Marcin Moskała

Kotlin Essentials

with exercises

Kotlin Essentials

Marcin Moskała

This book is available at http://leanpub.com/kotlin_developers

This version was published on 2024-12-16

ISBN 978-83-966847-0-7

Leanpub

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

For my friends, Agata and Michał Mazur

Contents

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Who is this book for?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

What will be covered?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

The Kotlin for Developers series

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

My story

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Conventions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Code conventions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercises and solutions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Suggestions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Acknowledgments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

What is Kotlin?

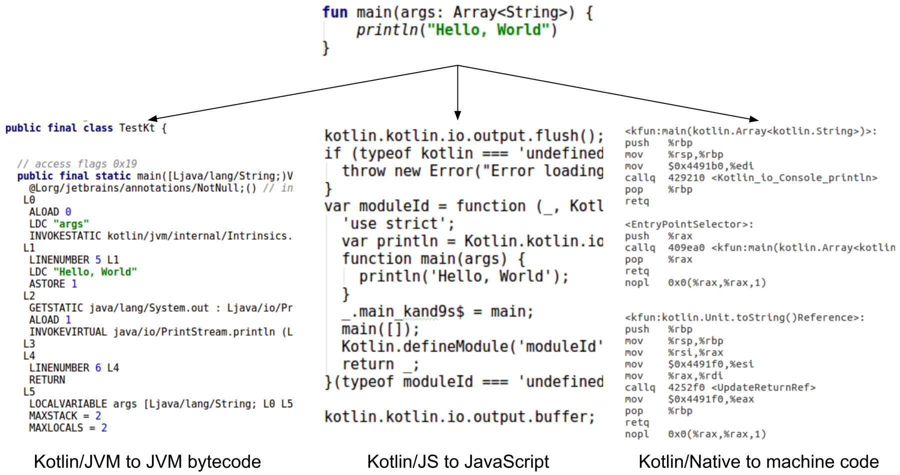
Kotlin is an open-source, multiplatform, multi-paradigm, statically typed, general-purpose programming language. But what does all this mean?

- Open-source means that the sources of the Kotlin compiler are freely available for modification and redistribution. Kotlin is primarily made by JetBrains, but now there is the Kotlin Foundation, which promotes and advances the development of this language. There is also a public process known as KEEP, which allows anyone to see and comment on official design change propositions.
- Multiplatform means that a language can be used on more than one platform. For instance, Kotlin can be used both on Android and iOS.
- Multi-paradigm means that a language has support for more than one programming paradigm. Kotlin has powerful support for both Object-Oriented Programming and Functional Programming.
- Statically typed means that each variable, object, and function has an associated type that is known at compile time.
- General-purpose means that a language is designed to be used for building software in a wide variety of application domains across a multitude of hardware configurations and operating systems.

These descriptions might not be clear now, but you will see them all in action throughout the book. Let's start by discussing Kotlin's multiplatform capabilities.

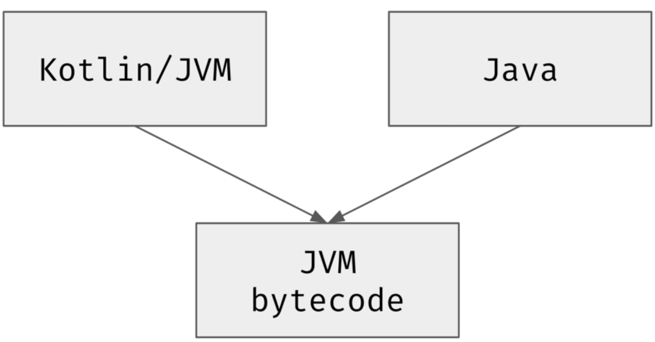
Kotlin platforms

Kotlin is a compiled programming language. This means that you can write some code in Kotlin and then use the Kotlin compiler to produce code in a lower-level language. Kotlin can currently be compiled into JVM bytecode (Kotlin/JVM), JavaScript (Kotlin/JS), or machine code (Kotlin/Native).

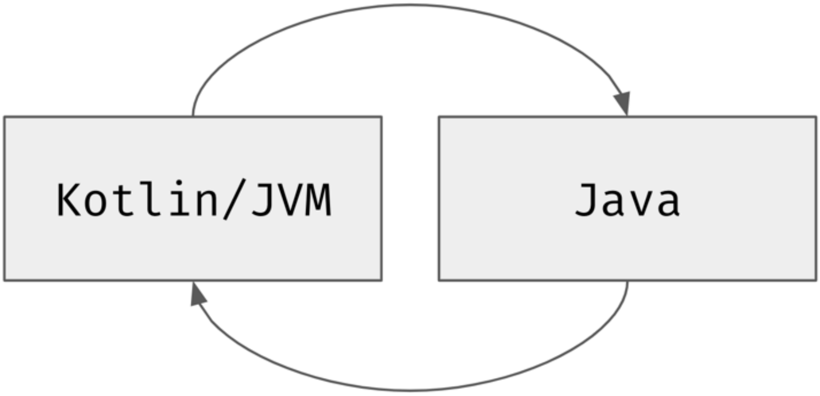


In this book, I would like to address all these compilation targets and, by default, show code that works on all of them, but I will concentrate on the most popular one: Kotlin/JVM.

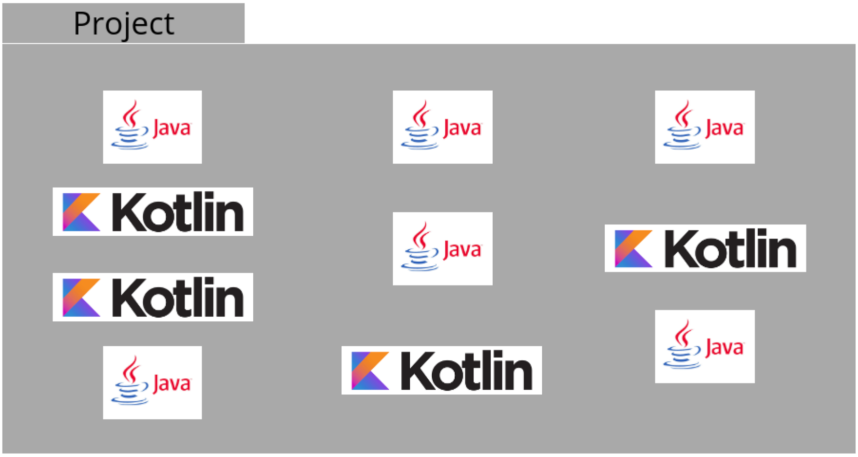
Kotlin/JVM is the technology that's used to compile Kotlin code into JVM bytecode. The result is nearly identical to the result of compiling Java code into JVM bytecode. We also use the term “Kotlin/JVM” to talk about code that will be compiled into JVM bytecode.



Kotlin/JVM and Java are fully interoperable. Any code written in Java can be used in Kotlin/JVM. Any Java library, including those based on annotation processing, can be used in Kotlin/JVM. Kotlin/JVM can use Java classes, modules, libraries, and the Java standard library. Any Kotlin/JVM code can be used in Java (except for suspending functions, which are a support for Kotlin Coroutines).



Kotlin and Java can be mixed in a single project. A typical scenario is that a project was initially developed in Java, but then its creators decided to start using Kotlin. To do this, instead of migrating the whole project, these developers decided to add Kotlin to it. So, whenever they add a new file, it will be a Kotlin file; furthermore, when they refactor old Java code, they will migrate it to Kotlin. Over time, there is more and more Kotlin code until it excludes Java completely.



One example of such a project is the Kotlin compiler itself. It was initially written in Java, but more and more files were migrated to Kotlin when it became stable enough. This process has been happening for years now; at the time of writing this book, the Kotlin compiler project still contains around 10% of Java code.

Now that we understand the relationship between Kotlin and Java, it is time to fight some misconceptions. Many see Kotlin as a layer of syntactic sugar on top of Java, but this is not true. Kotlin is a different language than Java. It has its own conventions and practices, and it has features that Java does not have, like multiplatform capabilities and coroutines. You don't need to know Java to understand Kotlin. In my opinion, Kotlin is a better first language than Java. Junior Kotlin developers do not need to know what the `equals` method is and how to override it. For them, it is enough to know the default class and data class equality[1]. They don't need to learn to write getters and setters, or how to implement a singleton or a builder pattern. Kotlin has a lower entry threshold than Java and does not need the JVM platform.

The Kotlin IDE

The most popular Kotlin IDEs (integrated development environments) are IntelliJ IDEA and Android Studio. However, you can also write programs in Kotlin using VS Code, Eclipse, Vim, Emacs, Sublime Text, and many more. You can also write Kotlin code online, for instance, using the official online IDE that can be found at this link play.kotlinlang.org.

Where do we use Kotlin?

Kotlin can be used as an alternative to Java, JavaScript, C++, Objective-C, etc. However, it is most mature on JVM, so it is currently mainly used as an alternative to Java.

Kotlin has become quite popular for backend development. I most often see it used with the Spring framework, but some projects use Kotlin with backend frameworks like Vert.x, Ktor, Micronaut, http4k or Javalin.

Kotlin has also practically become the standard language for Android development. Google has officially suggested that all Android applications should be written in Kotlin[2] and has announced that all their APIs will be designed primarily for Kotlin[3].

[1] It will be explained in the chapter *Data classes*.
[2] Source: techcrunch.com/2022/08/18/five-years-later-google-is-still-all-in-on-kotlin/
[3] Source: developer.android.com/kotlin/first

More and more projects are now taking advantage of the fact that Kotlin can be compiled for a few different platforms because this means that teams can write code that runs on both Android and iOS, or on both the backend and the frontend. Moreover, this cross-platform compatibility means that library creators can create one library for multiple platforms at the same time. Kotlin's multiplatform capabilities are already being used in many companies, and they are getting more and more popular.

It is also worth mentioning Jetpack Compose, which is a toolkit for building native UIs in Kotlin. It was initially developed for Android, but it uses Kotlin's multiplatform capabilities and can also be used to create views for websites, desktop applications, iOS applications, and other targets[4].

A lot of developers are using Kotlin for front-end development, mainly using React, and there is also a growing community of data scientists using Kotlin.

As you can see, there is already a lot that you can do in Kotlin, and there are more and more possibilities as each year passes. I am sure you will find good ways to apply your new knowledge once you've finished reading this book.

---

[4] At the moment, the maturity of these targets differs.

Your first program in Kotlin

The first step in our Kotlin adventure is to write a minimal program in this language. Yes, it's the famous "Hello, World!" program. This is what it looks like in Kotlin:

```
fun main() {
    println("Hello, World")
}
```

This is minimal, isn't it? We need no classes (like we do in Java), no objects (like `console` in JavaScript), and no conditions (like in Python when we start code in the IDE). We need the `main` function and the `println` function call with some text[5].

