the experimental coroutines APIs. We've published it to Maven, and we include it in the Kotlin distribution alongside the standard library.

Back to top

Stable JSON serialization

With Kotlin 1.4.0, we are shipping the first stable version of kotlinx.serialization

— 1.0.0-RC. Now we are pleased to declare the JSON serialization API in `kotlinx-serialization-core` (previously known as `kotlinx-serialization-runtime`) stable. Libraries for other serialization formats remain experimental, along with some advanced parts of the core library.

We have significantly reworked the API for JSON serialization to make it more consistent and easier to use. From now on, we'll continue developing the JSON serialization API in a backward-compatible manner. However, if you have used previous versions of it, you'll need to rewrite some of your code when migrating to 1.0.0-RC. To help you with this, we also offer the Kotlin Serialization Guide – the complete set of documentation for `kotlinx.serialization`. It will guide you through the process of using the most important features and it can help you address any issues that you might face.

> **Note**: `kotlinx-serialization` 1.0.0-RC only works with Kotlin compiler 1.4. Earlier compiler versions are not compatible.

Back to top

Scripting and REPL

In 1.4.0, scripting in Kotlin benefits from a number of functional and performance improvements along with other updates. Here are some of the key changes:

— New dependencies resolution API

— New REPL API

— Compiled scripts cache

— Artifacts renaming

To help you become more familiar with scripting in Kotlin, we've prepared a project with examples. It contains examples of the standard scripts (`*.main.kts`) and examples of uses of the Kotlin Scripting API and custom script definitions. Please give it a try and share your feedback using our issue tracker.

New dependencies resolution API

In 1.4.0, we've introduced a new API for resolving external dependencies (such as Maven artifacts), along with implementations for it. This API is published in the new artifacts `kotlin-scripting-dependencies` and `kotlin-scripting-dependencies-maven`. The previous dependency resolution functionality in `kotlin-script-util` library is now deprecated.

New REPL API

The new experimental REPL API is now a part of the Kotlin Scripting API. There are also several implementations of it in the published artifacts, and some have advanced functionality, such as code completion. We use this API in the Kotlin Jupyter kernel and now you can try it in your own custom shells and REPLs.

Compiled scripts cache

The Kotlin Scripting API now provides the ability to implement a compiled scripts cache, significantly speeding up subsequent executions of unchanged scripts. Our default advanced script implementation `kotlin-main-kts` already has its own cache.

Artifacts renaming

In order to avoid confusion about artifact names, we've renamed `kotlin-scripting-jsr223-embeddable` and `kotlin-scripting-jvm-host-embeddable` to just `kotlin-scripting-jsr223` and `kotlin-scripting-jvm-host`. These artifacts depend on the `kotlin-compiler-embeddable` artifact, which shades the bundled third-party libraries to avoid usage conflicts. With this renaming, we're making the usage of `kotlin-compiler-embeddable` (which is safer in general) the default for scripting artifacts. If, for some reason, you need artifacts that depend on the unshaded `kotlin-compiler`, use the artifact versions with the `-unshaded` suffix, such as `kotlin-scripting-jsr223-unshaded`. Note that this renaming affects only the scripting artifacts that are supposed to be used directly; names of other artifacts remain unchanged.

Back to top

Migrating to Kotlin 1.4.0

The Kotlin plugin's migration tools help you migrate your projects from earlier versions of Kotlin to 1.4.0.

Just change the Kotlin version to `1.4.0` and re-import your Gradle or Maven project. The IDE will then ask you about migration.

If you agree, it will run migration code inspections that will check your code and suggest corrections for anything that doesn't work or that is not recommended in 1.4.0.

Code inspections have different severity levels, to help you decide which suggestions to accept and which to ignore.

Migrating multiplatform projects

To help you start using the new features of Kotlin multiplatform in existing projects, we publish the migration guide for multiplatform projects.

