

# Kotlin

## IN ACTION

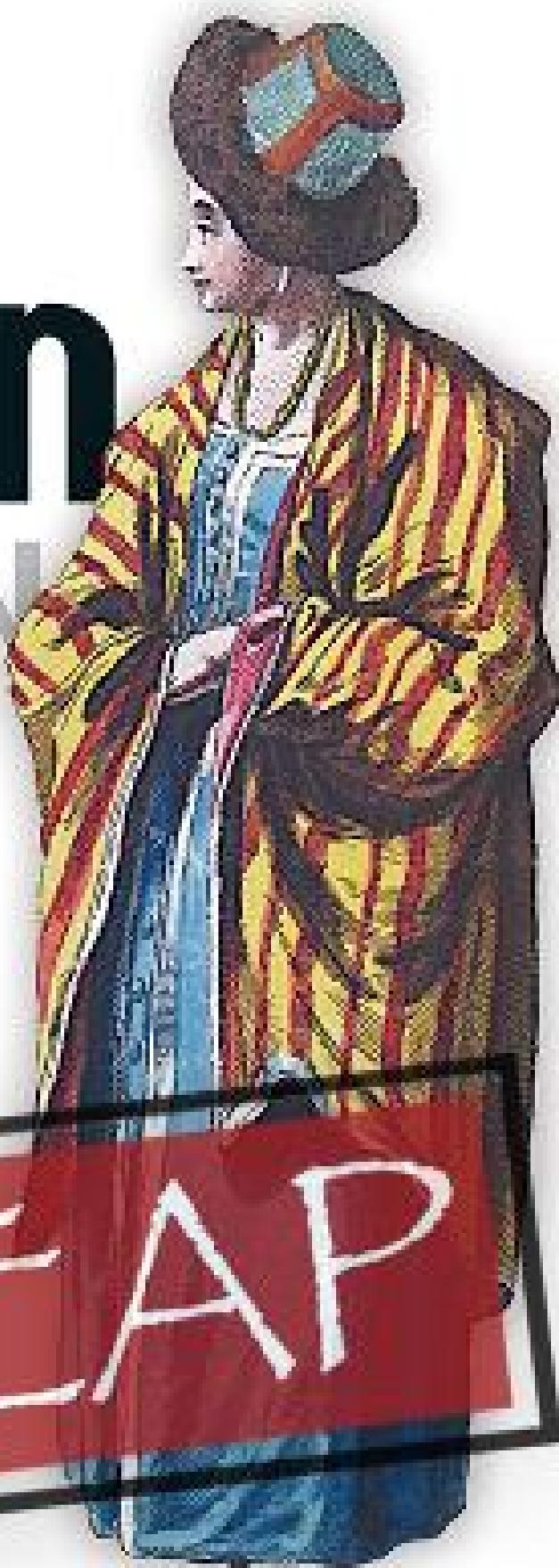
SECOND EDITION

Svetlana Isakova  
Roman Elizarov  
Sebastian Aigner  
Dmitry Jemerov

MEAP

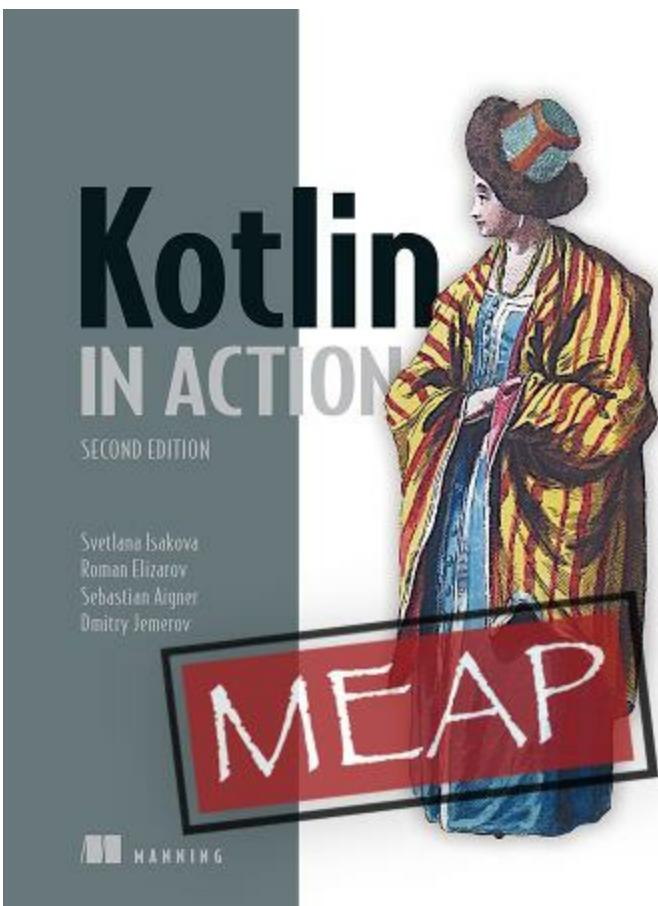


HANNING



# Kotlin in Action, Second Edition MEAP V09

1. [MEAP VERSION 9](#)
2. [Welcome](#)
3. [1 Kotlin: what and why](#)
4. [2 Kotlin basics](#)
5. [3 Defining and calling functions](#)
6. [4 Classes, objects, and interfaces](#)
7. [5 Programming with lambdas](#)
8. [6 Working with collections and sequences](#)
9. [7 Working with nullable values](#)
10. [8 Basic types, collections, and arrays](#)
11. [9 Operator overloading and other conventions](#)
12. [10 Higher-order functions: lambdas as parameters and return values](#)
13. [Appendix A. Building Kotlin projects](#)
14. [Appendix B. Documenting Kotlin code](#)



**MEAP VERSION 9**



# Welcome

Thank you for purchasing the MEAP edition of *Kotlin in Action, Second Edition*. We hope that you'll find this content useful and help us make the final version even better!

Since the first edition of *Kotlin in Action*, Kotlin has evolved a lot. Not only the language but the whole ecosystem around it has changed. Kotlin has received many updates, new features, and paradigms, as well as new library functionality. The whole topic of coroutines wasn't even a thing when we were working on the first edition, and now they're the default way to write asynchronous and concurrent code in Kotlin. Many new libraries and frameworks have appeared, and many existing ones have added support for the language.

With the second edition of *Kotlin in Action*, we want to continue to give experienced developers a comprehensive introduction to the language. We're updating it with the latest features, including concurrency paradigms with coroutines, something that has been heavily requested by the community.

Similar to the first edition, this book is written for developers who want to become fluent in Kotlin and already have some experience with programming using other languages. While working on the first edition, we explicitly targeted Java developers but received a lot of feedback that it was also suitable for developers from other backgrounds, like .NET. That's why knowledge of Java isn't a prerequisite this time, but of course, Kotlin will be especially easy to learn for those who know Java. We'll continue to include the intricacies of interoperability with Java as side-notes.

This book is also suitable for Kotlin developers who have a desire to deepen their knowledge of the language and use it to its full potential, both for application and library development.

Please post your questions and comments about the content in the [liveBook discussion forum](#). It will help us see what works well and work out which

parts can be improved!

— Svetlana Isakova, Roman Elizarov, Sebastian Aigner, and Dmitry Jemerov.

**In this book**

[MEAP VERSION 9](#) [About this MEAP](#) [Welcome](#) [Brief Table of Contents](#) [1](#)  
[Kotlin: what and why](#) [2](#) [Kotlin basics](#) [3](#) [Defining and calling functions](#) [4](#)  
[Classes, objects, and interfaces](#) [5](#) [Programming with lambdas](#) [6](#) [Working with collections and sequences](#) [7](#) [Working with nullable values](#) [8](#) [Basic types, collections, and arrays](#) [9](#) [Operator overloading and other conventions](#) [10](#)  
[Higher-order functions: lambdas as parameters and return values](#)  
[Appendix A. Building Kotlin projects](#) [Appendix B. Documenting Kotlin code](#)

# 1 Kotlin: what and why

## This chapter covers

- A basic demonstration of Kotlin
- The main traits of the Kotlin language
- Possibilities for server-side and Android development
- A glimpse into Kotlin Multiplatform
- What distinguishes Kotlin from other languages
- Writing and running code in Kotlin

What is Kotlin all about? It's a modern programming language on the Java Virtual Machine (JVM) and beyond. It's a general-purpose language, concise, safe and pragmatic. Independent programmers, small software shops, and large enterprises all have embraced Kotlin: millions of developers are now using it to write mobile apps, build server-side applications, create desktop software, and more.

Kotlin started as a "better Java": a language with improved developer ergonomics, that prevents common categories of errors, and that embraces modern language design paradigms, all while keeping the ability to be used everywhere Java was used. Over the last decade, Kotlin has managed to prove itself to be a good pragmatic, fit for many types of developers, projects, and platforms. Android is now Kotlin-first, meaning most of the Android development is done in Kotlin. For server-side development, Kotlin makes a strong alternative to Java, with native and well-documented Kotlin support in prevalent frameworks like Spring, and with pure-Kotlin frameworks exploiting the full potential of the language, like Ktor.

Kotlin combines ideas from existing languages that work well, but also brings innovative approaches, such as coroutines for asynchronous programming. Despite being started with JVM-only focus, Kotlin grew significantly beyond that, providing more "targets" to run on, including technology to create cross-platform solutions.

In this chapter, we'll take a detailed look at Kotlin's main traits.

## 1.1 A taste of Kotlin

Let's start with a small example to demonstrate what Kotlin looks like. Even in this first, short code snippet, you can see a lot of interesting features and concepts in Kotlin that will all be discussed in detail later throughout the book:

- Defining a Person data class with properties without the need to specify a body.
- Declaring read-only properties (name and age) with the val keyword
- Providing default values for arguments
- Explicit work with nullable values (Int?) in the type system, avoiding the "Billion Dollar Mistake" of NullPointerExceptions
- Top-level function definitions without the need of nesting them inside classes
- Named arguments when invoking functions and constructors
- Using trailing commas
- Using collection operations with lambda expressions
- It illustrates how collection functions like maxByOrNull help find the oldest person in the list.
- Providing fallback values when a variable is null via the Elvis operator (?:).
- Using string templates as an alternative to manual concatenation
- Using autogenerated functions for data classes, such as `toString`

The code is explained briefly, but please don't worry if something isn't clear right away. We will take plenty of time to discuss each and every detail of this code snippet throughout the book, so you'll be able to confidently write code just like this yourself.