This is the most popular (but not the only) variant of the "main" function. If we need arguments, we might include a parameter of type `Array<String>`:

```
fun main(args: Array<String>) {
    println("Hello, World")
}
```

There are also other forms of the `main` function:

```
fun main(vararg args: String) {
    println("Hello, World")
}
```

[5] The `println` function is implicitly imported from the standard library package `kotlin.io`.

```
class Test {
    companion object {
        @JvmStatic
        fun main(args: Array<String>) {
            println("Hello, World")
        }
    }
}
```

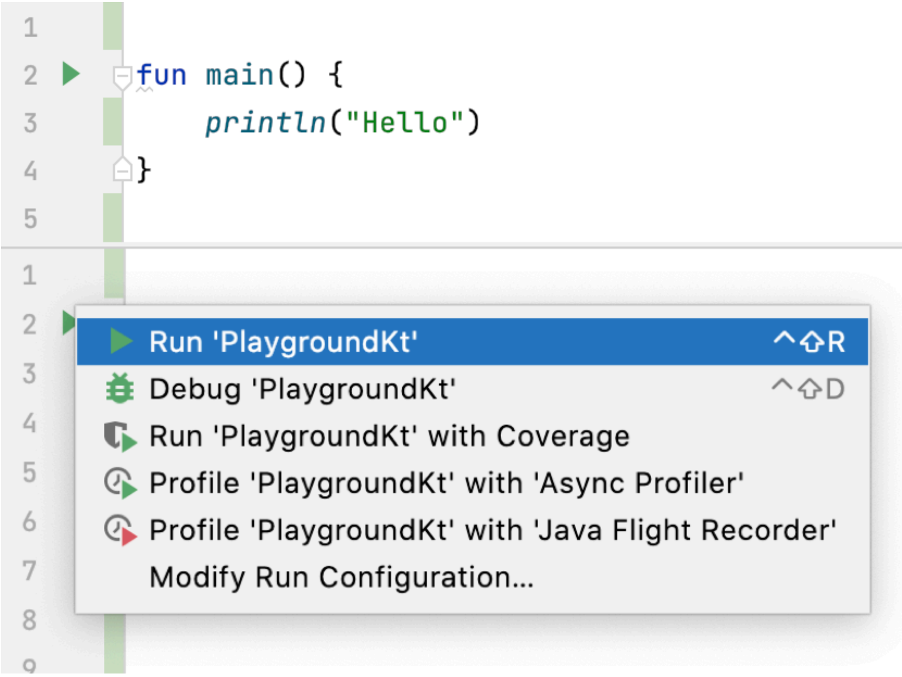
```
suspend fun main() {
    println("Hello, World")
}
```

Although these are all valid, let's concentrate on the simple `main` function as we will find it most useful. I will use it in nearly every example in this book. Such examples are usually completely executable if you just copy-paste them into IntelliJ or the Online Kotlin Playground[6].

```
fun main() {
    println("Hello, World")
}
```

All you need to do to start the `main` function in IntelliJ is click the green triangle which appears on the left side of the `main` function; this is called the "gutter icon", also known as the "Run" button.

[6]You can also find some chapters of this book online on the Kt. Academy website. These examples can be started and modified thanks to the Kotlin Playground feature.

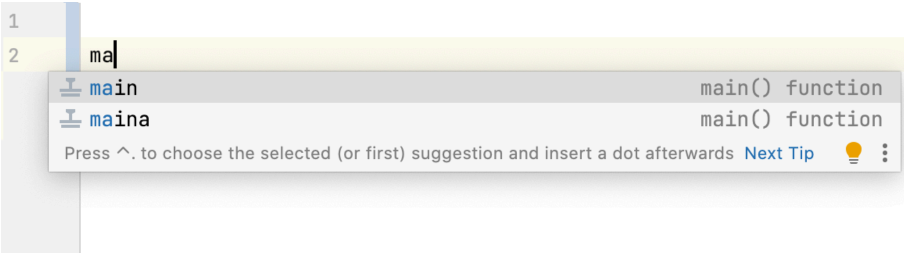


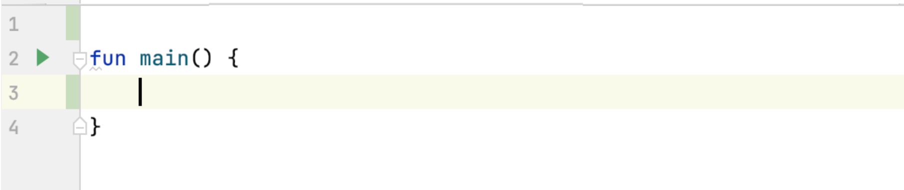
Live templates

If you decide to test or practice the material from this book[7], you will likely be writing the `main` function quite often. Here come *live templates* to help us. This is an IntelliJ feature that suggests using a template when you start typing its name in a valid context. So, if you start typing “main” or “maina” (for main with arguments) in a Kotlin file, you will be shown a suggestion that offers the whole `main` function.

---

[7] It makes me happy when people try to challenge what I am teaching. Be skeptical, and verify what you’ve learned; this is a great way to learn something new and deepen your understanding.





In most my workshops, I've used this template hundreds of times. Whenever I want to show something new with live coding, I open a "Playground" file, select all its content (Ctrl/command + A), type "main", confirm the live template with Enter, and I have a perfect space for showing how Kotlin works.

I also recommend you test this now. Open any Kotlin project (it is best if you have a dedicated project for playing with Kotlin), create a new file (you can name it "Test" or "Playground"), and create the `main` function with the live template "maina". Use the `print` function with some text, and run the code with the Run button.

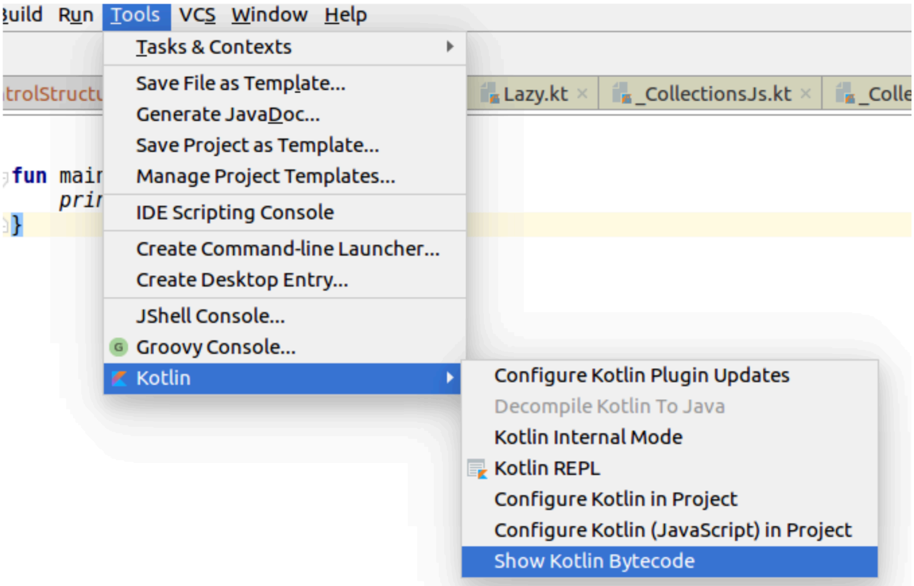
What is under the hood on JVM?

The most important target for Kotlin is JVM (Java Virtual Machine). On JVM, every element needs to be in a class. So, you might be wondering how it is possible that our main function can be started there if it is not in a class. Let's figure it out. On the way, we will learn to find out what our Kotlin code would look like if it were written in Java. This is Java developers' most powerful tool for learning how Kotlin works.

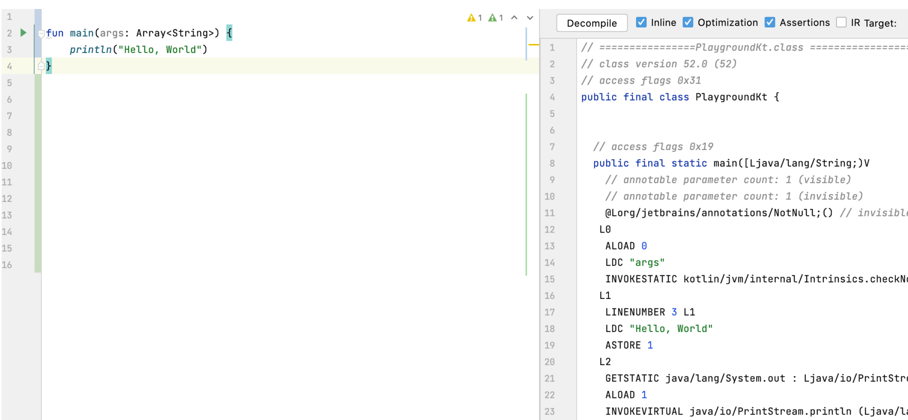
Let's start by opening or starting a Kotlin project in IntelliJ or Android Studio. Make a new Kotlin file called "Playground". Inside this, use the live template "maina" to create the main function with arguments and add `println("Hello, World")` inside.

```
fun main(args: Array<String>) {
    println("Hello, World")
}
```

Now, select from the tabs Tools > Kotlin > Show Kotlin Bytecode.



On the right side, a new tool should open. "Show Kotlin Bytecode" shows the JVM bytecode generated from this file.



This is a great place for everyone who likes reading JVM bytecode. Since not everyone is Jake Wharton, most of us might find the "Decompile" button useful. What it does is quite funny. We've just compiled our Kotlin code into JVM bytecode, and this button decompiles this bytecode into Java. As a result, we can see what our code would look like if it were written in Java[8].

```
public final class PlaygroundKt {
   public static final void main(@NotNull String[] args) {
      Intrinsics.checkNotNullParameter(args, paramName: "args");
      String var1 = "Hello, World";
      System.out.println(var1);
   }
}
```

This code reveals that our `main` function on JVM becomes a static function inside a class named `PlaygroundKt`. Where does this name come from? Try to guess. Yes, this is, by default, the file's name with the "Kt" suffix. The same happens to all other functions and properties defined outside of classes on JVM.

If we wanted to call our `main` function from Java code, we can call `PlaygroundKt.main({})`.

The name of `PlaygroundKt` can be changed by adding the `@file:JvmName("NewName")` annotation at the top of the file[9]. However, this does not change how elements defined in this file are used in Kotlin. It only influences how we will use such functions from Java. For example, to call our `main` function from Java now, we would need to call `NewName.main({})`.

If you have experience with Java, remember this tool as it can help you to understand:

- How Kotlin code works on a low level.
- How a certain Kotlin feature works under the hood.
- How to use a Kotlin element in Java.

There are proposals to make a similar tool to show JavaScript generated from Kotlin code when our target is Kotlin/JS. However, at the time of writing this book, the best you can do is to open the generated files yourself.

Packages and importing

When our project has more than one file, we need to use packages to organize them. Packages are a way to group files together and avoid name conflicts.

A file can specify package at the top of the file using the `package` keyword.

---

[8] This doesn't always work because the decompiler is not perfect, but it is really helpful anyway.

[9] More about this in the book *Advanced Kotlin*, chapter *Kotlin and Java interoperability*.

```
package com.marcinmoskala.domain.model

class User(val name: String)
```

If we don't specify a package, the file is in the default package. In real projects, it is recommended that package path should be the same as the directory path in our source files. Package can also include company domain in reverse order, like `com.marcinmoskala`. We name package using lowercase characters only.

If we want to use a function or class from another package, we need to import it. Imports are declared after the package declaration and before file elements'[10] declaration. They first specify the package name, then the name of the element. We can also use the `*` character to import all elements from a package.

```
package com.marcinmoskala.domain

import com.marcinmoskala.domain.model.User
// or
import com.marcinmoskala.domain.model.*

fun useUser() {
    val user = User("Marcin")
    // ...
}
```

Essential elements from Kotlin and Java strandard libraries are imported by default. For example, we can use `println` function without importing it.

Kotlin's developers rarely think about imports, because IntelliJ manage them automatically. When you use an element using IntelliJ suggestion, it will automatically add an import for you. If you use an element that is not imported, IntelliJ will suggest importing it. If you want to remove unused imports, you can use the "Remove unused imports" action (Ctrl/command + Alt + O). That is also why I decided to not show imports in most of the examples in this book.

Summary

We've learned about using `main` functions and creating them easily with live templates. We've also learned how to find out what our Kotlin code would look like if it were written in Java. For me, it seems like we have quite a nice toolbox for starting our adventure. So, without further ado, let's get on with that.

[10]By elements in the context of Kotlin programming we mean classes, functions, properties, object, interfaces, etc. We will discuss all element types in the following chapters.

Exercise: Your first program

Your task is to create a program that will print "Hello, World" to the console and then check what was the code compiled to on JVM.