Back to top

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:

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

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

It is equivalent to this Java code:

```
interface Foo {
    public static int answer = 42;
    public static void sayHello() {
        // ...
    }
}
```

Nested declarations in annotation classes

In Kotlin 1.3 it is possible for annotations to have nested classes, interfaces, objects, and companions:

```
annotation class Foo {
    enum class Direction { UP, DOWN, LEFT, RIGHT }

    annotation class Bar

    companion object {
        fun foo(): Int = 42
        val bar: Int = 42
    }
}
```

Parameterless `main`

By convention, the entry point of a Kotlin program is a function with a signature like `main(args: Array<String>)`, where `args` represent the command-line arguments passed to the program. However, not every application supports command-line arguments, so this parameter often ends up not being used.

Kotlin 1.3 introduced a simpler form of `main` which takes no parameters. Now "Hello, World" in Kotlin is 19 characters shorter!

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

Functions with big arity

In Kotlin, functional types are represented as generic classes taking a different number of parameters: `Function0<R>`, `Function1<P0, R>`, `Function2<P0, P1, R>`, ... This approach has a problem in that this list is finite, and it currently ends with `Function22`.

Kotlin 1.3 relaxes this limitation and adds support for functions with bigger arity:

```
fun trueEnterpriseComesToKotlin(block: (Any, Any, ... /* 42 more */, Any) -> Any) {
    block(Any(), Any(), ..., Any())
}
```

Progressive mode

Kotlin cares a lot about stability and backward compatibility of code: Kotlin compatibility policy says that "breaking changes" (e.g., a change which makes the code that used to compile fine, not compile anymore) can be introduced only in the major releases (1.2, 1.3, etc.).

We believe that a lot of users could use a much faster cycle, where critical compiler bug fixes arrive immediately, making the code more safe and correct. So, Kotlin 1.3 introduces *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 in Alpha. See details in the reference.

Kotlin 1.3 introduces a new kind of declaration — `inline class`. Inline classes can be viewed as a restricted version of the usual classes, in particular, inline classes must have exactly one property:

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

The Kotlin compiler will use this restriction to aggressively optimize runtime representation of inline classes and substitute their instances with the value of the underlying property where possible removing constructor calls, GC pressure, and enabling other optimizations:

```
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 in Beta. 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:

```
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:

```
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 in Kotlin 1.3.

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.

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:

```
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:

```
@CacheConfig(cacheNames = ["books", "default"])
public class BookRepositoryImpl {
    // ...
}
```

The array literal syntax is constrained to annotation arguments.

Lateinit top-level properties and local variables

The `lateinit` modifier can now be used on top-level properties and local variables. The latter can be used, for example, when a lambda passed as a constructor argument to one object refers to another object which has to be defined later:

```
class Node<T>(val value: T, val next: () -> Node<T>)

fun main(args: Array<String>) {
    // A cycle of three nodes:
    lateinit var third: Node<Int>

    val second = Node(2, next = { third })
    val first = Node(1, next = { second })

    third = Node(3, next = { first })

    val nodes = generateSequence(first) { it.next() }
    println("Values in the cycle: ${nodes.take(7).joinToString { it.value.toString() }}, ...")
}
```

Checking whether a lateinit var is initialized

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

```
    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:

```
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:

```
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:

```
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:

```
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 1.1

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 frontend development environment. See below for a more detailed list of changes.

Coroutines (experimental)

The key new feature in Kotlin 1.1 is *coroutines*, bringing the support of `async` / `await`, `yield` and similar programming patterns. The key feature of Kotlin's design is that the implementation of coroutine execution is part of the libraries, not the language, so you aren't bound to any specific programming paradigm or concurrency library.

A coroutine is effectively a light-weight thread that can be suspended and resumed later. Coroutines are supported through *suspending functions*: a call to such a function can potentially suspend a coroutine, and to start a new coroutine we usually use an anonymous suspending functions (i.e. suspending lambdas).

Let's look at `async` / `await` which is implemented in an external library, kotlinx.coroutines:

```
// runs the code in the background thread pool
fun asyncOverlay() = async(CommonPool) {
    // start two async operations
    val original = asyncLoadImage("original")
    val overlay = asyncLoadImage("overlay")
    // and then apply overlay to both results
    applyOverlay(original.await(), overlay.await())
}