**Listing 1.1. An early taste of Kotlin**

```
data class Person( #1
    val name: String, #2
    val age: Int? = null #3
)
```

```

fun main() { #4
    val persons = listOf(
        Person("Alice", age = 29), #5
        Person("Bob"), #6
    )
    val oldest = persons.maxBy { #7
        it.age ?: 0 #8
    }
    println("The oldest is: $oldest") #9
}

// The oldest is: Person(name=Alice, age=29) #10

```

Our first Kotlin code snippet demonstrates how to create a collection in Kotlin, fill it with some Person objects, and then find the oldest person in the collection, using default values where no age is specified.

When creating the list of people, it omits Bob's age, so null is used as a default value. To find the oldest person in the list, the `maxBy` function is used. The lambda expression passed to the function takes one parameter, and `it` is used as the default name of that parameter. The *Elvis operator* (`? :`) returns zero if age is null. Because Bob's age isn't specified, the Elvis operator replaces it with zero, so Alice wins the prize for being the oldest person.

You can also try to run this example on your own. The easiest option to do is to use the online playground at <https://play.kotlinlang.org/>. Type in the example and click the Run button, and the code will be executed.

Do you like what you've seen? Read on to learn more and become a Kotlin expert. We hope that soon you'll see such code in your own projects, not only in this book.

## 1.2 Kotlin's primary traits

Kotlin is a multi-paradigm language. It is statically typed, meaning many errors can be caught at compile time instead of at runtime. It combines ideas from object-oriented and functional languages, which helps you write elegant code and make use of additional powerful abstractions. It provides a powerful way to write asynchronous code, which is important in all development areas.

Just based on these short descriptions, you maybe already have an intuitive idea of the type of language Kotlin is. Let's look at these key attributes in more detail. First, let's see what kinds of applications you can build with Kotlin.

### **1.2.1 Kotlin use-cases: Android, server-side, anywhere Java runs, and more**

Kotlin's target is quite broad. The language doesn't focus on a single problem domain or address a single type of challenge faced by software developers today. Instead, it provides across-the-board productivity improvements for all tasks that come up during the development process and aims for an excellent level of integration with libraries that support specific domains or programming paradigms.

The most common areas to use Kotlin are:

- Building mobile applications that run on Android devices
- Building server-side code (typically, backends of web applications)

The initial goal of Kotlin was to provide a more concise, more productive, safer alternative to Java that's suitable in all contexts where Java can be used. That includes a broad variety of environments, from running small edge devices to the largest data centers. In all these use-case Kotlin fits perfectly, and developers can do their job with less code and fewer annoyances along the way.

But Kotlin works in other contexts as well. You can create cross-platforms apps with Kotlin Multiplatform Mobile, or run Kotlin in browser using Kotlin/JS. This book is focused mainly on the language itself and intricacies of the JVM target. You can find the extensive information about other Kotlin applications on the Kotlin website: <https://kotlinlang.org/>.

Next, let's look at the key qualities of Kotlin as a programming language.

### **1.2.2 Static typing makes Kotlin performant, reliable, and maintainable**

*Statically typed* programming languages come with a number of advantages, such as performance, reliability, maintainability, and tool support. The key point behind a statically typed language is that the type of every expression in a program is known at compile time. Kotlin is a statically typed programming language: The Kotlin compiler can validate that the methods and fields you’re trying to access on an object actually exist. This helps eliminate an entire class of bugs—rather than crash at runtime, if a field is missing or the return type of a function call isn’t as expected, you will already see these problems at compile time, allowing you to fix them earlier in the development cycle.

Following are some benefits of static typing:

- *Performance*—Calling methods is faster because there’s no need to figure out at runtime which method needs to be called.
- *Reliability*—The compiler uses types to verify the consistency of the program, so there are fewer chances for crashes at runtime.
- *Maintainability*—Working with unfamiliar code is easier because you can see what kind of types the code is working with.
- *Tool support*—Static typing enables reliable refactorings, precise code completion, and other IDE features.

This is in contrast to *dynamically typed* programming languages, like Python or JavaScript. Those languages let you define variables and functions that can store or return data of any type and resolve the method and field references at runtime. This allows for shorter code and greater flexibility in creating data structures. But the downside is that problems like misspelled names or invalid parameters passed to functions can’t be detected during compilation and can lead to runtime errors.

While the type of every expression in your program needs to be *known* at compile time, Kotlin doesn’t require you to *specify* the type of every variable explicitly in your source code. In many cases, the type of a variable can automatically be determined from the context, allowing you to omit the type declaration. Here’s the simplest possible example of this:

```
val x: Int = 1 #1  
val y = 1 #2
```

You’re declaring a variable, and because it’s initialized with an integer value, Kotlin automatically determines that its type is `Int`. The ability of the compiler to determine types from context is called *type inference*. Type inference in Kotlin means most of the extra verbosity associated with static typing disappears, because you don’t need to declare types explicitly.

If you look at the specifics of Kotlin’s type system, you’ll find many familiar concepts from other object-oriented programming languages. Classes and interfaces, for example, work as you may already expect from other experience. And if you happen to be a Java developer, your knowledge transfers especially easily to Kotlin, including topics like generics.

Something that may stand out to you is Kotlin’s support for *nullable types*, which lets you write more reliable programs by detecting possible `null` pointer exceptions at compile time, rather than experiencing them in the form of crashes at runtime. We’ll come back to nullable types later in [7](#) and discuss them in detail in [7](#), where we’ll also contrast them with other approaches for `null` values you might be familiar with.

Kotlin’s type system also has first-class support for *function types*. To see what this is about, let’s look at the main ideas of functional programming and see how it’s supported in Kotlin.

### 1.2.3 Combining functional and object-oriented makes Kotlin safe and flexible

As a multi-paradigm programming language, Kotlin combines the *object-oriented* approach with the *functional programming* style. The key concepts of functional programming are as follows:

- *First-class functions*—You work with functions (pieces of behavior) as values. You can store them in variables, pass them as parameters, or return them from other functions.
- *Immutability*—You work with immutable objects, which guarantees that their state can’t change after their creation.
- *No side effects*—You write *pure functions*—functions that return the same result given the same inputs and don’t modify the state of other

objects or interact with the outside world.

What benefits can you gain from writing code in the functional style? First, *conciseness*. Functional code can be more elegant and succinct when compared to its *imperative* counterpart: Instead of mutating variables and relying on loops and conditional branching, working with functions as values gives you much more power of abstraction.

Applying a functional programming style also lets you avoid duplication in your code. If you have similar code fragments that implement a similar task, but differ in some smaller details, you can easily extract the common part of the logic into a function, and pass the differing parts as arguments. Those arguments might themselves be functions. In Kotlin, you can express those using a concise syntax for lambda expressions.

The second benefit of functional code is *safe concurrency*. One of the biggest sources of errors in multithreaded programs is modification of the same data from multiple threads without proper synchronization. If you use immutable data structures and pure functions, you can be sure that such unsafe modifications won't happen, and you don't need to come up with complicated synchronization schemes.

Finally, functional programming means *easier testing*. Functions without side effects can be tested in isolation without requiring a lot of setup code to construct the entire environment that they depend on. When your functions don't interact with the outside world, you'll also have an easier time reasoning about your code and validating its behaviour without having to keep a larger, complex system in your head at all times.

Generally speaking, a functional programming style can be used with many programming languages, and many parts of it are advocated as good programming style. But not all languages provide the syntactic and library support required to use it effortlessly. Kotlin has a rich set of features to support functional programming from the get-go. These include the following:

- *Function types*, allowing functions to receive other functions as arguments or return other functions

- *Lambda expressions*, letting you pass around blocks of code with minimum boilerplate
- *Member references*, allowing you to use functions as values, for instance, pass them as arguments
- *Data classes*, providing a concise syntax for creating classes that can hold immutable data
- A rich set of *APIs* in the standard library for working with objects and collections in the functional style

The following snippet demonstrates a chain of actions to be performed with an input sequence. Having a given sequence of messages, the code finds "all senders of non-empty unread messages sorted by their names":

```
messages
    .filter { it.body.isNotBlank() && !it.isRead }
    .map(Message::sender)
    .distinct()
    .sortedBy(Sender::name)
```

The Kotlin standard library defines functions like `filter`, `map` and `sortedBy` for you to use. The Kotlin language supports lambda expressions and member references (like `Message::sender`), so that the arguments passed to these functions are really concise.

When writing code in Kotlin, you can combine both object-oriented and functional approaches and use the tools that are most appropriate for the problem you're solving: You get the full power of functional-style programming in Kotlin, and when you need it, you can work with mutable data and write functions with side effects, all without jumping through extra hoops. And, of course, working with frameworks that are based on interfaces and class hierarchies is just as easy as you would expect it to be.

## 1.2.4 Concurrent and asynchronous code becomes natural and structured with coroutines

Whether you're building an application running on a server, a desktop machine, or a mobile phone, *concurrency*, running multiple pieces of your code at the same time, is a topic that's almost unavoidable.

User interfaces need to remain responsive while long-running computations are running in the background. When interacting with services on the internet, applications often need to make more than one request at a time. Likewise, server-side applications are expected to keep serving incoming requests, even when a single request is taking much longer than usual. All of these applications need to operate *concurrently*, working on more than one thing at a time.

There have been many approaches to concurrency, from threads to callbacks, futures and promises to reactive extensions, and more.

Kotlin approaches the problem of concurrent and asynchronous programming using *suspendable computations* called *coroutines*, where code can suspend its execution, and resume its work at a later point.

In this example, you define a function `processUser` making three network calls by calling `authenticate`, `loadUserData`, and `loadImage`:

```
suspend fun processUser(credentials: Credentials) {
    val user = authenticate(credentials) #1
    val data = loadUserData(user) #2
    val profilePicture = loadImage(data.imageID) #3
    // ...
}

suspend fun authenticate(c: Credentials): User { /* ... */ } #4
suspend fun loadUserData(u: User): Data { /* ... */ }
suspend fun loadImage(id: Int): Image { /* ... */ }
```

A network call may take arbitrarily long. When performing each network request, the execution of the `processUser` function is *suspended* while waiting for the result. However, the thread this code is running on (and, by extension, the application itself), isn't *blocked*: While waiting for the result of `processUser`, it can do other tasks in the meantime, such as responding to user inputs.

You won't be able to write this code sequentially in an imperative fashion, one call after another, without blocking the underlying threads. With callbacks or reactive streams such simple consecutive logic becomes much more complicated.