1. Install IntelliJ IDEA if you do not have it yet.
2. Create a new Kotlin project.
3. Create a new Kotlin file.
4. Create a `main` function using Live Template "main".
5. Use the `println` function to print "Hello, World" to the console.
6. Run the program using the Run button (gutter icon).
7. Check the output in the Run tool window.
8. Select "Show Kotlin Bytecode" from the Tools > Kotlin menu.
9. Click "Decompile" in the Kotlin Bytecode window.
10. Check the output, and compare this generated Java file with the original Kotlin code.

Variables

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Basic types, their literals and operations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Numbers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Underscores in numbers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Other numeral systems

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

`Number` and conversion functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Operations on numbers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Operations on bits

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

`BigDecimal` and `BigInteger`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Booleans

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Equality

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Boolean operations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Characters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Strings

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: Basic values operations

Conditional statements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

if-statement

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

when-statement

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

when-statement with a value

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

is check

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Explicit casting

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Smart-casting

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

While and do-while statements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: Using when

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: Pretty time display

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Functions

When Andrey Breslav, the initial Kotlin creator, was asked about his favourite feature during a discussion panel at KotlinConf Amsterdam, he said it was functions[11]. In the end, functions are our programs' most important building blocks. If you look at real-life applications, most of the code either defines or calls functions.

```kotlin
class AppsRepository(private val context: Context, private val prefs: AppPrefs) {

    private val excludeSystem get() = prefs.settings.excludeSystem
    private val excludeDisabled get() = prefs.settings.excludeDisabled
    private val excludeStore get() = prefs.settings.excludeStore
    private val ignoredApps get() = prefs.ignoredApps()

    private fun getPackageInfos(options: Int = 0): Sequence<PackageInfo> = runCatching {
        return context.packageManager.getInstalledPackages(options).asSequence()
            .filter { !excludeSystem || it.applicationInfo.flags and ApplicationInfo.FLAG_SYSTEM == 0 }
            .filter { !excludeSystem || it.applicationInfo.flags and ApplicationInfo.FLAG_UPDATED_SYSTEM_APP == 0 }
            .filter { !excludeDisabled || it.applicationInfo.enabled }
            .filter { !excludeStore || !isAppStore(context.packageManager.getInstallerPackageName(it.packageName)) }
    }.getOrElse {
        Log.e("AppsRepository", "getPackageInfos", it)
        return sequenceOf()
    }

    fun getPackageInfosFiltered(options: Int = 0) = getPackageInfos(options).filter { !ignoredApps.contains(it.packageName) }

    fun getApps(options: Int = 0) = getPackageInfos(options).mapIndexed { i, app ->
        AppInstalled(
            i,
            app.name(context),
            app.packageName,
            app.versionName ?: "",
            app.versionCode,
            iconUri(app.packageName, app.applicationInfo.icon),
            ignoredApps.contains(app.packageName)
        )
    }.sortedBy { it.name }.sortedBy { it.ignored }.toList()

    fun getAppsFiltered(apps: Sequence<PackageInfo>) = apps.mapIndexed { i, app ->
        AppInstalled(
            i,
            app.name(context),
            app.packageName,
            app.versionName ?: "",
            app.versionCode,
            iconUri(app.packageName, app.applicationInfo.icon),
            ignoredApps.contains(app.packageName)
        )
    }.sortedBy { it.name }.sortedBy { it.ignored }

    // Checks if Play Store or Amazon Store
    private fun isAppStore(name: String?) = name?.contains("com.android.vending").orFalse() || name?.contains("com.amazon").orFalse()
```

As an example, I used a random class from the APKUpdater open-source project. Notice that nearly every line either defines or calls a function.

In Kotlin, we define functions using the `fun` keyword. This is why we have so much "fun" in Kotlin. With a bit of creativity, a function can consist only of `fun`:

[11]Source: youtu.be/heqjfkS4z2I?t=660

```
fun <Fun> `fun`(`fun`: Fun): Fun = `fun`
```

> This is the so-called *identity function*, a function that returns its argument without any modifications. It has a generic type parameter `Fun`, but this will be explained in the chapter *Generics*.

By convention, we name functions using lower camelCase syntax[12]. Formally, we can use characters, underscore _, and numbers (but not at the first position), but in general just characters should be used.

In Kotlin, we name functions with lowerCamelCase.

This is what a typical function looks like:

```
fun square(x: Double): Double {
    return x * x
}

fun main() {
    println(square(10.0)) // 100.0
}
```

Notice that the parameter type is specified after the variable name and a colon, and the result type is specified after a colon inside the parameter brackets. Such

[12]This rule has some exceptions. For example, on Android, Jetpack Compose functions should be named using UpperCamelCase by convention. Also, unit tests are often named with full sentences inside braces.

notation is typical of languages with powerful support for type inference because it is easier to add or remove explicit type definitions.

```
val a: Int = 123
// easy to transform from or to
val a = 123

fun add(a: Int, b: Int): Int = a + b

// easy to transform from or to
fun add(a: Int, b: Int) = a + b
```

To use a reserved keyword as a function name (like `fun` or `when`), use backticks, as in the example below. When a function has an illegal name, both its definition and calls require backticks.

Another use case for backticks is naming unit-test functions so that they can be described in plain English, as in the example below. This is not standard practice, but it is still quite a popular practice that many teams choose to adopt.

```
class CartViewModelTests {
    @Test
    fun `should show error dialog when no items loaded`() {
        ...
    }
}
```

Single-expression functions

Many functions in real-life projects just have a single expression[13], so they start and immediately use the `return` keyword. The `square` function defined above is a great example. For such functions, instead of defining the body with braces, we can use the equality sign (=) and just specify the expression that calculates the result without specifying `return`. This is *single-expression syntax*, and functions that use it are called *single-expression functions*.

[13]As a reminder, an expression is a part of our code that returns a value.

```kotlin
fun square(x: Double): Double = x * x

fun main() {
    println(square(10.0)) // 100.0
}
```

An expression can be more complicated and take multiple lines. This is fine as long as its body is a single statement.

```kotlin
fun findUsers(userFilter: UserFilter): List<User> =
    userRepository
        .getUsers()
        .map { it.toDomain() }
        .filter { userFilter.accepts(it) }
```

When we use single-expression function syntax, we can infer the result type. We don't need to, as explicit result type might still be useful for safety and readability[14], but we can.

```kotlin
fun square(x: Double) = x * x

fun main() {
    println(square(10.0)) // 100.0
}
```

Functions on all levels

Kotlin allows us to define functions on many levels, but this isn't very obvious as Java only allows functions inside classes. In Kotlin, we can define:

- functions in files outside any classes, called **top-level functions**,
- functions inside classes or objects, called **member functions** (they are also called **methods**),
- functions inside functions, called **local functions** or **nested functions**.

[14]See *Effective Kotlin Item 14: Consider referencing receivers explicitly*

```kotlin
// Top-level function
fun double(i: Int) = i * 2

class A {
    // Member function (method)
    private fun triple(i: Int) = i * 3

    // Member function (method)
    fun twelveTimes(i: Int): Int {
        // Local function
        fun fourTimes() = double(double(i))
        return triple(fourTimes())
    }
}

// Top-level function
fun main(args: Array<String>) {
    double(1) // 2
    A().twelveTimes(2) // 24
}
```

Top-level functions (defined outside classes) are often used to define utils, small but useful functions that help us with development. Top-level functions can be moved and split across files. In many cases, top-level functions in Kotlin are better than static functions in Java. Using them seems intuitive and convenient for developers.

However, it's a different story with local functions (defined inside functions). I often see that developers lack the imagination to use them (due to lack of exposure to them). Local functions are popular in JavaScript and Python, but there's nothing like this in Java. The power of local functions is that they can directly access or modify local variables. They are used to extract repetitive code inside a function that operates on local variables. Longer functions should tell a "story", and local subroutines can wrap a block expression in a descriptive name.

Take a look at the below example, which presents a function that validates a form. It checks conditions for the form fields. If a condition is not matched, we should show an error and change the local variable `isValid` to `false`, in which case we should not return from the function because we want to check all the fields (we should not stop at the first one that fails). This is an example of where a local function can help us extract repetitive behavior.

```
fun validateForm() {
    var isValid = true
    val errors = mutableListOf<String>()
    fun addError(view: FormView, error: String) {
        view.error = error
        errors += error
        isValid = false
    }

    val email = emailView.text
    if (email.isBlank()) {
        addError(emailView, "Email cannot be empty or blank")
    }

    val pass = passView.text.trim()
    if (pass.length < 3) {
        addError(passView, "Password too short")
    }

    if (isValid) {
        tryLogin(email, pass)
    } else {
        showErrors(errors)
    }
}
```

Parameters and arguments

A variable defined as a part of a function definition is called a **parameter**. The value that is passed when we call a function is called an **argument**.

```
fun square(x: Double) = x * x // x is a parameter

fun main() {
    println(square(10.0)) // 10.0 is an argument
    println(square(0.0)) // 0.0 is an argument
}
```

In Kotlin, parameters are read-only, so we cannot reassign their value.

```
fun a(i: Int) {
    i = i + 10 // ERROR
    // ...
}
```

If you need to modify a parameter variable, the only way is to shadow it with a local variable that is mutable.

```
fun a(i: Int) {
    var i = i + 10
    // ...
}
```

This is possible but discouraged. A parameter holds a value that was used as an argument, and this value should not change. A local read-write variable represents a different concept and should therefore have a different name.

`Unit` return type

In Kotlin, all functions have a result type, so every function call is an expression. When a type is not specified, the default result type is `Unit`, and the default result value is the `Unit` object.

```
fun someFunction() {}

fun main() {
    val res: Unit = someFunction()
    println(res) // kotlin.Unit
}
```

`Unit` is just a very simple object that is used as a placeholder when nothing else is returned. When you specify a function without an explicit result type, its result type will implicitly be `Unit`. When you define a function without `return` in the last line, it is the same as using `return` with no value. Using `return` with no value is the same as returning `Unit`.

```
fun a() {}

// the same as
fun a(): Unit {}

// the same as
fun a(): Unit {
    return
}

// the same as
fun a(): Unit {
    return Unit
}
```

Vararg parameters

Each parameter expects one argument, except for parameters marked with the `vararg` modifier. Such parameters accept any number of arguments.

```
fun a(vararg params: Int) {}

fun main() {
    a()
    a(1)
    a(1, 2)
    a(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
}
```

A good example of such a function is `listOf`, which produces a list from values used as arguments.

```
fun main() {
    println(listOf(1, 3, 5, 6)) // [1, 3, 5, 6]
    println(listOf("A", "B", "C")) // [A, B, C]
}
```

This means a vararg parameter holds a collection of values, therefore it cannot have the type of a single object. So the vararg parameter represents an array of the declared type, and we can iterate over arrays using a for loop (which will be explained in more depth in the next chapter).

```
fun concatenate(vararg strings: String): String {
    // The type of `strings` is Array<String>
    var accumulator = ""
    for (s in strings) accumulator += s
    return accumulator
}

fun sum(vararg ints: Int): Int {
    // The type of `ints` is IntArray
    var accumulator = 0
    for (i in ints) accumulator += i
    return accumulator
}

fun main() {
    println(concatenate()) //
    println(concatenate("A", "B")) // AB
    println(sum()) // 0
    println(sum(1, 2, 3)) // 6
}
```

We will get back to vararg parameters in the chapter *Collections*, in the section dedicated to arrays.