// launches new coroutine in UI context
launch(UI) {
    // wait for async overlay to complete
    val image = asyncOverlay().await()
    // and then show it in UI
    showImage(image)
}
```

Here, `async { ... }` starts a coroutine and, when we use `await()`, the execution of the coroutine is suspended while the operation being awaited is executed, and is resumed (possibly on a different thread) when the operation being awaited completes.

The standard library uses coroutines to support *lazily generated sequences* with `yield` and `yieldAll` functions. In such a sequence, the block of code that returns sequence elements is suspended after each element has been retrieved, and resumed when the next element is requested. Here's an example:

```
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:

```
external class Node {
    val firstChild: Node

    fun appendChild(child: Node): Node

    fun removeChild(child: Node): Node

    // etc
}
```

Improved import handling

You can now describe declarations which should be imported from JavaScript modules more precisely. If you add the `@JsModule("<module-name>")` annotation on an external declaration it will be properly imported to a module system (either CommonJS or AMD) during the compilation. For example, with CommonJS the declaration will be imported via `require(...)` function. Additionally, if you want to import a declaration either as a module or as a global JavaScript object, you can use the `@JsNonModule` annotation.

For example, here's how you can import JQuery into a Kotlin module:

```
external interface JQuery {
    fun toggle(duration: Int = definedExternally): JQuery
    fun click(handler: (Event) -> Unit): JQuery
}

@JsModule("jquery")
@JsNonModule
@JsName("$")
external fun jquery(selector: String): JQuery
```

In this case, JQuery will be imported as a module named `jquery`. Alternatively, it can be used as a $-object, depending on what module system Kotlin compiler is configured to use.

You can use these declarations in your application like this:

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

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

```
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

```
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:

```
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:

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

Use a function returning nullable value:

```
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

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

or

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

See for loop.

`while` loop

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

See while loop.

when expression

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

See when expression.

Ranges

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

```
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:

```
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`)

```
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`)

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

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

Java 7's try with resources

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

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 **Kotlin style guide** from the menu.

To verify that your code is formatted according to the style guide, go to **Settings | Editor | Inspections** 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 abstract return type:

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

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

fun Foo(): 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 four spaces for indentation. Do not use tabs.

For curly braces, put the opening brace in the end of the line where the construct begins, and the closing brace on a separate line aligned horizontally with the opening construct.

```
if (elements != null) {
    for (element in elements) {
        // ...
    }
}
```

> In Kotlin, semicolons are optional, and therefore line breaks are significant. The language design assumes Java-style braces, and you may encounter surprising behavior if you try to use a different formatting style.

Horizontal whitespace

Put spaces around binary operators (`a + b`). Exception: don't put spaces around the "range to" operator (`0..i`).

Do not put spaces around unary operators (`a++`)

Put spaces between control flow keywords (`if`, `when`, `for` and `while`) and the corresponding opening parenthesis.

Do not put a space before an opening parenthesis in a primary constructor declaration, method declaration or method call.

```
class A(val x: Int)

fun foo(x: Int) { ... }

fun bar() {
    foo(1)
}
```

Never put a space after `(`, `[`, or before `]`, `)`.

Never put a space around `.` or `?.`: `foo.bar().filter { it > 2 }.joinToString()`, `foo?.bar()`

Put a space after `//`: `// This is a comment`

Do not put spaces around angle brackets used to specify type parameters: `class Map<K, V> { ... }`

Do not put spaces around `::`: `Foo::class`, `String::length`

Do not put a space before `?` used to mark a nullable type: `String?`

As a general rule, avoid horizontal alignment of any kind. Renaming an identifier to a name with a different length should not affect the formatting of either the declaration or any of the usages.

Colon

Put a space before `:` in the following cases:

— when it's used to separate a type and a supertype;

— when delegating to a superclass constructor or a different constructor of the same class;

— after the `object` keyword.

Don't put a space before `:` when it separates a declaration and its type.

Always put a space after `:`.

```
abstract class Foo<out T : Any> : IFoo {
    abstract fun foo(a: Int): T
}

class FooImpl : Foo() {
    constructor(x: String) : this(x) { /*...*/ }

    val x = object : IFoo { /*...*/ }
}
```

Class 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 (four 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 / fun // as a modifier in `fun interface`
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 four 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 four spaces relative to statement begin. Put the closing parentheses of the condition together with the opening curly brace on a separate line:

```
if (!component.isSyncing &&
    !hasAnyKotlinRuntimeInScope(module)
) {
    return createKotlinNotConfiguredPanel(module)
}
```

> 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)
}
```

Trailing commas

A trailing comma is a comma symbol after the last item of a series of elements:

```
class Person(
    val firstName: String,
    val lastName: String,
    val age: Int, // trailing comma
)
```

Using trailing commas has several benefits:

— It makes version-control diffs cleaner – as all the focus is on the changed value.