In the following example, you load two images concurrently, then wait for the loading to be completed and return the overlay as the result:

```
suspend fun loadAndOverlay(first: String, second: String): Image
    coroutineScope {
        val firstDeferred = async { loadImage(first) } #1
        val secondDeferred = async { loadImage(second) } #2
        overlay(firstDeferred.await(), secondDeferred.await()) #3
    }
```

*Structured concurrency* helps you manage the lifetime of your coroutines. In this example, two loading processes are started in a structured way (from the same *coroutine scope*). It guarantees that if one loading fails, the second one gets automatically cancelled.

Coroutines are also a very lightweight abstraction, meaning you can launch millions of concurrent jobs without significant performance penalties. Together with abstractions like *cold* and *hot flows*, and *channels* that facilitate communication, Kotlin coroutines become a powerful tool for building concurrent applications.

The entire third part of this book will be dedicated to learning ins and outs of coroutines, and understanding how you can best apply them for your use cases.

### **1.2.5 Kotlin can be used for any purpose: it's free, open source, and open to contributions**

The Kotlin language, including the compiler, libraries, and all related tooling, is entirely open source and free to use for any purpose. It's available under the Apache 2 license; development happens in the open on GitHub (<http://github.com/jetbrains/kotlin>). There are many ways to contribute to the development of Kotlin and its community:

- The project welcomes code contributions for new features and fixes around the Kotlin compiler and its associated tooling.
- By providing bug reports and feedback you can help improve the experience when developing with Kotlin for everyone.
- Potential new language features are discussed at length in the Kotlin

community, and input from Kotlin developers like yourself plays a big role in driving the language forward and evolving it.

You also have a choice of multiple open source IDEs for developing your Kotlin applications: IntelliJ IDEA Community Edition and Android Studio are fully supported. (Of course, IntelliJ IDEA Ultimate works as well.)

Now that you understand what kind of language Kotlin is, let's see how the benefits of Kotlin work in specific practical applications.

## 1.3 Areas in which Kotlin is often used

As we mentioned earlier, two of the main areas where Kotlin is being used are server-side and Android development. Let's look at those areas in turn and see why Kotlin is a good fit for them.

### 1.3.1 Powering backends: server-side development with Kotlin

Server-side programming is a fairly broad concept. It encompasses all the following types of applications and much more:

- Web applications that return HTML pages to a browser
- Backends for mobile or single-page applications that expose a JSON API over HTTP
- Microservices that communicate with other microservices over an RPC protocol or message bus

Developers have been building these kinds of applications on the JVM for many years and have accumulated a huge stack of frameworks and technologies to help build them. Such applications usually aren't developed in isolation or started from scratch. There's almost always an existing system that is being extended, improved, or replaced, and new code has to integrate with existing parts of the system, which may have been written many years ago.

In this environment especially, Kotlin profits from its seamless interoperability with existing Java code. Regardless of whether you're

writing a new component or migrating the code of an existing service to Kotlin, Kotlin will fit right in. You won't run into problems when you need to extend Java classes in Kotlin or annotate the methods and fields of a class in a certain way. And the benefit is that the code of your system will be more compact, more reliable, and easier to maintain.

Another big advantage of using Kotlin is better reliability for your application. Kotlin's type system, with its precise tracking of `null` values, makes the problem of `null` pointer exceptions much less pressing. Most of the code that would lead to a `NullPointerException` at runtime in Java fails to compile in Kotlin, ensuring that you fix the error before the application gets to the production environment.

Modern frameworks, such as Spring (<https://spring.io/>), provide first-class support for Kotlin out of the box. Beyond the seamless interoperability, these frameworks include additional extensions and make use of techniques which make it feel as if they were designed for Kotlin in the first place.

In this example, you're defining a simple Spring Boot application, that serves a list of `Greeting` objects, consisting of an ID and some text, as JSON via HTTP. Concepts from the Spring framework transfer directly to Kotlin: you use the same annotations (`@SpringBootApplication`, `@RestController`, `@GetMapping`) as you would when using Java:

#### **Listing 1.2. Writing Spring Boot applications in Kotlin**

```
@SpringBootApplication #1
class DemoApplication

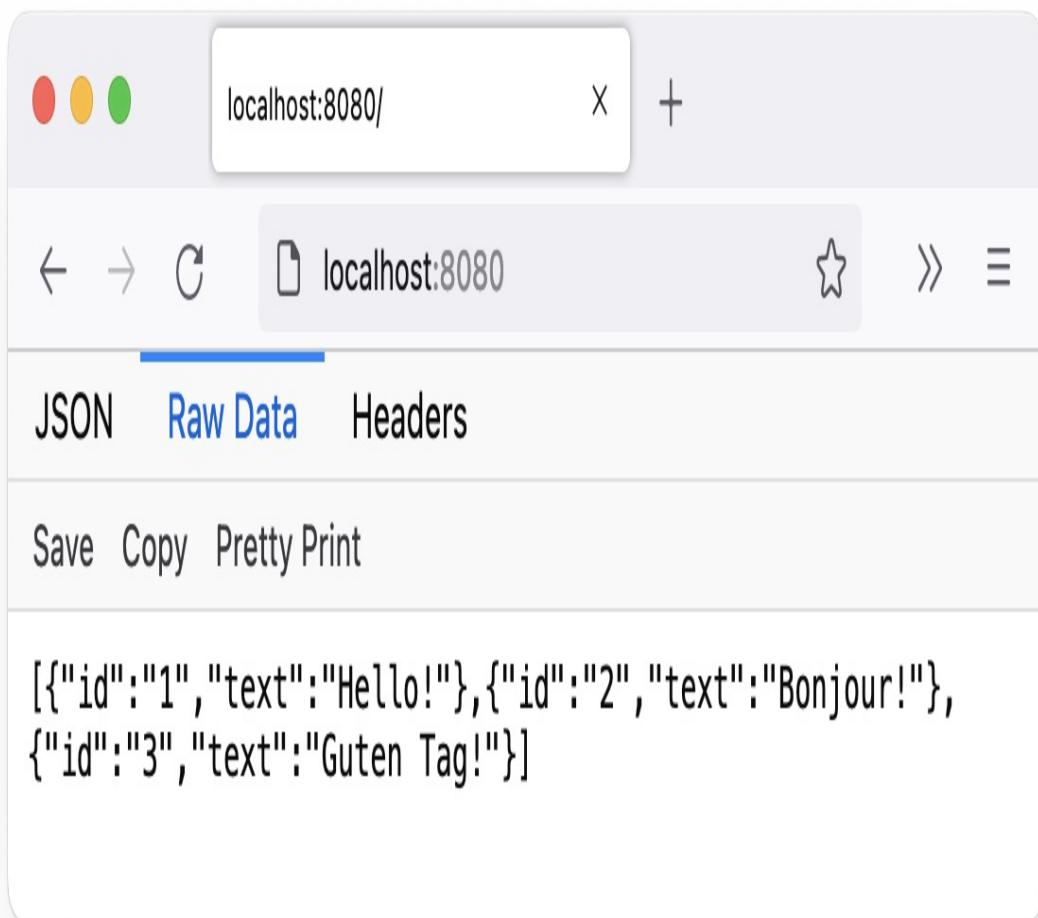
fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}

@RestController
class GreetingResource {
    @GetMapping
    fun index(): List<Greeting> = listOf( #2
        Greeting(1, "Hello!"),
        Greeting(2, "Bonjour!"),
        Greeting(3, "Guten Tag!"),
    )
}
```

```
}

data class Greeting(val id: Int, val text: String) #2
```

**Figure 1.1.** By combining Kotlin with industry-proven frameworks like Spring, writing an application that serves JSON via HTTP only takes two dozen lines of code.



Check the Kotlin or Spring websites to find more information about using

Spring with Kotlin (<https://kotlinlang.org/docs/jvm-spring-boot-restful.html>).

Kotlin also enjoys an ever-growing ecosystem of its own libraries, including server-side frameworks. As an example, Ktor (<https://ktor.io/>) is a connected applications framework for Kotlin built by JetBrains. It powers products like JetBrains Space (<https://jetbrains.space>) and Toolbox (<https://jetbrains.com/toolbox>), and has been adopted by companies like Adobe.

As a Kotlin framework, Ktor makes full use of the capabilities of the language. For example, it defines a custom *domain-specific language* (DSL) to declare how HTTP requests are routed through the application. Rather than configuring your application using annotations or XML files, you can use a DSL from Ktor to configure the routing of your server-side application, with constructs that look like they are a part of the Kotlin language, but are completely custom for the framework – something you’ll learn how to do yourself in **Chapter 13**.

In this example, you’re defining three routes, `/world`, `/greet`, and `/greet/{entityId}`, using the `get`, `post`, and `route` DSL constructs from Ktor:

**Listing 1.3. A Ktor app uses a DSL to route HTTP requests. While the DSL looks like it is part of the language, it is defined entirely by the framework, without the requirement for external configuration files or modifying the compiler.**

```
fun main() {
    embeddedServer(Netty, port = 8000) {
        routing { #1
            get ("/world") { #2
                call.respondText("Hello, world!")
            }
            route("/greet") {
                get { /* . . . */ }
                post("/{entityId}") { /* . . . */ } #3
            }
        }
    }.start(wait = true)
}
```

DSLs flexibly combine Kotlin language features, and are often used for

configuration, the construction of complex objects, or *object-relational mapping* (ORM) tasks, translating objects into their database representation and vice versa.

Other Kotlin server-side frameworks like http4k (<https://http4k.org/>) strongly embrace the functional nature of Kotlin code, and provide simple and uniform abstractions for requests and responses. In short: Whether you're looking to use a battle-tested industry standard framework for your next large project, or need a lightweight framework for your next microservice, you can rest assured there's a framework waiting for you Kotlin's extensive ecosystem.

### 1.3.2 Mobile Development: Android is Kotlin-first

The most-used mobile operating system in the world, Android, started officially supporting Kotlin as a language for building apps in 2017. Only two years later, in 2019, after a lot of positive feedback from developers, Android became Kotlin-first, making Kotlin the default choice for new apps. Since then, Google's development tools, their Jetpack libraries (<https://developer.android.com/jetpack>), samples, documentation, and training content all primarily focus on Kotlin.

Kotlin is a good fit for mobile apps: these types of applications usually need to be delivered quickly while ensuring reliable operation on a large variety of devices. Kotlin's language features turn Android development into a much more productive and pleasant experience. Common development tasks can be accomplished with much less code. The Android KTX library (<https://developer.android.com/kotlin/ktx>), built by the Android team, improves your experience even further by adding Kotlin-friendly adapters around many standard Android APIs.