Named parameter syntax and default arguments

When we declare functions, we often specify optional parameters. A good example is `joinToString`, which transforms an iterable into a `String`. It can be used without any arguments, or we might change its behavior with concrete arguments.

```
fun main() {
    val list = listOf(1, 2, 3, 4)
    println(list.joinToString()) // 1, 2, 3, 4
    println(list.joinToString(separator = "-")) //  1-2-3-4
    println(list.joinToString(limit = 2)) //  1, 2, ...
}
```

Many more functions in Kotlin use optional parametrization, but how is this done? It is enough to place an equality sign after a parameter and then specify the default value.

```
fun cheer(how: String = "Hello,", who: String = "World") {
    println("$how $who")
}

fun main() {
    cheer() // Hello, World
    cheer("Hi") // Hi World
}
```

Values specified this way are created on-demand when there is no parameter for their position. This is not Python, therefore they are not stored statically, which is why it's safe to use mutable values as default arguments.

```
fun addOneAndPrint(list: MutableList<Int> = mutableListOf()) {
    list.add(1)
    println(list)
}

fun main() {
    addOneAndPrint() // [1]
    addOneAndPrint() // [1]
    addOneAndPrint() // [1]
}
```

> In Python, the analogous code would produce `[1]`, `[1, 1]`, and `[1, 1, 1]`.

When we call a function, we can specify an argument's position by its parameter name, like in the example below. This way, we can specify later optional positions without specifying previous ones. This is called *named parameter syntax*.

```
fun cheer(how: String = "Hello,", who: String = "World") {
    print("$how $who")
}

fun main() {
    cheer(who = "Group") // Hello, Group
}
```

Named parameter syntax is very useful for improving our code's readability. When an argument's meaning is not clear, it is better to specify a parameter name for it.

```
fun main() {
    val list = listOf(1, 2, 3, 4)
    println(list.joinToString("-")) // 1-2-3-4
    // better
    println(list.joinToString(separator = "-")) //  1-2-3-4
}
```

Naming arguments also prevents mistakes that are a result of changing parameter positions.

```
class User(
    val name: String,
    val surname: String,
)

val user = User(
    name = "Norbert",
    surname = "Moskała",
)
```

In the above example, without named arguments a developer might flip the `name` and `surname` positions; if named arguments were not used here, this would lead to an incorrect name and surname in the object. Named arguments protect us from such situations.

It is considered a good practice to use the named arguments convention when we call functions with many arguments, some of whose meanings might not be obvious to developers reading our code in the future.

Function overloading

In Kotlin, we can define functions with the same name in the same scope (file or class) as long as they have different parameter types or a different number of parameters. This is known as function **overloading**. Kotlin decides which function to execute based on the types of the specified arguments.

```
fun a(a: Any) = "Any"
fun a(i: Int) = "Int"
fun a(l: Long) = "Long"

fun main() {
    println(a(1)) // Int
    println(a(18L)) // Long
    println(a("ABC")) // Any
}
```

A practical example of function overloading is providing multiple function variants for user convenience.

```
import java.math.BigDecimal

class Money(val amount: BigDecimal, val currency: String)

fun pln(amount: BigDecimal) = Money(amount, "PLN")
fun pln(amount: Int) = pln(amount.toBigDecimal())
fun pln(amount: Double) = pln(amount.toBigDecimal())
```

Infix syntax

Methods with a single parameter can use the `infix` modifier, which allows a special kind of function call: without the dot and the argument parentheses.

```
class View
class ViewInteractor {
    infix fun clicks(view: View) {
        // ...
    }
}

fun main() {
    val aView = View()
    val interactor = ViewInteractor()

    // regular notation
    interactor.clicks(aView)
    // infix notation
    interactor clicks aView
}
```

This notation is used by some functions from Kotlin stdlib (Standard Library), like the `and`, `or` and `xor` bitwise operations on numbers (presented in the chapter *Basic types, their literals and operations*).

```
fun main() {
    // infix notation
    println(0b011 and 0b001) // 1, that is 0b001
    println(0b011 or 0b001) // 3, that is 0b011
    println(0b011 xor 0b001) // 2, that is 0b010

    // regular notation
    println(0b011.and(0b001)) // 1, that is 0b001
    println(0b011.or(0b001)) // 3, that is 0b011
    println(0b011.xor(0b001)) // 2, that is 0b010
}
```

Infix notation is only for our convenience. It is an example of Kotlin syntactic sugar - syntax that is designed only to make things easier to read or express.

> Regarding the position of operators or functions in relation to their operands or arguments, we use three kinds of position types: prefix, infix, and postfix. Prefix notation is when we place the operator or function **before** the operands or arguments[15]. A good example is a plus or minus placed before a single number (like `+12` or `-3.14`). One might argue that a top-level function call also uses prefix notation because the function name comes before the arguments (like `maxOf(10, 20)`). Infix notation is when we place the operator or function **between** the operands or arguments[16]. A good example is a plus or minus between two numbers (like `1 + 2` or `10 - 7`). One might argue that a method call with arguments also uses infix notation because the function name comes between the receiver (the object we call this method on) and arguments (like `account.add(money)`). In Kotlin, we use the term "infix notation" more restrictively to reference the special notation we use for methods with the `infix` modifier. Postfix notation is when we place the operator or function **after** the operands or arguments[17]. In modern programming, postfix notation is practically not used anymore. One might argue that calling a method with no arguments is postfix notation, as in `str.uppercase()`.

[15] From the Latin word praefixus, which means "fixed in front".

[16] From the Latin word infixus, the past participle of infigere, which we might translate as "fixed in between".

[17] Made from the prefix "post-", which means "after, behind", and the word "fix", meaning "fixed in place".

Function formatting

When a function declaration (name, parameters, and result type) is too long to fit in a single line, we split it such that every parameter definition is on a different line, and the beginning and end of the function declaration are also on separate lines.

```
fun veryLongFunction(
    param1: Param1Type,
    param2: Param2Type,
    param3: Param3Type,
): ResultType {
    // body
}
```

Classes are formatted in the same way[18]:

```
class VeryLongClass(
    val property1: Type1,
    val property2: Type2,
    val property3: Type3,
) : ParentClass(), Interface1, Interface2 {
    // body
}
```

When a function call[19] is too long, we format it similarly: each argument is on a different line. However, there are exceptions to this rule, such as keeping multiple vararg parameters on the same line.

```
fun makeUser(
    name: String,
    surname: String,
): User = User(
    name = name,
    surname = surname,
)

class User(
    val name: String,
    val surname: String,
```

[18]We will discuss classes later in this book, in the chapter *Classes and interfaces*.
[19]A constructor call is also considered a function call in Kotlin.

```
)

fun main() {
    val user = makeUser(
        name = "Norbert",
        surname = "Moskała",
    )

    val characters = listOf(
        "A", "B", "C", "D", "E", "F", "G", "H", "I", "J",
        "K", "L", "M", "N", "O", "P", "R", "S", "T", "U",
        "W", "X", "Y", "Z",
    )
}
```

In this book, the width of my lines is much smaller than in regular projects, so I am forced to break lines much more often than I would like to.

Notice that when I specify arguments or parameters, I sometimes add a comma at the end. This is a so-called **trailing comma**. Such notation is optional.

```
fun printName(
    name: String,
    surname: String, // <- trailing comma
) {
    println("$name $surname")
}

fun main() {
    printName(
        name = "Norbert",
        surname = "Moskała", // <- trailing comma
    )
}
```

I like using trailing comma notation because it makes it easier to add another element in the future. Without it, adding or removing an element requires not only a new line but also an additional comma after the last element. This leads to meaningless line modifications on Git, which makes it harder to read what has actually changed in our project. Some developers don't like trailing comma notation, which can sometimes lead to a holy war. Decide in your team if you like it or not, and be consistent in your projects.

```
fun printName(
    name: String,
    surname: String,
) {
-   println("$name $surname")
}

fun main() {
    printName(
        name = "Norbert",
        surname = "Moskała",
    )
}
```

```
fun printName(
    name: String,
    surname: String,
+   middleName: String? = null,
) {
+   if (middleName != null) {
+       println("$name $middleName $surname")
+   } else {
+       println("$name $surname")
+   }
}

fun main() {
    printName(
        name = "Norbert",
        surname = "Moskała",
+       middleName = "Jan",
    )
}
```

Adding a parameter and an argument on git when a trailing comma is used.

```

fun printName(
    name: String,
-   surname: String
) {
-   println("$name $surname")
}

fun main() {
    printName(
        name = "Norbert",
-       surname = "Moskała"
    )
}
```

```

fun printName(
    name: String,
+   surname: String,
+   middleName: String? = null
) {
+   if (middleName != null) {
+       println("$name $middleName $surname")
+   } else {
+       println("$name $surname")
+   }
}

fun main() {
    printName(
        name = "Norbert",
+       surname = "Moskała",
+       middleName = "Jan"
    )
}
```

Adding a parameter and an argument on git when a trailing comma is not used.

Summary

As you can see, functions in Kotlin have a lot of powerful features. Single-expression syntax makes simple functions shorter. Named and default arguments help us improve safety and readability. The `Unit` result type makes every function call an expression. Vararg parameters allow any number of arguments to be used for one parameter position. Infix notation introduces a more convenient way to call certain kinds of functions. Trailing commas minimize the number of changes on git. All this is for our convenience. For now though, let's move on to another topic: using a for-loop.

Exercise: Person details display

Your task is to implement `formatPersonDisplay` function: It should have `String` result type and the following parameters:

- `name` of type `String?` and default value `null`.
- `surname` of type `String?` and default value `null`.
- `age` of type `Int?` and default value `null`.

> Beware! Parameter types should include `?`, so those should be `String?` and `Int?` instead of `String` and `Int`. This is because we want to allow passing `null` as a parameter value. This will be explained in the chapter *Nullability*.

Function should return a string in the following format: `"{name} {surname} ({age})"`. If any of the parameters is `null`, it should be omitted from the result. If all parameters are `null`, it should return an empty string.

Here are some examples of how the function should work:

```
println(formatPersonDisplay("John", "Smith", 42))
// John Smith (42)
println(formatPersonDisplay("Alex", "Simonson"))
// Alex Simonson
println(formatPersonDisplay("Peter", age = 25))
// Peter (25)
println(formatPersonDisplay(surname="Johnson", age=18))
// Johnson (18)
```

Example usage and unit tests can be found in the MarcinMoskala/kotlin-exercises project on GitHub in the file `essentials/functions/PersonDisplay.kt`. You can clone this project and solve this exercise locally.

In this project, example usage and unit tests are commented not to prevent other files from compilation. To uncomment them, select commented lines and use `command + /` on Mac (`Ctrl + /` on Windows)

Hint: You can use `trim` function on a string to remove leading and trailing whitespace characters.

The power of the for-loop

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Ranges

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Break and continue

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Use cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: Range Operations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Nullability

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Safe calls

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Not-null assertion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Smart-casting

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

The Elvis operator

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Extensions on nullable types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

`null` is our friend

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

lateinit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: User Information Processor

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Classes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Member functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Properties

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Constructors

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Classes representing data in Kotlin and Java

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Inner classes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: Implementing the Product class

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Inheritance

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Overriding elements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Parents with non-empty constructors

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Super call

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Abstract class

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Visibility

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Any

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: GUI View Hierarchy Simulation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Data classes

In Kotlin, we say that all classes inherit from the `Any` superclass, which is at the top of the class hierarchy[20]. Methods defined in `Any` can be called on all objects. These methods are:

- `equals` - used when two objects are compared using `==`,
- `hashCode` - used by collections that use the hash table algorithm,
- `toString` - used to represent an object as a string, e.g., in a string template or the `print` function.

Thanks to these methods, we can represent any object as a string or check the equality of any two objects.

```
// Any formal definition
open class Any {
   open operator fun equals(other: Any?): Boolean
   open fun hashCode(): Int
   open fun toString(): String
}

class A // Implicitly inherits from Any

fun main() {
   val a = A()
   a.equals(a)
   a == a
   a.hashCode()
   a.toString()
   println(a)
}
```

> Truth be told, `Any` is represented as a class, but it should actually be considered the head of the type hierarchy, but with some special functions. Consider the fact that `Any` is also the supertype of all interfaces, even though interfaces cannot inherit from classes.