— It makes adding and reordering elements – there is no need to add or delete the comma if you manipulate elements.

— It simplifies code generation, for example, for object initializers. The last element can also have a comma.

Trailing commas are entirely optional – your code will still work without them. The Kotlin style guide encourages the use of trailing commas at the declaration site and leaves it optional for the call site.

To enable trailing commas in the IntelliJ IDEA formatter, go to **Settings | Editor | Code Style | Kotlin**, open the **Other** tab and select the **Use trailing comma** option.

Kotlin supports trailing commas in the following cases:

— Enumerations
— Value arguments
— Class properties and parameters
— Function value parameters
— Parameters with optional type (including setters)
— Indexing suffix
— Lambda parameters
— when entry
— Collection literals (in annotations)
— Type arguments
— Type parameters
— Destructuring declarations

Enumerations

```
enum class Direction {
    NORTH,
    SOUTH,
    WEST,
    EAST, // trailing comma
}
```

Value arguments

```
fun shift(x: Int, y: Int) { /*...*/ }

shift(
    25,
    20, // trailing comma
)

val colors = listOf(
    "red",
    "green",
    "blue", // trailing comma
)
```

Class properties and parameters

```
class Customer(
    val name: String,
    val lastName: String, // trailing comma
)

class Customer(
    val name: String,
    lastName: String, // trailing comma
)
```

Function value parameters

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

constructor(
    x: Comparable<Number>,
    y: Iterable<Number>, // trailing comma
) {}

fun print(
    vararg quantity: Int,
    description: String, // trailing comma
) {}
```

Parameters with optional type (including setters)

```
val sum: (Int, Int, Int) -> Int = fun(
    x,
    y,
    z, // trailing comma
): Int {
    return x + y + x
}
println(sum(8, 8, 8))
```

Indexing suffix

```
class Surface {
    operator fun get(x: Int, y: Int) = 2 * x + 4 * y - 10
}
fun getZValue(mySurface: Surface, xValue: Int, yValue: Int) =
    mySurface[
        xValue,
        yValue, // trailing comma
    ]
```

Lambda parameters

```
fun main() {
    val x = {
            x: Comparable<Number>,
            y: Iterable<Number>, // trailing comma
        ->
        println("1")
    }

    println(x)
}
```

when entry

```
fun isReferenceApplicable(myReference: KClass<*>) = when (myReference) {
    Comparable::class,
    Iterable::class,
    String::class, // trailing comma
        -> true
    else -> false
}
```

Collection literals (in annotations)

```
annotation class ApplicableFor(val services: Array<String>)

@ApplicableFor([
    "serializer",
    "balancer",
    "database",
    "inMemoryCache", // trailing comma
])
fun run() {}
```

Type arguments

```
fun <T1, T2> foo() {}

fun main() {
    foo<
            Comparable<Number>,
            Iterable<Number>, // trailing comma
            >()
}
```

Type parameters

```
class MyMap<
        MyKey,
        MyValue, // trailing comma
        > {}
```

Destructuring declarations

```
data class Car(val manufacturer: String, val model: String, val year: Int)
val myCar = Car("Tesla", "Y", 2019)

val (
    manufacturer,
    model,
    year, // trailing comma
) = myCar

val cars = listOf<Car>()
fun printMeanValue() {
    var meanValue: Int = 0
    for ((
        _,
        _,
        year, // trailing comma
    ) in cars) {
        meanValue += year
    }
    println(meanValue/cars.size)
}
printMeanValue()
```

Documentation comments

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

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

Short comments can be placed on a single line:

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

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

```
// Avoid doing this:

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

// Do this instead:

/**
 * Returns the absolute value of the given [number].
 */
fun abs(number: Int) { /*...*/ }
```

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)
```