Google's Jetpack Compose toolkit (<https://developer.android.com/jetpack/compose>) for building native user interfaces for Android is also designed for Kotlin from the ground up. It embraces Kotlin's language features, and gives you the ability to write less, simpler, and easier-to-maintain code when building the UI of your mobile applications.

Here's an example of Jetpack Compose, just to give you a taste of what Android development with Kotlin feels like. The following code shows the message and expands or hides the details on click:

```
@Composable
fun MessageCard(modifier: Modifier, message: Message) {
    var isExpanded by remember { mutableStateOf(false) } #1
    Column(modifier.clickable { isExpanded = !isExpanded }) { #2
        Text(message.body) #3
        if (isExpanded) { #4
            MessageDetails(message) #5
        }
    }
}

@Composable
fun MessageDetails(message: Message) { ... }
```

You can write the whole UI in Kotlin, and use the regular Kotlin syntax like if-expressions or loops. In this example, we show a UI element representing additional details only if the user clicked on the card to get the expanded view. With Kotlin, you can extract custom logic of representing different parts of the UI into functions, like `MessageDetails`, having more elegant code as a result.

Embracing Kotlin on Android also means more reliable code, fewer `NullPointerExceptions`, and fewer messages that read "Unfortunately, process has stopped". As an example, Google themselves managed to reduce the number of `NullPointerException` crashes in their "Google Home" app by 30% after switching the development of new features to Kotlin.

Using Kotlin doesn't introduce any new compatibility concerns or performance disadvantages to your apps, either. Kotlin is fully compatible with Java 8 and above, and the code generated by the compiler is executed efficiently. The runtime used by Kotlin is fairly small, so you won't experience a large increase in the size of the compiled application package. And when you use lambdas, many of the Kotlin standard library functions will inline them. Inlining lambdas ensures that no new objects will be created and the application won't suffer from extra GC pauses.

You'll benefit from all the cool new language features of Kotlin, and your

users will still be able to run your application on their devices, even if they don't run the latest version of Android.

### 1.3.3 Multiplatform: Sharing business logic and minimizing duplicate work on iOS, JVM, JS and beyond

Kotlin is also a *multiplatform* language. In addition to the Java Virtual Machine, Kotlin supports the following targets:

- It can be compiled to JavaScript, allowing you to run Kotlin code in the browser and runtimes such as Node.js.
- With Kotlin/Native, you can compile Kotlin code to native binaries, allowing you to target iOS and other platforms with self-contained programs.
- Kotlin/Wasm, a target that is still being developed at the moment of writing, will make it possible for you to compile your Kotlin code to the WebAssembly binary instruction format, and allowing you to run your code on the WebAssembly virtual machines that ship in modern browsers and other runtimes.

Kotlin also lets you specify which parts of your software should be shared between different targets, and which parts have a platform-specific implementation, on a very fine-grained level. Because this control is very fine-grained, you can mix and match the best combination of common and platform-specific code. This mechanism, which Kotlin calls *expect/actual*, allows you to take advantage of platform-specific functionality from your Kotlin code. This effectively mitigates the classic problem of "targeting the lowest common denominator" that cross-platform toolkits usually face, where you are limited to a subset of operations that is available on all platforms that you target.

A major use case we have seen for code sharing is mobile applications targeting both Android and iOS. With Kotlin Multiplatform, you only have to write your business logic once, but can use it in both iOS and Android targets in a completely native fashion, and even make use of the respective APIs, toolkits, and capabilities that these platforms offer. Similarly, sharing code between a server-side service and a JavaScript application running in the

browser helps you reduce duplicate work, help you keep validation logic in sync, and more.

If you want to learn more about the specifics of these additional platforms, as well as Kotlin's support for sharing code and multiplatform programming, please refer to the "Multiplatform programming" section of the Kotlin website (<https://kotlinlang.org/docs/multiplatform.html>).

Having looked at a selection of things that make Kotlin great, let's now look at Kotlin's philosophy—the main characteristics that distinguish Kotlin from other languages.

## 1.4 The philosophy of Kotlin

When we talk about Kotlin, we like to say that it's a pragmatic, concise, safe language with a focus on interoperability. What exactly do we mean by each of those words? Let's look at them in turn.

### 1.4.1 Kotlin is a pragmatic language designed to solve real-world problems

Being *pragmatic* means a simple thing to us: Kotlin is a practical language designed to solve real-world problems. Its design is based on many years of industry experience creating large-scale systems, and its features are chosen to address use cases encountered by many software developers. Moreover, developers worldwide have been using Kotlin for roughly a decade now. Their continued feedback has shaped each released version of the language. This makes us confident in saying that Kotlin helps solve problems in real projects.

Kotlin also is not a research language. It mostly relies on features and solutions that have already appeared in other programming languages and have proven to be successful. This reduces the complexity of the language and makes it easier to learn by letting you rely on familiar concepts. When new features are introduced, they remain in an "experimental" state for quite a long time. This makes it possible for the language design team to gather feedback, and allows the final design of a feature to be tweaked and fine-

tuned before it is added as a stable part of the language.

Kotlin doesn't enforce using any particular programming style or paradigm. As you begin to study the language, you can use the style and techniques that are familiar to you. Later, you'll gradually discover the more powerful features of Kotlin, such as extension functions ([3.3](#)), its expressive type-system ([7](#) and [8](#)), higher-order functions (**Chapter 10**) and many more. You will learn to apply them in your own code, which will make it concise and idiomatic.

Another aspect of Kotlin's pragmatism is its focus on tooling. A smart development environment is just as essential for a developer's productivity as a good language; and because of that, treating IDE support as an afterthought isn't an option. In the case of Kotlin, the IntelliJ IDEA plug-in is developed in lockstep with the compiler, and language features are always designed with tooling in mind.

The IDE support also plays a major role in helping you discover the features of Kotlin. In many cases, the tools will automatically detect common code patterns that can be replaced by more concise constructs, and offer to fix the code for you. By studying the language features used by the automated fixes, you can learn to apply those features in your own code as well.

### **1.4.2 Kotlin is concise, making the intent of your code clear while reducing boilerplate**

It's common knowledge that developers spend more time reading existing code than writing new code. Imagine you're a part of a team developing a big project, and you need to add a new feature or fix a bug. What are your first steps? You look for the exact section of code that you need to change, and only then do you implement a fix. You read a lot of code to find out what you have to do. This code might have been written recently by your colleagues, or by someone who no longer works on the project, or by you, but long ago. Only after understanding the surrounding code can you make the necessary modifications.

The simpler and more concise the code is, the faster you'll understand what's

going on. Of course, good design plays a significant role here. So does the choice of expressive names, ensuring that your variables, functions, and classes are accurately described by their names. But the choice of the language and its conciseness are also important. The language is *concise* if its syntax clearly expresses the intent of the code you read and doesn't obscure it with boilerplate required to specify how the intent is accomplished.

Kotlin tries hard to ensure that all the code you write carries meaning and isn't just there to satisfy code structure requirements. A lot of the standard boilerplate of object-oriented languages, such as getters, setters, and the logic for assigning constructor parameters to fields, is implicit in Kotlin and doesn't clutter your source code. Semicolons can also be omitted in Kotlin, removing a bit of extra clutter from your code, and its powerful type inference spares you from explicitly specifying types where the compiler can deduce them from the context.

Kotlin has a rich standard library that lets you replace these long, repetitive sections of code with library method calls. Kotlin's support for lambdas and anonymous functions (function literals that are used like expressions) makes it easy to pass small blocks of code to library functions. This lets you encapsulate all the common parts in the library and keep only the unique, task-specific portion in the user code.

At the same time, Kotlin doesn't try to collapse the source code to the smallest number of characters possible. For example, Kotlin supports overloading a fixed set of operators, meaning you can provide custom implementations for `+`, `-`, `in`, or `[]`. However, users can't define their *own*, custom operators. This prevents developers from replacing method names with cryptic punctuation sequences, which would be harder to read, and more difficult to find in documentation systems as opposed to using expressive names.

More concise code takes less time to write and, more important, less time to read and comprehend. This improves your productivity and lets you get things done faster.

### **1.4.3 Kotlin is safe, protecting you from whole categories of**

## errors

In general, when we speak of a programming language as *safe*, we mean its design prevents certain kinds of errors in a program. Of course, this isn't an absolute quality; no language prevents all possible errors. In addition, preventing errors usually comes at a cost. You need to give the compiler more information about the intended operation of the program, so the compiler can then verify that the information matches what the program does. Because of that, there's always a trade-off between the level of safety you get and the loss of productivity required to put in more detailed annotations.

Running on the JVM already provides a lot of safety guarantees: for example, memory safety, preventing buffer overflows, and other problems caused by incorrect use of dynamically allocated memory. As a statically typed language on the JVM, Kotlin also ensures the type safety of your applications. And Kotlin goes further: It is easy to define immutable variables (via the `val` keyword) and quick to group them in immutable (data) classes, resulting in additional safety for multithreaded applications.

Beyond that, Kotlin aims to prevent errors happening at runtime by doing checks during compile time. Most important, Kotlin strives to remove the `NullPointerException` from your program. Kotlin's type system tracks values that can and can't be `null` and forbids operations that can lead to a `NullPointerException` at runtime. The additional cost required for this is minimal: marking a type as nullable takes only a single character, a question mark at the end:

```
fun main() {
    var s: String? = null #1
    var s2: String = "" #2

    println(s.length) #3
    println(s2.length) #4
}
```

To complement this, Kotlin provides many convenient ways to handle nullable data. This helps greatly in eliminating application crashes.

Another type of exception that Kotlin helps avoid is the class cast exception,

which happens when you cast an object to a type without first checking that it has the right type. Kotlin combines check and cast into a single operation. That means once you've checked the type, you can refer to members of that type without any additional casts, redeclarations, or checks.

In this example, `is` performs a type-check on the `value` variable, which may be of `Any` type. The compiler knows that in the `true` branch of the conditional, `value` has to be of type `String`, so it can safely permit the usage of methods from that type (a so-called *smart-cast*, which we'll get to know in more detail in [2.3.6](#)).

```
fun modify(value: Any) {  
    if (value is String) { #1  
        println(value.uppercase()) #2  
    }  
}
```

Next, let's talk specifically about Kotlin for JVM target: Kotlin provides seamless interoperability with Java.