[20]So `Any` is an analog to `Object` in Java, JavaScript or C#. There is no direct analog in C++.

The default implementations of `equals`, `hashCode`, and `toString` are strongly based on the object's address in memory. The `equals` method returns `true` only when the address of both objects is the same, which means the same object is on both sides. The `hashCode` method typically transforms an address into a number. `toString` produces a string that starts with the class name, then the at sign "@", then the unsigned hexadecimal representation of the hash code of the object.

```
class A

fun main() {
    val a1 = A()
    val a2 = A()

    println(a1.equals(a1)) // true
    println(a1.equals(a2)) // false
    // or
    println(a1 == a1) // true
    println(a1 == a2) // false

    println(a1.hashCode()) // Example: 149928006
    println(a2.hashCode()) // Example: 713338599

    println(a1.toString()) // Example: A@8efb846
    println(a2.toString()) // Example: A@2a84aee7
    // or
    println(a1) // Example: A@8efb846
    println(a2) // Example: A@2a84aee7
}
```

By overriding these methods, we can decide how a class should behave. Consider the following class `A`, which is equal to other instances of the same class and returns a constant hash code and string representation.

```
class A {
    override fun equals(other: Any?): Boolean = other is A

    override fun hashCode(): Int = 123

    override fun toString(): String = "A()"
}

fun main() {
    val a1 = A()
    val a2 = A()
```

```
    println(a1.equals(a1)) // true
    println(a1.equals(a2)) // true
    // or
    println(a1 == a1) // true
    println(a1 == a2) // true

    println(a1.hashCode()) // 123
    println(a2.hashCode()) // 123

    println(a1.toString()) // A()
    println(a2.toString()) // A()
    // or
    println(a1) // A()
    println(a2) // A()
}
```

I've dedicated separate items in the *Effective Kotlin* book to implementing a custom `equals` and `hashCode`[21], but in practice we rarely need to do that. As it turns out, in modern projects we almost solely operate on only two kinds of objects:

- Active objects, like services, controllers, repositories, etc. Such classes don't need to override any methods from `Any` because the default behavior is perfect for them.
- Data model class objects, which represent bundles of data. For such objects, we use the `data` modifier, which overrides the `toString`, `equals`, and `hashCode` methods. The `data` modifier also implements the methods `copy` and `componentN` (`component1`, `component2`, etc.), which are not inherited and cannot be modified[22].

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

val player = Player(0, "Gecko", 9999)
```

Let's discuss the aforementioned implicit data class methods and the differences between regular class behavior and data class behavior.

[21]These are *Item 42: Respect the contract of* `equals` and *Item 43: Respect the contract of* `hashCode`.

[22]This type of class is so popular that in Java it is common practice to autogenerate `equals`, `hashCode`, and `toString` in IntelliJ or using the Lombok library.

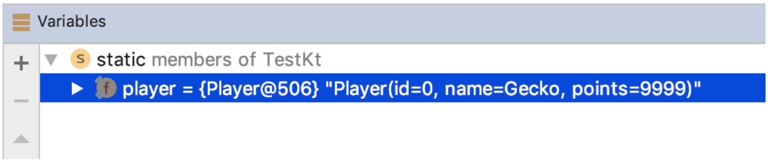
Transforming to a string

The default `toString` transformation produces a string that starts with the class name, then the at sign "@", and then the unsigned hexadecimal representation of the hash code of the object. The purpose of this is to display the class name and to determine whether two strings represent the same object or not.

```
class FakeUserRepository

fun main() {
    val repository1 = FakeUserRepository()
    val repository2 = FakeUserRepository()
    println(repository1) // e.g. FakeUserRepository@8efb846
    println(repository1) // e.g. FakeUserRepository@8efb846
    println(repository2) // e.g. FakeUserRepository@2a84aee7
}
```

With the `data` modifier, the compiler generates a `toString` that displays the class name and then pairs with the name and value for each primary constructor property. We assume that data classes are represented by their primary constructor properties, so all these properties, together with their values, are displayed during a transformation to a string. This is useful for logging and debugging.

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

fun main() {
    val player = Player(0, "Gecko", 9999)
    println(player)
    // Player(id=0, name=Gecko, points=9999)
    println("Player: $player")
    // Player: Player(id=0, name=Gecko, points=9999)
}
```



Objects equality

In Kotlin, we check the equality of two objects using `==`, which uses the `equals` method from `Any`. So, this method decides if two objects should be considered equal or not. By default, two different instances are never equal. This is perfect for active objects, i.e., objects that work independently of other instances of the same class and possibly have a protected mutable state.

```
class FakeUserRepository

fun main() {
    val repository1 = FakeUserRepository()
    val repository2 = FakeUserRepository()
    println(repository1 == repository1) // true
    println(repository1 == repository2) // false
}
```

Classes with the `data` modifier represent bundles of data; they are considered equal to other instances if:

- both are of the same class,
- their primary constructor property values are equal.

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

fun main() {
    val player = Player(0, "Gecko", 9999)
    println(player == Player(0, "Gecko", 9999)) // true
    println(player == Player(0, "Ross", 9999)) // false
}
```

This is what a simplified implementation of the `equals` method generated by the `data` modifier for the `Player` class looks like:

```
override fun equals(other: Any?): Boolean = other is Player &&
    other.id == this.id &&
    other.name == this.name &&
    other.points == this.points
```

> Implementing a custom `equals` is described in *Effective Kotlin, Item 42: Respect the contract of* `equals`.

Hash code

Another method from `Any` is `hashCode`, which is used to transform an object into an `Int`. With a `hashCode` method, the object instance can be stored in the hash table data structure implementations that are part of many popular classes, including `HashSet` and `HashMap`. The most important rule of the `hashCode` implementation is that it should:

- be consistent with `equals`, so it should return the same `Int` for equal objects, and it must always return the same hash code for the same object.
- spread objects as uniformly as possible in the range of all possible `Int` values.

The default `hashCode` is based on an object's address in memory. The `hashCode` generated by the `data` modifier is based on the hash codes of this object's primary constructor properties. In both cases, the same number is returned for equal objects.

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

fun main() {
    println(Player(0, "Gecko", 9999).hashCode()) // 2129010918
    println(Player(0, "Gecko", 9999).hashCode()) // 2129010918
    println(Player(0, "Ross", 9999).hashCode())  // 79159602
}
```

To learn more about the hash table algorithm and implementing a custom `hashCode` method, see *Effective Kotlin, Item 43: Respect the contract of* `hashCode`.

Copying objects

Another method generated by the `data` modifier is `copy`, which is used to create a new instance of a class but with a concrete modification. The idea is very simple: it is a function with parameters for each primary constructor property, but each of these parameters has a default value, i.e., the current value of the associated property.

```
// This is how copy generated by data modifier
// for Person class looks like under the hood
fun copy(
   id: Int = this.id,
   name: String = this.name,
   points: Int = this.points
) = Player(id, name, points)
```

This means we can call `copy` with no parameters to make a copy of our object with no modifications, but we can also specify new values for the properties we want to change.

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

fun main() {
   val p = Player(0, "Gecko", 9999)

   println(p.copy()) // Player(id=0, name=Gecko, points=9999)

   println(p.copy(id = 1, name = "New name"))
   // Player(id=1, name=New name, points=9999)

   println(p.copy(points = p.points + 1))
   // Player(id=0, name=Gecko, points=10000)
}
```

Note that `copy` creates a shallow copy of an object; so, if our object holds a mutable state, a change in one object will be a change in all its copies too.

```
data class StudentGrades(
    val studentId: String,
    // Code smell: Avoid using mutable objects in data classes
    val grades: MutableList<Int>
)

fun main() {
    val grades1 = StudentGrades("1", mutableListOf())
    val grades2 = grades1.copy(studentId = "2")
    println(grades1) // Grades(studentId=1, grades=[])
    println(grades2) // Grades(studentId=2, grades=[])
    grades1.grades.add(5)
    println(grades1) // Grades(studentId=1, grades=[5])
    println(grades2) // Grades(studentId=2, grades=[5])
    grades2.grades.add(1)
    println(grades1) // Grades(studentId=1, grades=[5, 1])
    println(grades2) // Grades(studentId=2, grades=[5, 1])
}
```

We do not have this problem when we use `copy` for immutable classes, i.e., classes with only `val` properties that hold immutable values. `copy` was introduced as special support for immutability (for details, see *Effective Kotlin*, *Item 1: Limit mutability*).

```
data class StudentGrades(
    val studentId: String,
    val grades: List<Int>
)

fun main() {
    var grades1 = StudentGrades("1", listOf())
    var grades2 = grades1.copy(studentId = "2")
    println(grades1) // Grades(studentId=1, grades=[])
    println(grades2) // Grades(studentId=2, grades=[])
    grades1 = grades1.copy(grades = grades1.grades + 5)
    println(grades1) // Grades(studentId=1, grades=[5])
    println(grades2) // Grades(studentId=2, grades=[])
    grades2 = grades2.copy(grades = grades2.grades + 1)
    println(grades1) // Grades(studentId=1, grades=[5])
    println(grades2) // Grades(studentId=2, grades=[1])
}
```

Notice that data classes are unsuitable for objects that must maintain invariant constraints on mutable properties. For example, in the `User` example below, the class would not be able to guarantee that the `name` and `surname` values are not blank if these variables were mutable (so, defined with `var`). Data classes are perfectly fit for immutable properties, whose constraints might be checked during the creation of these objects. In the example below, we can be sure that the `name` and `surname` values are not blank in an instance of `User`.

```
data class User(
    val name: String,
    val surname: String,
) {
    init {
        require(name.isNotBlank())
        // throws exception if name is blank
        require(surname.isNotBlank())
        // throws exception if surname is blank
    }
}
```

Destructuring

Kotlin supports a feature called position-based destructuring, which lets us assign multiple variables to components of a single object. For that, we place our variable names in round brackets.

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

fun main() {
    val player = Player(0, "Gecko", 9999)
    val (id, name, pts) = player
    println(id) // 0
    println(name) // Gecko
    println(pts) // 9999
}
```

This mechanism relies on position, not names. The object on the right side of the equality sign needs to provide the functions `component1`, `component2`, etc., and the variables are assigned to the results of these methods.

```
val (id, name, pts) = player
// is compiled to
val id: Int = player.component1()
val name: String = player.component2()
val pts: Int = player.component3()
```

This code works because the `data` modifier generates `componentN` functions for each primary constructor parameter, according to their order in the constructor.

These are currently all the functionalities that the `data` modifier provides. Don't use it if you don't need `toString`, `equals`, `hashCode`, `copy` or destructuring. If you need some of these functionalities for a class representing a bundle of data, use the `data` modifier instead of implementing the methods yourself.

When and how should we use destructuring?

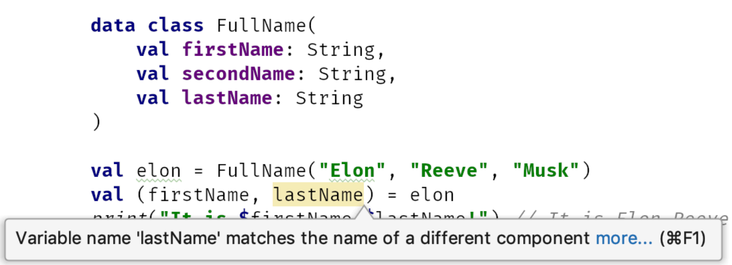
Position-based destructuring has pros and cons. Its biggest advantage is that we can name variables however we want, so we can use names like `country` and `city` in the example below. We can also destructure anything we want as long as it provides `componentN` functions. This includes `List` and `Map.Entry`, both of which have `componentN` functions defined as extensions:

```
fun main() {
    val visited = listOf("Spain", "Morocco", "India")
    val (first, second, third) = visited
    println("$first $second $third")
    // Spain Morocco India

    val trip = mapOf(
        "Spain" to "Gran Canaria",
        "Morocco" to "Taghazout",
        "India" to "Rishikesh"
    )
    for ((country, city) in trip) {
        println("We loved $city in $country")
        // We loved Gran Canaria in Spain
        // We loved Taghazout in Morocco
        // We loved Rishikesh in India
    }
}
```

On the other hand, position-based destructuring is dangerous. We need to adjust every destructuring when the order or number of elements in a data class changes. When we use this feature, it is very easy to introduce errors into our code by changing the order of the primary constructor's properties.

```
data class FullName(
   val firstName: String,
   val secondName: String,
   val lastName: String
)

val elon = FullName("Elon", "Reeve", "Musk")
val (name, surname) = elon
print("It is $name $surname!") // It is Elon Reeve!
```

We need to be careful with destructuring. It is useful to use the same names as data class primary constructor properties. In the case of an incorrect order, an IntelliJ/Android Studio warning will be shown. It might even be useful to upgrade this warning to an error.



Destructuring a single value in lambda is very confusing, especially since parentheses around arguments in lambda expressions are either optional or required in some languages.

```
data class User(
   val name: String,
   val surname: String,
)

fun main() {
   val users = listOf(
       User("Nicola", "Corti")
   )
   users.forEach { u -> println(u) }
   // User(name=Nicola, surname=Corti)
   users.forEach { (u) -> println(u) }
```

```
    // Nicola
}
```

Data class limitations

The idea behind data classes is that they represent a bundle of data; their constructors allow us to specify all this data, and we can access it through destructuring or by copying them to another instance with the `copy` method. This is why only primary constructor properties are considered by the methods defined in data classes.

```
data class Dog(
    val name: String,
) {
    // Bad practice, avoid mutable properties in data classes
    var trained = false
}

fun main() {
    val d1 = Dog("Cookie")
    d1.trained = true
    println(d1) // Dog(name=Cookie)
    // so nothing about trained property

    val d2 = d1.copy()
    println(d1.trained) // true
    println(d2.trained) // false
    // so trained value not copied
}
```

Data classes are supposed to keep all the essential properties in their primary constructor. Inside the body, we should only keep redundant immutable properties, which means properties whose value is distinctly calculated from primary constructor properties, like `fullName`, which is calculated from `name` and `surname`. Such values are also ignored by data class methods, but their value will always be correct because it will be calculated when a new object is created.

```kotlin
data class FullName(
    val name: String,
    val surname: String,
) {
    val fullName = "$name $surname"
}

fun main() {
    val d1 = FullName("Cookie", "Moskała")
    println(d1.fullName) // Cookie Moskała
    println(d1) // FullName(name=Cookie, surname=Moskała)

    val d2 = d1.copy()
    println(d2.fullName) // Cookie Moskała
    println(d2) // FullName(name=Cookie, surname=Moskała)
}
```

You should also remember that data classes must be **final** and so cannot be used as a super-type for inheritance.

Prefer data classes instead of tuples

Data classes offer more than what is generally provided by tuples. Historically, they replaced tuples in Kotlin since they are considered better practice[23]. The only tuples that are left are `Pair` and `Triple`, but these are data classes under the hood:

```kotlin
data class Pair<out A, out B>(
    val first: A,
    val second: B
) : Serializable {

    override fun toString(): String =
        "($first, $second)"
}
```

[23]Kotlin had support for tuples when it was still in the beta version. We were able to define a tuple by brackets and a set of types, like `(Int, String, String, Long)`. What we achieved behaved the same as data classes in the end, but it was far less readable. Can you guess what type this set of types represents? It can be anything. Using tuples is tempting, but using data classes is nearly always better. This is why tuples were removed, and only `Pair` and `Triple` are left.

```
data class Triple<out A, out B, out C>(
    val first: A,
    val second: B,
    val third: C
) : Serializable {

    override fun toString(): String =
        "($first, $second, $third)"
}
```

The easiest way to create a `Pair` is by using the `to` function. This is a generic infix extension function, defined as follows (we will discuss both generic and extension functions in later chapters).

```
infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

Thanks to the infix modifier, a method can be used by placing its name between arguments, as the infix name suggests. The result `Pair` is typed, so the result type from the `"ABC" to 123` expression is `Pair<String, Int>`.

```
fun main() {
    val p1: Pair<String, Int> = "ABC" to 123
    println(p1) // (ABC, 123)
    val p2 = 'A' to 3.14
    // the type of p2 is Pair<Char, Double>
    println(p2) // (A, 123)
    val p3 = true to false
    // the type of p3 is Pair<Boolean, Boolean>
    println(p3) // (true, false)
}
```

These tuples remain because they are very useful for local purposes, like:

- When we immediately name values:

```
val (description, color) = when {
    degrees < 5 -> "cold" to Color.BLUE
    degrees < 23 -> "mild" to Color.YELLOW
    else -> "hot" to Color.RED
}
```

- To represent an aggregate that is not known in advance, as is commonly the case in standard library functions:

```
val (odd, even) = numbers.partition { it % 2 == 1 }
val map = mapOf(1 to "San Francisco", 2 to "Amsterdam")
```

In other cases, we prefer data classes. Take a look at an example: let's say that we need a function that parses a full name into a name and a surname. One might represent this name and surname as a `Pair<String, String>`:

```
fun String.parseName(): Pair<String, String>? {
    val indexOfLastSpace = this.trim().lastIndexOf(' ')
    if (indexOfLastSpace < 0) return null
    val firstName = this.take(indexOfLastSpace)
    val lastName = this.drop(indexOfLastSpace)
    return Pair(firstName, lastName)
}

// Usage
fun main() {
   val fullName = "Marcin Moskała"
   val (firstName, lastName) = fullName.parseName() ?: return
}
```

The problem is that when someone reads this code, it is not clear that `Pair<String, String>` represents a full name. What is more, it is not clear what the order of the values is, therefore someone might think that the surname goes first:

```
val fullName = "Marcin Moskała"
val (lastName, firstName) = fullName.parseName() ?: return
print("His name is $firstName") // His name is Moskała
```

To make usage safer and the function easier to read, we should use a data class instead:

```
data class FullName(
    val firstName: String,
    val lastName: String
)

fun String.parseName(): FullName? {
    val indexOfLastSpace = this.trim().lastIndexOf(' ')
    if (indexOfLastSpace < 0) return null
    val firstName = this.take(indexOfLastSpace)
    val lastName = this.drop(indexOfLastSpace)
```

```
    return FullName(firstName, lastName)
}

// Usage
fun main() {
    val fullName = "Marcin Moskała"
    val (firstName, lastName) = fullName.parseName() ?: return
    print("His name is $firstName $lastName")
    // His name is Marcin Moskała
}
```

This costs nearly nothing and improves the function significantly:

- The return type of this function is more clear.
- The return type is shorter and easier to pass forward.
- If a user destructures variables with correct names but in incorrect positions, a warning will be displayed in IntelliJ.

If you don't want this class in a wider scope, you can restrict its visibility. It can even be private if you only need to use it for some local processing in a single file or class. It is worth using data classes instead of tuples. Classes are cheap in Kotlin, so don't be afraid to use them in your projects.

Summary

In this chapter, we've learned about `Any`, which is a superclass of all classes. We've also learned about methods defined by `Any`: `equals`, `hashCode`, and `toString`. We've also learned that there are two primary types of objects. Regular objects are considered unique and do not expose their details. Data class objects, which we made using the `data` modifier, represent bundles of data (we keep them in primary constructor properties). They are equal when they hold the same data. When transformed to a string, they print all their data. They additionally support destructuring and making a copy with the `copy` method. Two generic data classes in Kotlin stdlib are `Pair` and `Triple`, but (apart from certain cases) we prefer to use custom data classes instead of these. Also, for the sake of safety, when we destructure a data class, we prefer to match the variable names with the parameter names.

Now, let's move on to a topic dedicated to special Kotlin syntax that lets us create objects without defining a class.

Exercise: Data class practice

1. Create a data class for a `Person` with a `name` and `age` property of types `String` and `Int`.
2. Create a `Person` instance with name "John" and age 30.
3. Print the `Person` instance.
4. Create a copy of the `Person` instance with name "Jane".
5. Create a new `Person` instance with name "Jane" and age 30.
6. Check if the two `Person` instances are equal.
7. Print the `hashCode` of all the `Person` instances.
8. Destructure the `Person` instance created using `copy` (so the one with name "Jane") into two variables, and print values of those variables.

Objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Object expressions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Object declaration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Companion objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Data object declarations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Constant values

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: Pizza factory

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Throwing exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Defining exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Catching exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

A try-catch block used as an expression

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

The finally block

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Important exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

The hierarchy of exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: Catching exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Enum classes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Data in enum values

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Enum classes with custom methods

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: Days of the week enum

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Sealed classes and interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Sealed classes and `when` expressions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Sealed vs enum

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Use cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Annotation classes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Meta-annotations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Annotating the primary constructor

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

List literals

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Extensions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Extension functions under the hood

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Extension properties

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Extensions vs members

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Extension functions on object declarations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Member extension functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Use cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: Conversion and measurement unit creation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Collections

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

The hierarchy of interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Mutable vs read-only types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Creating collections

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Lists

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Modifying lists

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Checking a list's size or if it is empty

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Lists and indices

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Checking if a list contains an element

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Iterating over a list

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Sets

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Modifying sets

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Elements in a set are unique

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Checking a set's size or if it is empty

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Checking if a set contains an element

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Iterating over sets

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Maps

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Finding a value by a key

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Adding elements to a map

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Checking if a map contains a key

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Checking map size

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Iterating over maps

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Mutable maps

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Using arrays in practice

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Arrays of primitives

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Vararg parameters and array functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: Inventory management

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Operator overloading

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

An example of operator overloading

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Arithmetic operators

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

The `in` operator

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

The iterator operator

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

The equality and inequality operators

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Comparison operators

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

The indexed access operator

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Augmented assignments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Unary prefix operators

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Increment and decrement

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

The invoke operator

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Precedence

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: Money operations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

The beauty of Kotlin's type system

The Kotlin type system is amazingly designed. Many features that look like special cases are just a natural consequence of how the type system is designed. For instance, thanks to the type system, in the example below the type of `surname` is `String`, the type of `age` is `Int`, and we can use `return` and `throw` on the right side of the Elvis operator.