#### **1.4.4 Kotlin is interoperable, allowing reuse of existing Java code to the highest degree**

Regarding interoperability, your first concern probably is, "Can I use my existing libraries?" With Kotlin, the answer is, "Yes, absolutely." Regardless of the kind of APIs the library requires you to use, you can work with them from Kotlin. You can call Java methods, extend Java classes and implement interfaces, apply Java annotations to your Kotlin classes, and so on.

Unlike some other JVM languages, Kotlin goes even further with interoperability, making it effortless to call Kotlin code from Java as well. No tricks are required: Kotlin classes and methods can be called exactly like regular Java classes and methods. This gives you the ultimate flexibility in mixing Java and Kotlin code anywhere in your project. When you start adopting Kotlin in your Java project, you can run the Java-to-Kotlin converter on any single class in your codebase, and the rest of the code will continue to compile and work without any modifications. This works regardless of the role of the class you've converted—something we'll take a

closer look at in section **For Java developers: move code automatically with the Java-to-Kotlin converter**.

Another area where Kotlin focuses on interoperability is its use of existing Java libraries to the largest degree possible. For example, Kotlin's collections rely almost entirely on Java standard library classes, extending them with additional functions for more convenient use in Kotlin. (We'll look at the mechanism for this in more detail in [3.3](#).) This means you never need to wrap or convert objects when you call Java APIs from Kotlin, or vice versa. All the API richness provided by Kotlin comes at no cost at runtime.

The Kotlin tooling also provides full support for cross-language projects. It can compile an arbitrary mix of Java and Kotlin source files, regardless of how they depend on each other. IDE features inside IntelliJ IDEA and Android Studio work across languages as well, allowing you to:

- Navigate freely between Java and Kotlin source files
- Debug mixed-language projects and step between code written in different languages
- Refactor your Java methods and have their use in Kotlin code correctly updated, and vice versa

**Figure 1.2. When using the "Find Usages" action in IntelliJ IDEA, it finds results across Kotlin and Java files in the same project. Other IDE features, such as refactorings and navigation, work just as smoothly across both languages.**

example - Car.kt [example.main]

java > CarManufacturer

Add Configuration... ▶ ⚙️ 🔍 ⚙️ ⚙️ ⚙️

Project Structure

Car.kt

```
data class Car()
    val tires: Int = 4
```

CarInspector

CarManufacturer

kotlin

Car

CarAnnouncer

Find: Car in Project and Libraries

Usages in Project and Libraries 5 results

- CarAnnouncer.kt 1 result
  - fun announce(c: Car) {
- CarInspector.java 1 result
  - void inspect(Car c) {
- CarManufacturer.java 2 results
  - Car manufacture() {
  - return new Car();

Select item to preview

Preview

Call Hierarchy

Dataflow to Here

Dataflow from Here

Find Problems Version Control Profiler Terminal TODO Build Dependencies

Gradle sync finished in 10 s 793 ms (3 minutes ago)

3:1 LF UTF-8 4 spaces 🔒 🌐

Hopefully by now we've convinced you to give Kotlin a try. Now, how can you start using it? In the next section, we'll discuss the process of compiling and running Kotlin code, both from the command line and using different tools.

## 1.5 Using the Kotlin tools

Let's have an overview of the Kotlin tools. First, let's discuss how to set up your environment to run the Kotlin code.

### 1.5.1 Setting up and running the Kotlin code

You can run small snippets online, or install an IDE - the best experience you'll get with IntelliJ IDEA or Android Studio. We provide the basic information here, but the best up-to-date tutorials are on the Kotlin website. If you need the detailed information about getting your environment set up, or information about different compilation targets, please refer to the "Getting started" section of the Kotlin website (<https://kotlinlang.org/docs/getting-started.html>).

### Try Kotlin without installation with the Kotlin online playground

The easiest way to try Kotlin doesn't require any installation or configuration. At <https://play.kotlinlang.org/>, you can find an online playground where you can write, compile, and run small Kotlin programs. The playground has code samples demonstrating the features of Kotlin, as well as a series of exercises for learning Kotlin interactively. Alongside it, the Kotlin documentation (<https://kotlinlang.org/docs>) also has a number of interactive samples that you can run right in the browser.

These are the quickest ways to run short snippets of Kotlin, but they provides less assistance and guidance. It's a very minimal development environment that is missing features such as autocomplete or inspections that can tell you how to improve your Kotlin code. The web version is also doesn't support user interactions via the standard input stream, or working with files and

directories. However, these features are all conveniently available inside IntelliJ IDEA and Android Studio.

## **Plug-in for IntelliJ IDEA and Android Studio**

The IntelliJ IDEA plug-in for Kotlin has been developed in parallel with the language, and it's a full-featured development environment for Kotlin. It's mature and stable, and it provides a complete set of tools for Kotlin development.

The Kotlin plug-in is included out of the box with IntelliJ IDEA and Android Studio, so no additional setup is necessary. You can use either the free, open source IntelliJ IDEA Community Edition or Android Studio, or the commercial IntelliJ IDEA Ultimate. In IntelliJ IDEA, select Kotlin in the New Project dialog, and you're good to go. In Android Studio, create a new project, and you can directly start writing Kotlin.

You can also check the "Get started with Kotlin/JVM" tutorial with detailed instructions and screenshots on how to create a project in IntelliJ IDEA: <https://kotlinlang.org/docs/jvm-get-started.html>.

## **For Java developers: move code automatically with the Java-to-Kotlin converter**

Getting up to speed with a new language is never effortless. Fortunately, we've built a nice little shortcut that lets you speed up your learning and adoption by relying on your existing knowledge of Java. This tool is the automated Java-to-Kotlin converter.

As you start learning Kotlin, the converter can help you express something when you don't remember the exact syntax. You can write the corresponding snippet in Java and then paste it into a Kotlin file, and the converter will automatically offer to translate the code into Kotlin. The result won't always be the most idiomatic, but it will be working code, and you'll be able to make progress with your task.

The converter is also great for introducing Kotlin into an existing Java

project. When you need to write a new class, you can do it in Kotlin right from the beginning. But if you need to make significant changes to an existing class, you may also want to use Kotlin in the process. That's where the converter comes into play. You convert the class into Kotlin first, and then you add the changes using all the benefits of a modern language.

Using the converter in IntelliJ IDEA is extremely easy. You can either copy a Java code fragment and paste it into a Kotlin file, or invoke the Convert Java File to Kotlin File action if you need to convert an entire file.

### **1.5.2 Compiling Kotlin code**

Kotlin is a compiled language, which means before you can run Kotlin code, you need to compile it. As we discussed in [1.3.3](#), the Kotlin code can be compiled to different targets:

- JVM bytecode (stored in .class files) to run on Java Virtual Machine
- JVM bytecode to be further transformed and run on Android
- Native targets to run natively on different operating systems
- JavaScript (and WebAssembly) to run in browser

For the Kotlin compiler, it's not important whether the produced JVM bytecode runs on JVM or is further transformed and runs on Android. Android Runtime (ART) transforms the JVM bytecode to Dex bytecode and runs it instead. For more details on how it works on Android please refer to the documentation <https://source.android.com/devices/tech/dalvik>.

Since the main target for this book is Kotlin/JVM, let's discuss in more details how the compilation process works there. You can find information about other targets on the Kotlin website.

### **Compilation process for Kotlin/JVM**

Kotlin source code is normally stored in files with the extension .kt. When compiling Kotlin code for the JVM target, the compiler analyzes the source code and generates .class files, just like the Java compiler does. The generated .class files are then packaged and executed using the standard

procedure for the type of application you're working on.

In the simplest case, you can use the `kotlinc` command to compile your code from the command line and use the `java` command to execute your code:

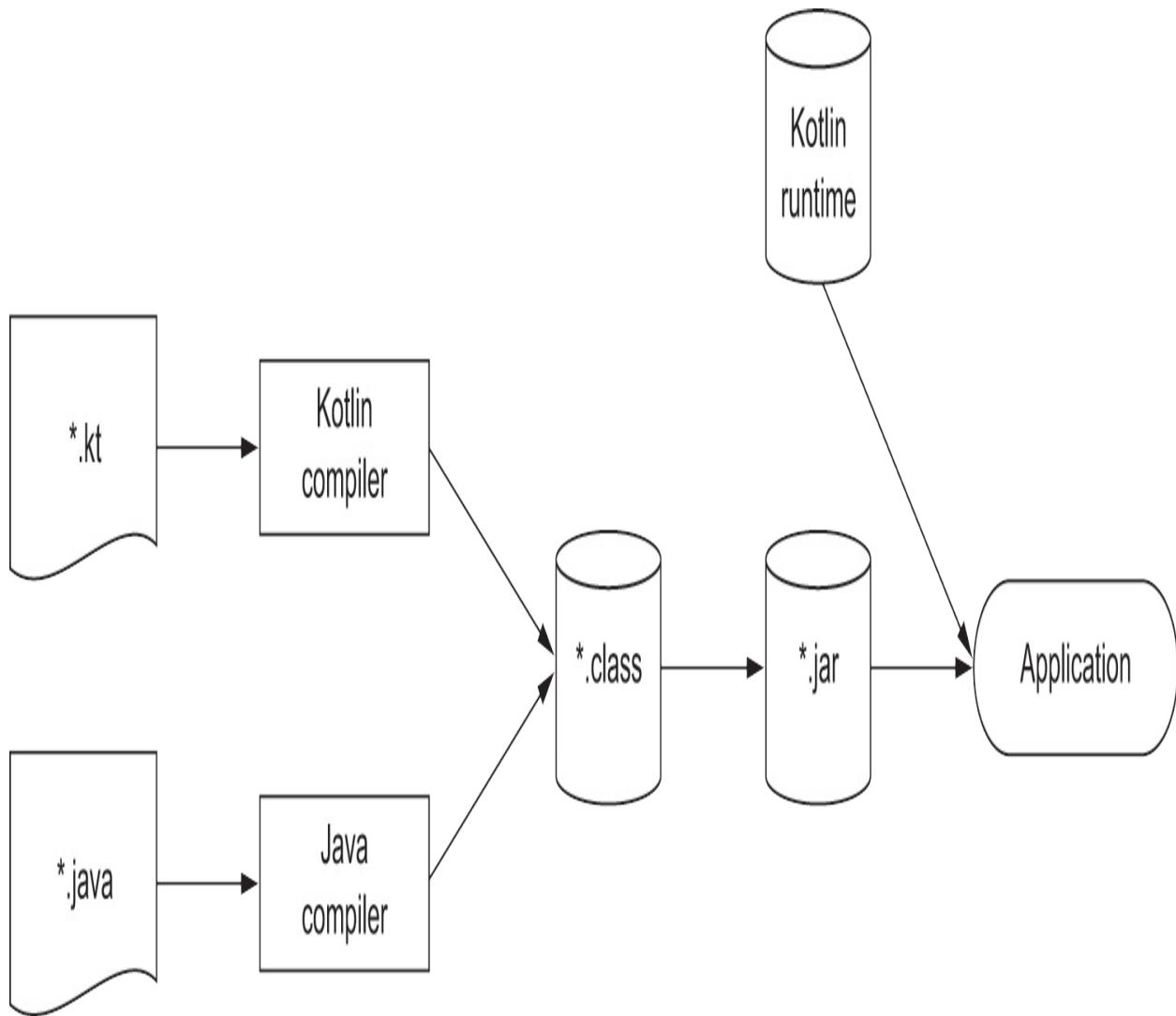
```
kotlinc <source file or directory> -include-runtime -d <jar name>
java -jar <jar name>
```

Java Virtual Machine can run `.class` files compiled from the Kotlin code without knowing whether they were written initially in Java or in Kotlin. Kotlin built-in classes and their APIs, however, differ from those in Java, and to correctly run the compiled code, JVM needs the additional information as a dependency: *Kotlin runtime library*. When compiling code from the command line, we explicitly invoked `-include-runtime` to include this runtime library into the resulting jar file.

Kotlin runtime library contains the definitions of Kotlin's basic classes like `Int` or `String` and some extensions that Kotlin adds to the standard Java APIs. The runtime library needs to be distributed with your application.

A simplified description of the Kotlin build process is shown in [1.3](#).

**Figure 1.3. Kotlin build process**



In addition, you need the *Kotlin standard library* included as a dependency in your application. In theory, you can write the Kotlin code without it, but in practice you never need to do so. The standard library contains the definitions of such fundamental classes like `List`, `Map` or `Sequence`, and many methods for working with them. We'll be discussing the most important classes and their APIs in detail in this book.

In most real-life cases, you'll be using a build system such as Gradle or Maven to compile your code. Kotlin is compatible with these build systems. All of those build systems also support mixed-language projects that combine Kotlin and Java in the same codebase. Maven and Gradle take care of including both the Kotlin runtime library and (for the latest versions) Kotlin standard library as dependencies of your application, so you don't need to

include them explicitly.

The best up-to-date way to check the details of how to set up the project with the build system of your choice is the Kotlin documentation: please check <https://kotlinlang.org/docs/gradle.html> and <https://kotlinlang.org/docs/maven.html> sections. For a quick start, you don't need to know all the peculiarities, you can simply create a new project, and the correct build file with the necessary dependencies will be generated for you.

## 1.6 Summary

- Kotlin is statically typed and supports type inference, allowing it to maintain correctness and performance while keeping the source code concise.
- Kotlin supports both object-oriented and functional programming styles, enabling higher-level abstractions through first-class functions and simplifying testing and multithreaded development through the support of immutable values.
- Coroutines are a lightweight alternative to threads, and help make asynchronous code feel natural by allowing you to write logic that looks similar to sequential code, and help structure concurrent code in parent-child relationships.
- Kotlin works well for server-side applications, with Kotlin-first frameworks like Ktor and http4k, as well as fully supporting all existing Java frameworks like Spring Boot.
- Android is Kotlin-first, with development tools, libraries, samples and documentation all primarily focused on Kotlin.
- Kotlin Multiplatform brings your Kotlin code to targets beyond the JVM, including iOS and the web.
- Kotlin is free and open source, and supports multiple build systems and IDEs.
- IntelliJ IDEA and Android Studio allow you to navigate smoothly across code written in both Kotlin and Java.
- The Kotlin playground (<https://play.kotlinlang.org>) is a fast way to try Kotlin without any setup.
- The automated Java-to-Kotlin converter allows you to bring your

existing code and knowledge to Kotlin.

- Kotlin is pragmatic, safe, concise, and interoperable, meaning it focuses on using proven solutions for common tasks, preventing common errors such as `NullPointerExceptions`, supporting compact and easy-to-read code, and providing unrestricted integration with Java.

# 2 Kotlin basics

## This chapter covers

- Declaring functions, variables, classes, enums, and properties
- Control structures in Kotlin
- Smart casts
- Throwing and handling exceptions

In this chapter, you'll learn the basics of the Kotlin language required to write your first working Kotlin programs. These include basic building blocks that you encounter all throughout Kotlin programs, like variables and functions. You'll also get acquainted with different ways of representing data in Kotlin, via enum, as well as classes and their properties.

The different control structures you'll learn throughout this chapter will give you the tools needed to use conditional logic in your programs, as well as iterate using loops, and will learn what makes these constructs special when compared to other languages like Java.

We'll also introduce the basic mechanics of types in Kotlin, starting with the concept of a *smart cast*, an operation that combines a type check and a cast into one operation. You'll see how this helps you remove redundancy from your code without sacrificing safety. We'll also briefly talk about exception handling, and Kotlin's philosophy behind it.

By the end of this chapter, you'll already be able to combine these basic bits and pieces of the Kotlin language to write your own working Kotlin code, even if it might not be the most *idiomatic*.

### What's "idiomatic Kotlin"?

When discussing Kotlin code, a certain phrase often reoccurs: *idiomatic Kotlin*. You'll certainly hear this phrase throughout this book, but you might also hear it when talking to your colleagues, when attending community

events, or at conferences. Clearly, it's worth understanding what is meant by it.

Simply said, idiomatic Kotlin is how a "native Kotlin speaker" writes code, using language features and syntactic sugar where appropriate. Such code is made up of *idioms*—recognizable structures that address problems you're trying to solve in "the Kotlin way". Idiomatic code fits in with the programming style generally accepted by the community, and follows the recommendations of the language designers.

Like any skill, learning to write idiomatic Kotlin takes time and practice. As you progress through this book, inspect the provided code samples, and write your own code, you will gradually develop an intuition to what idiomatic Kotlin code looks and feels like, and will gain the ability to independently apply these learnings in your own code.

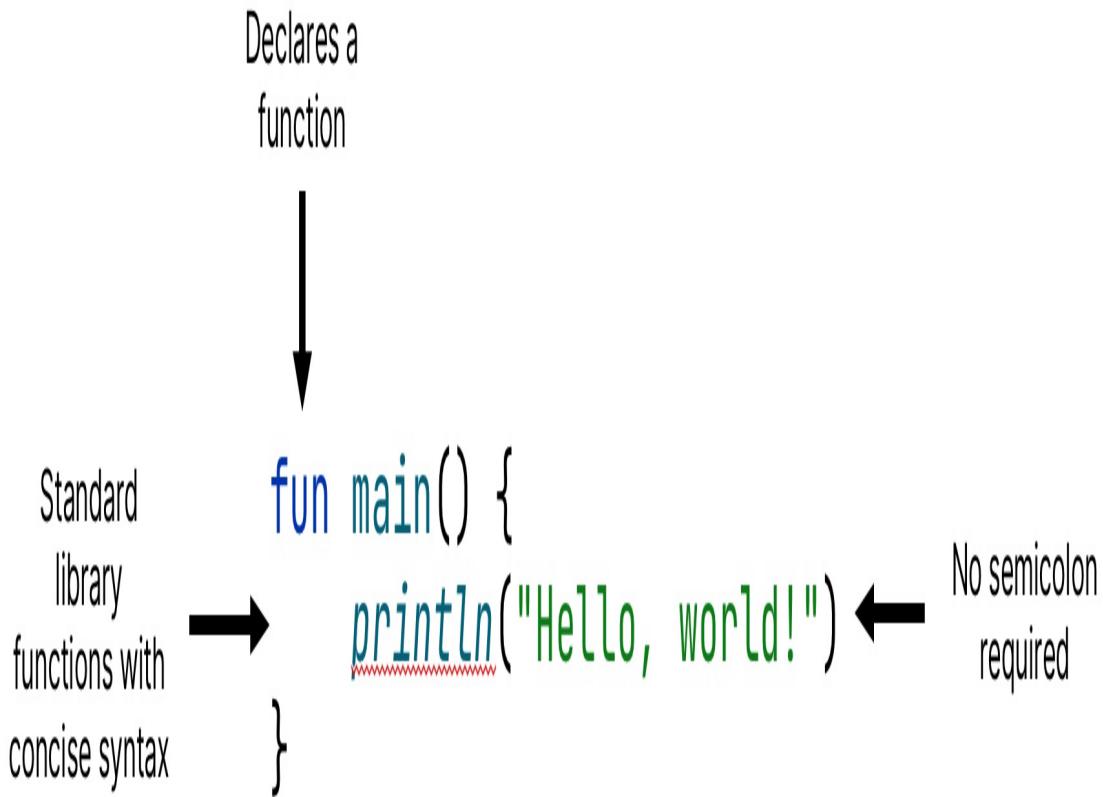
## 2.1 Basic elements: functions and variables

This section introduces you to the basic elements that every Kotlin program consists of: functions and variables. You'll write your very first Kotlin program, see how Kotlin lets you omit many type declarations and how it encourages you to avoid using mutable data where possible—and why that's a good thing.

### 2.1.1 Writing your first Kotlin program: Hello, world!

Let's start our journey into the world of Kotlin with a classical example: a program that prints "Hello, world!". In Kotlin, it's just one function:

**Figure 2.1. "Hello World!" in Kotlin**



We can observe a number of features and parts of the language syntax in this simple code snippet already:

- The `fun` keyword is used to declare a function. Programming in Kotlin is lots of fun, indeed!
- The function can be declared at the top level of any Kotlin file; you don't need to put it in a class.
- You can specify the `main` function as the entrypoint for your application at the top level, and without additional arguments (other languages may require you to always accept an array of command line parameters, for example).

- Kotlin emphasizes conciseness: You just write `println` to get your text to be displayed in the console. The Kotlin standard library provides many wrappers around standard Java library functions (such as `System.out.println`), with more concise syntax, and `println` is one of them.
- You can (and should) omit the semicolon from the end of a line, just as in many other modern languages.

So far, so good! We'll discuss some of these topics in more detail later. Now, let's explore the function declaration syntax.