```
fun processPerson(person: Person?) {
    val name = person?.name ?: "unknown"

    val surname = person?.surname ?: return

    val age = person?.age ?: throw Error("Person must have age")

    // ...
}
```

The typing system also gives us very convenient nullability support, smart type inference, and much more. In this chapter, we will reveal a lot of Kotlin magic. I always love talking about this in my workshops because I see the stunning beauty of how Kotlin's type system is so well designed that all these pieces fit perfectly together and give us a great programming experience. I find this topic fascinating, but I will also try to add some useful hints that show where this knowledge can be useful in practice. I hope you will enjoy discovering it as much as I did.

What is a type?

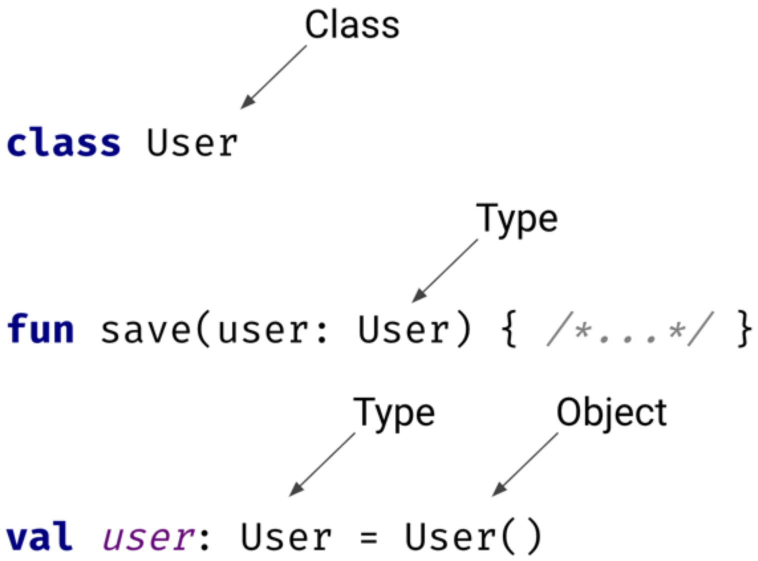
Before we start talking about the type system, we should first explain what a type is. Do you know the answer? Think about it for a moment.

Types are commonly confused with classes, but these two terms represent totally different concepts. Take a look at the example below. You can see `User` used four times. Can you tell me which usages are classes, which are types, and which are something else?

```kotlin
class User

fun save(user: User) { /*...*/ }

val user: User = User()
```

After the `class` keyword, you define a class name. A class is a template for objects that defines a set of properties and methods. When we call a constructor, we create an object. Types are used here to specify what kind of objects we expect to have in the variables[24].

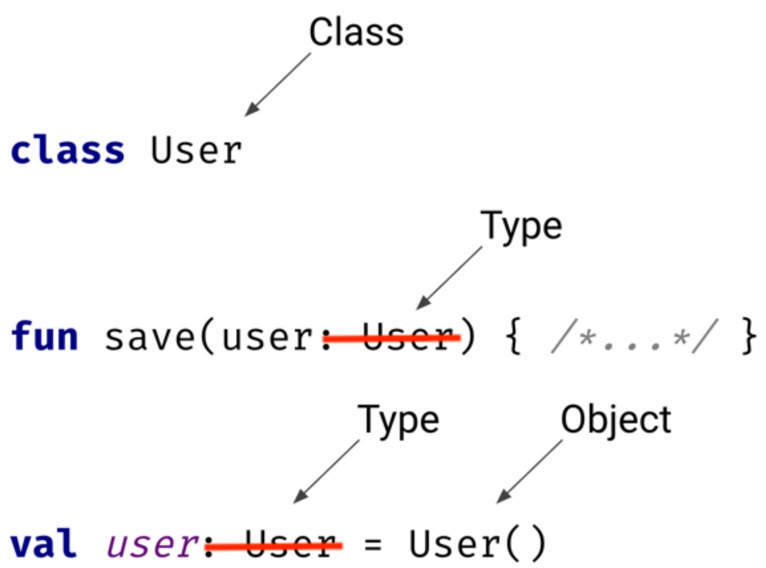
[24]Parameters are also variables.



Why do we have types?

Let's do a thought experiment for a moment. Kotlin is a statically typed language, so all variables and functions must be typed. If we do not specify their types explicitly, they will be inferred. But let's take a step back and imagine that you are a language designer who is deciding what Kotlin should look like. It is possible to drop all these requirements and eliminate all types completely. The compiler does not really need them[25]. It has classes that define how objects should be created, and it has objects that are used during execution. What do we lose if we get rid of types? Mostly safety and developers' convenience.

[25]Except when figuring out which function to choose in the case of overloading.



It is worth mentioning that many languages do support classes and objects but not types. Among them, there is JavaScript[26] and (not long ago) Python - two of the most popular languages in the world[27]. However, types do offer us value, which is why in the JavaScript community more and more people use TypeScript (which is basically JavaScript plus types), and Python has introduced support for types.

So why do we have types? They are mainly for us, developers. A type tells us what methods or properties we can use on an object. A type tells us what kind of value can be used as an argument. Types prevent the use of incorrect objects, methods, or properties. They give us safety, and suggestions are provided by the IDE. The compiler also benefits from types as they are used to better optimize our code or to decide which function should be chosen when its name is overloaded. Still, it is developers who are the most important beneficent of types.

So what is a type? **It can be considered as a set of things we can do with an object.**

---

[26]Formally, JavaScript supports weak typing, but in this chapter we discuss static typing, which is not supported by JavaScript.

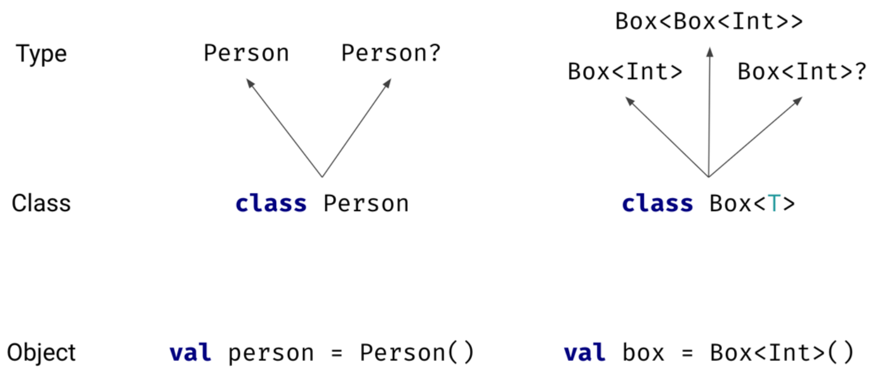
[27]It all depends on what we measure, but Python, Java, and JavaScript take the first three positions in most rankings. In some, they are beaten by C, which is widely used for very low-level development, like developing processors for cars or refrigerators.

Typically, it is a set of methods and properties.

The relation between classes and types

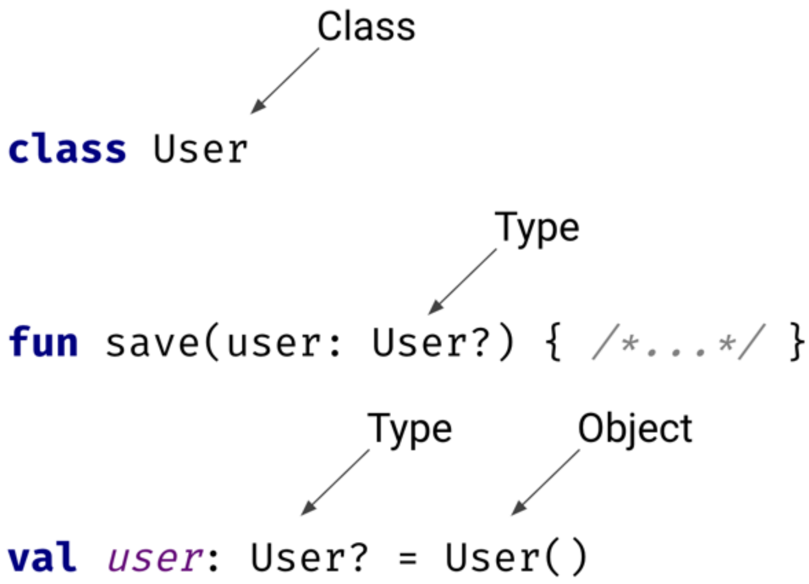
We say that classes generate types. Think of the class `User`. It generates two types. Can you name them both? One is `User`, but the second is not `Any` (`Any` is already in the type hierarchy). The second new type generated by the class `User` is `User?`. Yes, the nullable variant is a separate type.

There are classes that generate many more types: generic classes. The `Box<T>` class theoretically generates an infinite number of types.



Class vs type in practice

This discussion might sound very theoretical, but it already has some practical implications. Note that classes cannot be nullable, but types can. Consider the initial example, where I asked you to point out where `User` is a type. Only in positions that represent types can you use `User?` instead of `User`.



Member functions are defined on classes, so their receiver cannot be nullable or have type arguments[28]. Extension functions are defined on types, so they can be nullable or defined on a concrete generic type. Consider the `sum` function,, which is an extension of `Iterable<Int>`, or the `isNullOrBlank` function, which is an extension of `String?`.

```
fun Iterable<Int>.sum(): Int {
    var sum: Int = 0
    for (element in this) {
        sum += element
    }
    return sum
}

@OptIn(ExperimentalContracts::class)
inline fun CharSequence?.isNullOrBlank(): Boolean {
    // (skipped contract definition)
    return this == null || this.isBlank()
}
```

[28]Type arguments and type parameters will be better explained in the chapter *Generics*.

The relationship between types

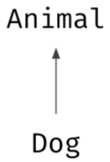
Let's say that we have a class `Dog` and its superclass `Animal`.

```
open class Animal
class Dog : Animal()
```

Wherever an `Animal` type is expected, you can use a `Dog`, but not the other way around.

```
fun petAnimal(animal: Animal) {}
fun petDog(dog: Dog) {}

fun main() {
    val dog: Dog = Dog()
    val dogAnimal: Animal = dog // works
    petAnimal(dog) // works
    val animal: Animal = Animal()
    val animalDog: Dog = animal // compilation error
    petDog(animal) // compilation error
}
```

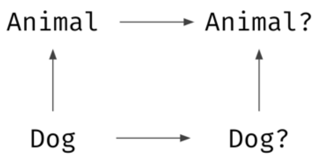
Why? Because there is a concrete relationship between these types: `Dog` is a subtype of `Animal`. By rule, when A is a subtype of B, we can use A where B is expected. We might also say that `Animal` is a supertype of `Dog`, and a subtype can be used where a supertype is expected.



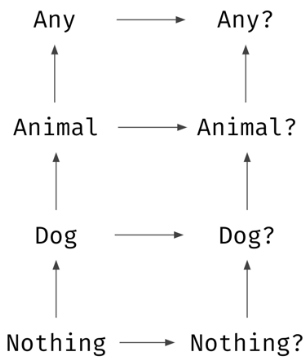
There is also a relationship between nullable and non-nullable types. A non-nullable can be used wherever a nullable is expected.

```
fun petDogIfPresent(dog: Dog?) {}
fun petDog(dog: Dog) {}

fun main() {
    val dog: Dog = Dog()
    val dogNullable: Dog? = dog
    petDogIfPresent(dog) // works
    petDogIfPresent(dogNullable) // works
    petDog(dog) // works
    petDog(dogNullable) // compilation error
}
```

This is because the non-nullable variant of each type is a subtype of the nullable variant.



The superclass of all the classes in Kotlin is `Any`, which is similar to `Object` in Java. The supertype of all the types is not `Any`, it is `Any?`. `Any` is a supertype of all non-nullable types. We also have something that is not present in Java and most other mainstream languages: the subtype of all the types, which is called `Nothing`. We will talk about it soon.



`Any` is only a supertype of non-nullable types. So, wherever `Any` is expected, nullable types will not be accepted. This fact is also used to set a type parameter's upper boundary to accept only non-nullable types[29].

```
fun <T : Any> String.parseJson(): T = ...
```

`Unit` does not have any special place in the type hierarchy. It is just an object declaration that is used when a function does not specify a result type.

```
object Unit {
    override fun toString() = "kotlin.Unit"
}
```

Let's talk about a concept that has a very special place in the typing hierarchy: let's talk about `Nothing`.

The subtype of all the types: Nothing

`Nothing` is a subtype of all the types in Kotlin. If we had an instance of this type, it could be used instead of everything else (like a Joker in the card game Rummy). It's no wonder that such an instance does not exist. `Nothing` is an empty type (also known as a bottom type, zero type, uninhabited type, or never type), which means it has no values. It is literally impossible to make an instance of type `Nothing`, but this type is still really useful. I will tell you more: some functions declare `Nothing` as their result type. You've likely used such functions many times already. What functions are those? They declare `Nothing` as a result type, but they cannot return it because this type has no instances. But what can these functions do? Three things: they either need to run forever, end the program, or throw an exception. In all cases, they never return, so the `Nothing` type is not only valid but also really useful.