## 2.1.2 Declaring functions with parameters and return values

The first function you wrote didn't actually return any meaningful values. However, the purpose of functions is often to compute and subsequently return some kind of result. For example, you may want to write a simple function `max` that takes two integer numbers `a` and `b` and returns the larger of the two. So, what would that look like?

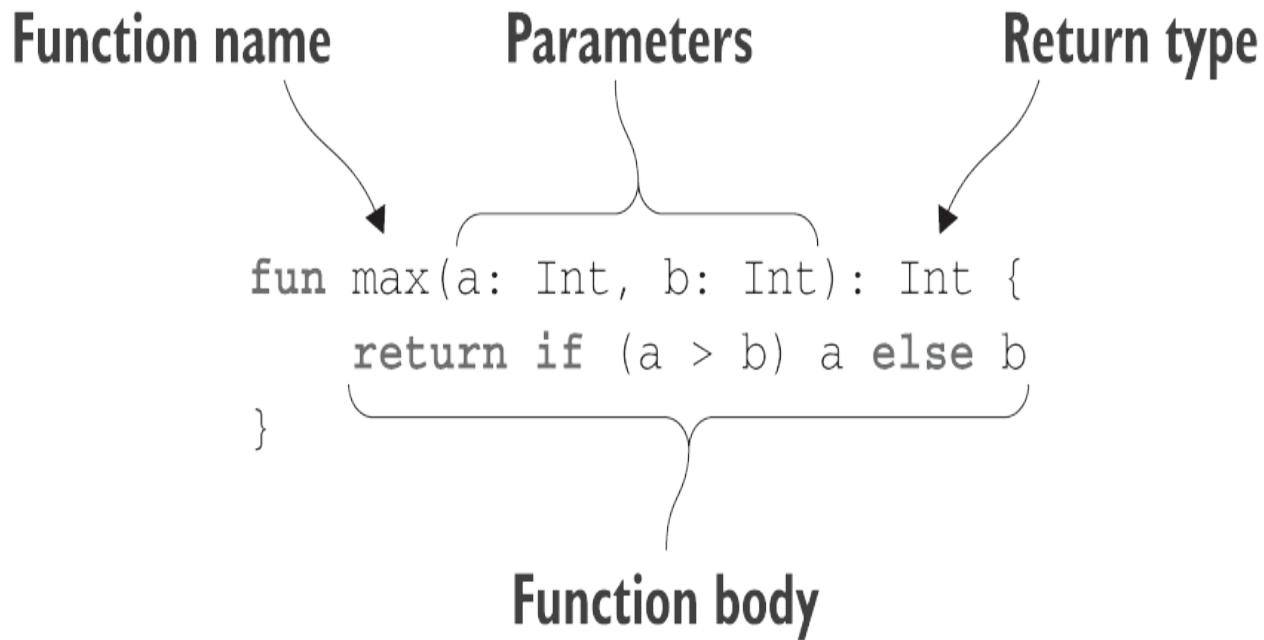
The function declaration starts with the `fun` keyword, followed by the function name: `max`, in this case. It's followed by the parameter list in parentheses. Here, we declare two parameters, `a` and `b`, both of type `Int`. In Kotlin, you first specify the parameter name, and then specify the type, separated by a colon. The return type comes after the parameter list, separated from it by a colon.

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

[2.2](#) shows you the basic structure of a function. Note that in Kotlin, `if` is an expression with a result value. You can think of `if` returning a value from either of its branches. This makes it similar to the ternary operator in other languages like Java, where the same construct might look like `(a > b) ? a : b`.

**Figure 2.2. A Kotlin function is introduced with the `fun` keyword. Parameters and their types follow in parentheses, each one of them annotated with a name and a type, separated by a colon. Its return type is specified after the end of the parameter list. Functions just like this one are a**

basic building block of any Kotlin program.



You can then call your function by using its name, providing the arguments in parentheses (you'll learn about different ways for calling Kotlin functions in [3.2.1](#).)

```
fun main() {  
    println(max(1, 2))  
    // 2  
}
```

#### Parameters and return type of the `main` function

As you already saw in the "Hello, World" example, the entry point of every Kotlin program is its `main` function. This function can either be declared with no parameters, or with an array of strings as its arguments (`args: Array<String>`). In the latter case, each element in the array corresponds to a command line parameter passed to your application. In any case, the `main` function does not return any value.

#### The difference between expressions and statements

In Kotlin, `if` is an expression, not a statement. The difference between an expression and a statement is that an expression has a value, which can be

used as part of another expression, whereas a statement is always a top-level element in its enclosing block and doesn't have its own value. In Kotlin, most control structures, except for the loops (`for`, `while`, and `do/while`) are expressions, which sets it apart from other languages like Java. Specifically, the ability to combine control structures with other expressions lets you express many common patterns concisely, as you'll see later in the book. As a sneak peek, here are some snippets that are valid in Kotlin:

```
val x = if(myBoolean) 3 else 5
val direction = when(inputString) {
    "u" -> UP
    "d" -> DOWN
}
val number = try {
    inputString.toInt()
} catch(nfe: NumberFormatException) {
    -1
}
```

On the other hand, Kotlin enforces that assignments are always statements in Kotlin—that is, when assigning a value to a variable, this assignment operation doesn't itself return a value.

This helps avoid confusion between comparisons and assignments, which is a common source of mistakes in languages that treat them as expressions, such as Java or C/C++. That means the following isn't valid Kotlin code:

```
val number: Int
val alsoNumber = i = getNumber()
// ERROR: Assignments are not expressions,
// and only expressions are allowed in this context
```

### 2.1.3 Making function definitions more concise by using expression bodies

In fact, you can make your `max` function even more concise. Since its body consists of a single expression (`if (a > b) a else b`), you can use that expression as the entire body of the function, removing the curly braces and the `return` statement. Instead, you can place the single expression right after an equals sign `=`:

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

If a function is written with its body in curly braces, we say that this function has a *block body*. If it returns an expression directly, it has an *expression body*.



#### Convert between expression body and block body in IntelliJ IDEA and Android Studio

IntelliJ IDEA and Android Studio provide intention actions to convert between the two styles of functions: "Convert to expression body" and "Convert to block body." You can find them via the lightbulb icon when your cursor is placed on the function, or via the Alt + Enter (or Option + Return on macOS) keyboard shortcut.

Functions that have an expression body are a frequent occurrence in Kotlin code. You've already seen that they are quite convenient when your function happens to a trivial one-liner that intends to give a conditional check or an often-used operation a memorable name. But they also find use when functions evaluate a single, more complex expression, such as `if`, `when`, or `try`. You'll see such functions later in this chapter, when we talk about the `when` construct.

You could simplify your `max` function even more and omit the return type:

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

At first sight, this might seem puzzling to you. How can there be functions without return-type declarations? You've already learned that Kotlin is a statically typed language—so doesn't it require for every expression to have a type at compile time?

Indeed, every variable and every expression has a type, and every function has a return type. But for expression-body functions, the compiler can analyze the expression used as the body of the function and use its type as the function return type, even when it's not specified explicitly. This type of analysis is usually called *type inference*.

Note that omitting the return type is allowed only for functions with an

expression body. For functions with a block body that return a value, you have to specify the return type and write the `return` statements explicitly. That's a conscious choice. A real-world function often is long and can contain several `return` statements; having the return type and the `return` statements written explicitly helps you quickly grasp what can be returned.

#### **Keep your return types explicit when writing a library**

If you are authoring libraries that other developers depend upon, you may want to, you may want refrain from using inferred return types for functions that are part of your public API. By explicitly specifying the types of your functions, you can avoid accidental signature changes that could cause errors in the code of your library's consumers. In fact, Kotlin provides tooling in the form of compiler options that can automatically check that you're explicitly specifying return types. You'll learn more about this *explicit API mode* in [4.1.3](#).

Let's look at the syntax for variable declarations next.

#### **2.1.4 Declaring variables to store data**

Another basic building block which you'll commonly use in all your Kotlin programs are variables, which allow you to store data. A variable declaration in Kotlin starts with a keyword (`val` or `var`), followed by the name for the variable. While Kotlin lets you omit the type for many variable declarations (thanks to its powerful *type inference* that you've already seen in [2.1.3](#)), you can always explicitly put the type after the variable name. For example, if you need to store one of the most famous questions and its respective answer in a Kotlin variable, you could do so by specifying two variables `question` and `answer` with their explicit types—`String` for the textual question, and `Int` for the integer answer:

```
val question: String =  
    "The Ultimate Question of Life, the Universe, and Everything"  
val answer: Int = 42
```

You can also omit the type declarations, making the example a bit more concise:

```
val question =  
    "The Ultimate Question of Life, the Universe, and Everything"  
val answer = 42
```

Just as with expression-body functions, if you don't specify the type, the compiler analyzes the initializer expression and uses its type as the variable type. In this case, the initializer, 42, is of type `Int`, so the variable `answer` will have the same type.

If you use a floating-point constant, the variable will have the type `Double`:

```
val yearsToCompute = 7.5e6 #1
```

The number types, along other basic types, are covered in more depth in [8.1](#).

If you're not initializing your variable immediately, but only assigning it at a later point, the compiler won't be able to infer the type for the variable. In this case, you need to specify its type explicitly:

```
fun main() {  
    val answer: Int  
    answer = 42  
}
```

## 2.1.5 Marking a variable as read-only or reassignable

To control when a variable can be assigned a new value, Kotlin provides you with two keywords, `val` and `var`, for declaring variables:

1. `val` (from *value*) declares a *read-only reference*. A variable declared with `val` can be assigned only once. After it has been initialized, it can't be reassigned a different value. (For comparison, in Java, this would be expressed via the `final` modifier.)
2. `var` (from *variable*) declares a *re assignable reference*. You can assign other values to such a variable, even after it has been initialized. (This behavior is analogous to a regular, non-final variable in Java)

By default, you should strive to declare all variables in Kotlin with the `val` keyword. Change it to `var` only if necessary. Using read-only references, immutable objects, and functions without side effects allows you to take

advantage of the benefits offered by the *functional programming* style. We touched briefly on its advantages in [1.2.3](#), and we'll return to this topic in [5](#).

A `val` variable must be initialized exactly once during the execution of the block where it's defined. But you can initialize it with different values depending on some condition, as long as the compiler can ensure that only one of the initialization statements will be executed.