```
fun runForever(): Nothing {
    while (true) {
        // no-op
    }
}

fun endProgram(): Nothing {
    exitProcess(0)
}
```

[29]I will explain type parameters' upper boundaries in the chapter *Generics*.

```
fun fail(): Nothing {
    throw Error("Some error")
}
```

I have never found a good use case for a function that runs forever, and ending a program is not very common, but we often use functions that throw exceptions. Who hasn't ever used `TODO()`? This function throws a `NotImplementedError` exception. There is also the `error` function from the standard library, which throws an `IllegalStateException`.

```
inline fun TODO(): Nothing = throw NotImplementedError()

inline fun error(message: Any): Nothing =
    throw IllegalStateException(message.toString())
```

`TODO` is used as a placeholder in a place where we plan to implement some code.

```
fun fib(n: Int): Int = TODO()
```

`error` is used to signal an illegal situation:

```
fun get(): T = when {
    left != null -> left
    right != null -> right
    else -> error("Must have either left or right")
}
```

This result type is significant. Let's say that you have an if-condition that returns either `Int` or `Nothing`. What should the inferred type be? The closest supertype of both `Int` and `Nothing` is `Int`. This is why the inferred type will be `Int`.

```
// the inferred type of answer is Int
val answer = if (timeHasPassed) 42 else TODO()
```

The same rule applies when we use the Elvis operator, a when-expression, etc. In the example below, the type of both `name` and `fullName` is `String` because both `fail` and `error` declare `Nothing` as their result type. This is a huge convenience.

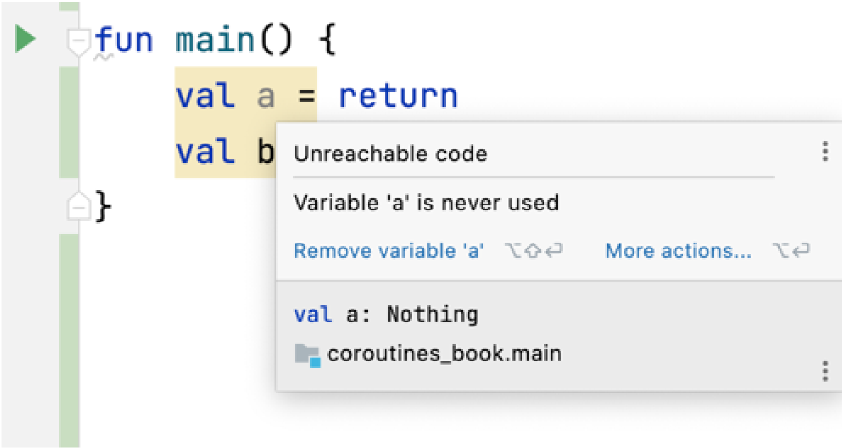
```
fun processPerson(person: Person?) {
    // the inferred type of name is String
    val name = person?.name ?: fail()
    // the inferred type of fullName is String
    val fullName = when {
        !person.middleName.isNullOrBlank() ->
            "$name ${person.middleName} ${person.surname}"
        !person.surname.isNullOrBlank() ->
            "$name ${person.surname}"
        else ->
            error("Person must have a surname")
    }
    // ...
}
```

The result type from return and throw

I will start this subchapter with something strange: did you know that you can place `return` or `throw` on the right side of a variable assignment?

```
fun main() {
    val a = return
    val b = throw Error()
}
```

This doesn't make any sense as both `return` and `throw` end the function, so we will never assign anything to such variables (like `a` and `b` in the example above). This assignment is an unreachable piece of code. In Kotlin, it just causes a warning.



The code above is correct from the language point of view because both `return` and `throw` are expressions, which means they declare a result type. This type is `Nothing`.

```
fun main() {
    val a: Nothing = return
    val b: Nothing = throw Error()
}
```

This explains why we can place `return` or `throw` on the right side of the Elvis operator or in a when-expression.

```
fun processPerson(person: Person?) {
    val name = person?.name ?: return
    val fullName = when {
        !person.middleName.isNullOrBlank() ->
            "$name ${person.middleName} ${person.surname}"
        !person.surname.isNullOrBlank() ->
            "$name ${person.surname}"
        else -> return
    }
    // ...
}
```

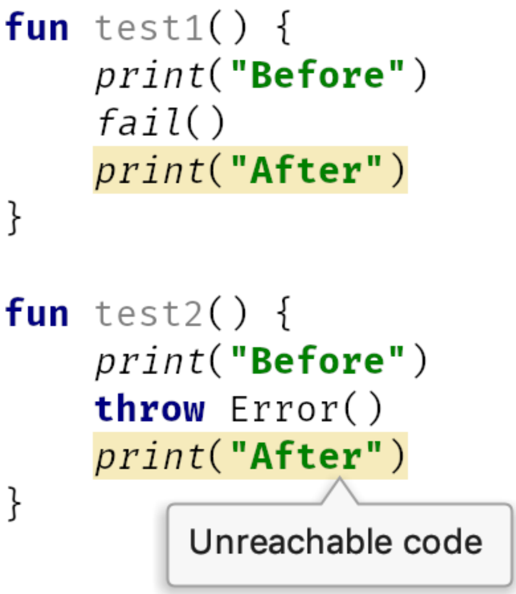
```kotlin
fun processPerson(person: Person?) {
    val name = person?.name ?: throw Error("Name is required")
    val fullName = when {
        !person.middleName.isNullOrBlank() ->
            "$name ${person.middleName} ${person.surname}"
        !person.surname.isNullOrBlank() ->
            "$name ${person.surname}"
        else -> throw Error("Surname is required")
    }
    // ...
}
```

Both `return` and `throw` declare `Nothing` as their result type. As a consequence, Kotlin will infer `String` as the type of both `name` and `fullName` because `String` is the closest supertype of both `String` and `Nothing`.

So, now you can say that you know `Nothing`. Just like John Snow.

When is some code not reachable?

When an element declares `Nothing` as a return type, it means that everything after its call is not reachable. This is reasonable: there are no instances of `Nothing`, so it cannot be returned. This means a statement that declares `Nothing` as its result type will never complete in a normal way, so the next statements are not reachable. This is why everything after either `fail` or `throw` will be unreachable.



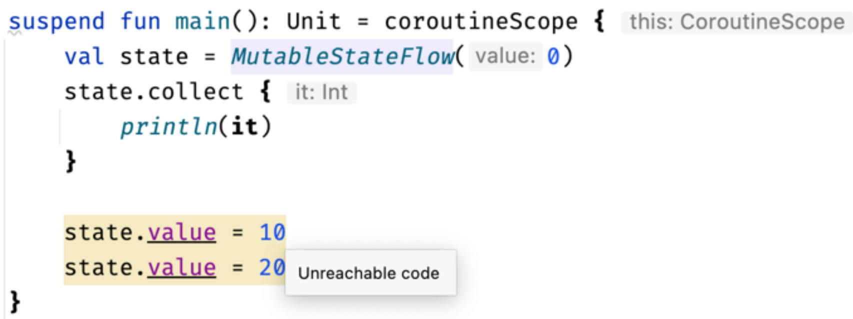
It's the same with `return`, `TODO`, `error`, etc. If a non-optional expression declares `Nothing` as its result type, everything after that is unreachable. This is a simple rule, but it's useful for the compiler. It's also useful for us since it gives us more possibilities. Thanks to this rule, we can use `TODO()` in a function instead of returning a value. Anything that declares `Nothing` as a result type ends the function (or runs forever), so this function will not end without returning or throwing first.

```
fun fizzBuzz(): String {
    TODO()
}
```

I would like to end this topic with a more advanced example that comes from the Kotlin Coroutines library. There is a `MutableStateFlow` class, which represents a mutable value whose state changes can be observed using the `collect` method. The thing is that `collect` suspends the current coroutine until whatever it observes is closed, but a StateFlow cannot be closed. This is why this `collect` function declares `Nothing` as its result type.

```
public interface SharedFlow<out T> : Flow<T> {
    public val replayCache: List<T>
    override suspend fun collect(
        collector: FlowCollector<T>
    ): Nothing
}
```

That is very useful for developers who are not aware of how `collect` works. Thanks to the result type, IntelliJ informs them that the code they place after `collect` is unreachable.



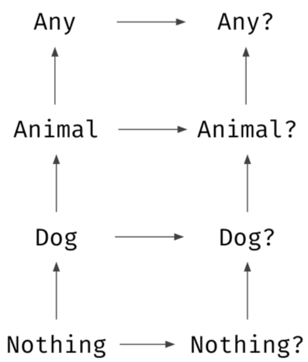
SharedFlow cannot be closed, so its `collect` function will never return, therefore it declares `Nothing` as its result type.

The type of null

Let's see another peculiar thing. Did you know that you can assign `null` to a variable without setting an explicit type? What's more, such a variable can be used wherever `null` is accepted.

```
fun main() {
    val n = null
    val i: Int? = n
    val d: Double? = n
    val str: String? = n
}
```

This means that `null` has its type, which is a subtype of all nullable types. Take a look at the type hierarchy and guess what type this is.



I hope you guessed that the type of `null` is `Nothing?`. Now think about the inferred type of `a` and `b` in the example below.

```
val a = if (predicate) "A" else null

val b = when {
    predicate2 -> "B"
    predicate3 -> "C"
    else -> null
}
```

In the if-expression, we search for the closest supertype of the types from both branches. The closest supertype of `String` and `Nothing?` is `String?`. The same is true about the when-expression: the closest supertype of `String`, `String`, and `Nothing?` is `String?`. Everything makes sense.

For the same reason, whenever we require `String?`, we can pass either `String` or `null`, whose type is `Nothing?`. This is clear when you take a look at the type hierarchy. `String` and `Nothing?` are the only non-empty subtypes of `String?`.

SOCRATES KNOWS, THAT HE KNOWS NOTHING

Summary

In this chapter, we've learned the following:

- A class is a template for creating objects. A type defines expectations and functionalities.
- Every class generates a nullable and a non-nullable type.
- A nullable type is a supertype of the non-nullable variant of this type.
- The supertype of all types is `Any?`.
- The supertype of non-nullable types is `Any`.
- The subtype of all types is `Nothing`.
- When a function declares `Nothing` as a return type, this means that it will throw an error or run infinitely.
- Both `throw` and `return` declare `Nothing` as their result type.
- The Kotlin compiler understands that when an expression declares `Nothing` as a result type, everything after that is unreachable.
- The type of `null` is `Nothing?`, which is the subtype of all nullable types.

In the next chapter, we are going to discuss generics, and we'll see how they are important for our type system.

Exercise: The closest supertype of types

What is the closest supertype of the following types?
- `Int` and `Double`
- `Double` and `Number`
- `String` and `Nothing`
- `Float` and `Double?`
- `String` and `Float`
- `Char` and `Nothing?`
- `Nothing` and `Any`
- `Nothing?` and `Any`
- `Char?` and `Nothing?`
- `Nothing?` and `Any?`

Generics

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Generic functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Generic classes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Generic classes and nullability

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Generic interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Type parameters and inheritance

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Type erasure

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Generic constraints

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Star projection

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Underscore operator for type arguments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise: Stock

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Final words

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Final Project: Workout manager

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Exercise solutions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Basic values operations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Using when

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Pretty time display

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Person details display

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Range Operations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: User Information Processor

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Implementing the Product class

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: GUI View Hierarchy Simulation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Data class practice

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Pizza factory

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Catching exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Days of the week enum

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Conversion and measurement unit creation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Inventory management

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Money operations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: The closest supertype of types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Stock

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.

Solution: Workout manager

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/kotlin_developers.