You may find yourself in the situation where you want to assign the the contents of a `result` variable depending on the return value of another function, like `canPerformOperation`. Because the compiler is smart enough to know that exactly one of the two potential assignments will be executed, you can still specify `result` as a read-only reference using the `val` keyword:

```
fun canPerformOperation(): Boolean {
    return true
}

fun main() {
    val result: String
    if (canPerformOperation()) {
        result = "Success"
    } else {
        result = "Can't perform operation"
    }
}
```

Note that, even though a `val` reference is itself read-only and can't be changed once it has been assigned, the object that it points to may be mutable. For example, adding an element to a mutable list, which is referenced by a read-only reference, is perfectly okay:

```
fun main() {
    val languages = mutableListOf("Java") #1
    languages.add("Kotlin") #2
}
```

In [8.2.2](#), we'll discuss mutable and read-only objects in more detail.

Even though the `var` keyword allows a variable to change its value, its type is fixed. For example, if you decided mid-program that the `answer` variable

should store a string instead of an integer, you would be met with a compile error:

```
fun main() {  
    var answer = 42  
    answer = "no answer" #1  
}
```

There's an error on the string literal because its type (`String`) isn't as expected (`Int`). The compiler infers the variable type only from the initializer and doesn't take subsequent assignments into account when determining the type.

If you need to store a value of a mismatching type in a variable, you must manually convert or coerce the value into the right type. We'll discuss number conversions in [8.1.4](#).

Now that you know how to define variables, it's time to see some new tricks for referring to values of those variables. Specifically, you'll see how you can provide some nicer and structured outputs for your first Kotlin programs.

## 2.1.6 Easier string formatting: string templates

Let's get back to the "Hello World" example that opened this section, and extend it with some extra features that occur commonly in all kinds of Kotlin programs.

You'll add a bit of personalization, by having the program greet people by name. If the user specifies a name via the standard input, then your program uses it in the greeting. In case the user doesn't actually provide any input, you'll just have to greet all of Kotlin instead. Such a greeting program could look as follows, and showcases a few features you haven't seen before:

**Listing 2.1. Using string templates**

```
fun main() {  
    val input = readln()  
    val name = if (input.isNotBlank()) input else "Kotlin"  
    println("Hello, $name!") #1  
}
```

This example introduces a feature called *string templates*, and also briefly shows an example of how you could read simple user input. In the code, you read `input` from the standard input stream via the `readln()` function (which is available in any Kotlin file, alongside others). You then declare a variable `name` and initialize its value using an `if`-expression. If the standard input exists and is not blank, `name` is assigned the value of `input`. Otherwise it's assigned a default value "Kotlin". Finally, you use it in the string literal passed to `println`.

Like many scripting languages, Kotlin allows you to refer to local variables in string literals by putting the `$` character in front of the variable name. This is equivalent to Java's string concatenation ("Hello, " + `name` + "!") but is more compact and just as efficient.<sup>[1]</sup> And of course, the expressions are statically checked, and the code won't compile if you try to refer to a variable that doesn't exist.

If you need to include the `$` character in a string, you escape it with a backslash:

```
fun main() {
    println("\$x") #1
    // $x
}
```

String templates in Kotlin are quite powerful, since they don't limit you to referencing individual variables. If you want your greeting to be a bit more adventurous, and greet your user by the length of their name, you can also provide a more complex expression in the string template. All it takes is putting curly braces around the expression:

```
fun main() {
    val name = readln()
    if (name.isNotBlank()) {
        println("Hello, ${name.length}-letter person!") #1
    }
}
```

Now that you know that you can include arbitrary expressions inside string templates, and you already knew that `if` is an expression in Kotlin. Combining the two, you can now rewrite your greeting program to include

the conditional directly inside the string template:

```
fun main() {  
    val name = readln()  
    println("Hello, ${if (name.isBlank()) "someone" else name}!")  
}
```

Notice how you can even use an expression that itself contains double quotes within a string template!

Later, in [3.5](#), we'll return to strings and talk more about what you can do with them.

Now you already know a few of the most basic building blocks to write your own Kotlin programs—functions and variables. Let's go one step up in the hierarchy and look at classes and how they help you encapsulate and group related data in an object-oriented fashion. This time, you'll use the Java-to-Kotlin converter to help you get started using the new language features.

[\[1\]](#) For JVM 1.8 targets, the compiled code creates a `StringBuilder` and appends the constant parts and variable values to it. Applications targeting JVM 9 or above compile string concatenations into more efficient dynamic invocations via `invokedynamic`.

## 2.2 Encapsulating behavior and data: classes and properties

Just like other object-oriented programming languages, Kotlin provides the abstraction of a *class*. Kotlin's concepts in this area will be familiar to you, but you'll find that many common tasks can be accomplished with much less code compared to other object-oriented languages. This section will introduce you to the basic syntax for declaring classes. We'll go into more detail in [4](#).

To begin, let's look at a simple POJO ("Plain Old Java Object") `Person` class that so far contains only one property, `name`.

**Listing 2.2. Simple Java class Person**

```
/* Java */
public class Person {
    private final String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

You can see that in Java, this requires quite some code. The constructor body is entirely repetitive, as it only assigns the parameters to the fields that have the corresponding names. As per convention, to access the name field, the Person class should also provide a getter function getName, which also just returns the contents of the field. This kind of repetition often happens in Java. In Kotlin, this logic can be expressed without so much boilerplate.

In section **For Java developers: move code automatically with the Java-to-Kotlin converter**, we introduced the Java-to-Kotlin converter: a tool that automatically replaces Java code with Kotlin code that does the same thing. Let's look at the converter in action and convert the Person class to Kotlin.

#### **Listing 2.3. Person class converted to Kotlin**

```
class Person(val name: String)
```

Looks good, doesn't it? If you've tried another modern object-oriented language, you may have seen something similar. As you can see, Kotlin offers a concise syntax for declaring classes, especially classes that contain only data but no code. We'll discuss their relationship to a very similar concept in Java 14 and above, *Records*, in section **Data classes and immutability: the copy() method**.

Note that the modifier `public` disappeared during the conversion from Java to Kotlin. In Kotlin, `public` is the default visibility, so you can omit it.

### **2.2.1 Associating data with a class and making it accessible:**

## properties

The idea of a class is to encapsulate data and code that works on that data into a single entity. In Java, the data is stored in fields, which are usually private. If you need to let clients of the class access that data, you provide *accessor methods*: a getter and possibly a setter. You saw an example of this in the `Person` class. The setter can also contain additional logic for validating the passed value, sending notifications about the change and so on.

In Java, the combination of the field and its accessors is often referred to as a *property*, and many frameworks make heavy use of that concept. In Kotlin, properties are a first-class language feature, which entirely replaces fields and accessor methods. You declare a property in a class the same way you declare a variable: with the `val` and `var` keywords. A property declared as `val` is read-only, whereas a `var` property is mutable and can be changed.

For example, you could expand your `Person` class, which already contained a read-only `name` property, with a mutable `isStudent` property:

**Listing 2.4. Declaring a mutable property in a class**

```
class Person(  
    val name: String, #1  
    var isStudent: Boolean #2  
)
```

Basically, when you declare a property, you declare the corresponding accessors (a getter for a read-only property, and both a getter and a setter for a writable one). By default, the implementation of these accessors is trivial: a field is created to store the value, the getter returns the value of this field, and the setter updates its value. But if you want to, you may declare a custom accessor that uses different logic to compute or update the property value.

The concise declaration of the `Person` class in [2.4](#) hides the same underlying implementation as the original Java code: it's a class with private fields that is initialized in the constructor and can be accessed through the corresponding getter. That means you can use this class from Java and from Kotlin the same way, independent of where it was declared. The use looks

identical. Here's how you can use the Kotlin class Person from Java code, creating a new Person object with the name Bob who is a student—until he graduates:

**Listing 2.5. Using the Person class from Java**

```
public class Demo {  
    public static void main(String[] args) {  
        Person person = new Person("Bob", true);  
        System.out.println(person.getName());  
        // Bob  
        System.out.println(person.isStudent());  
        // true  
        person.setStudent(false); // Graduation!  
        System.out.println(person.isStudent());  
        // false  
    }  
}
```

Note that this looks the same when Person is defined in Java and in Kotlin. Kotlin's name property is exposed to Java as a getter method called getName. The getter and setter naming rule has an exception: if the property name starts with is, no additional prefix for the getter is added and in the setter name, is is replaced with set. Thus, from Java, you can call isStudent() and setStudent() to access the isStudent property .

If you convert [2.5](#) to Kotlin, you get the following result.

**Listing 2.6. Using the Person class from Kotlin**

```
fun main() {  
    val person = Person("Bob", true) #1  
    println(person.name) #2  
    // Bob  
    println(person.isStudent) #2  
    // true  
    person.isStudent = false // Graduation! #3  
    println(person.isStudent)  
    // false  
}
```

Now, instead of invoking the getter explicitly, you reference the property

directly. The logic stays the same, but the code is more concise. Setters of mutable properties work the same way: while in Java, you use `person.setStudent(false)` to symbolize a graduation; in Kotlin, you use the property syntax directly, and write `person.isStudent = false`.



### Tip

You can also use the Kotlin property syntax for classes defined in Java. Getters in a Java class can be accessed as `val` properties from Kotlin, and getter/setter pairs can be accessed as `var` properties. For example, if a Java class defines methods called `getName` and `setName`, you can access it as a property called `name`. If it defines `isStudent` and `setStudent` methods, the name of the corresponding Kotlin property will be `isStudent`.

In most cases, the property has a corresponding backing field that stores the property value (you'll learn more about backing fields in Kotlin in [4.2.4](#)). But if the value can be computed on the fly—for example, by deriving it from other properties—you can express that using a custom getter.

## 2.2.2 Computing properties instead of storing their values: custom accessors

Let's see how you could provide a custom implementation of a property accessor. One common case for this is when a property is a direct result of some other properties in the object. If you have a `Rectangle` class that stores `width` and `height`, you can provide a property `isSquare` that is true when `width` and `height` are equal. Because this is a property you can check "on the go", computing it on access, you don't need to store that information as a separate field. Instead, you can provide a custom getter, whose implementation computes the square-ness of the `Rectangle` every time the property is accessed:

```
class Rectangle(val height: Int, val width: Int) {  
    val isSquare: Boolean  
        get() { #1  
            return height == width  
        }  
}
```

Note that you don't have to use the full syntax with curly braces. Just like any other function, you can define the getter using expression body syntax we learned about in [2.1.3](#), and write `val isSquare get() = height == width`, as well. As you can see, the expression body syntax also allows you to omit explicitly specifying the property type, having the compiler infer the type for you instead.

Regardless of the syntax you choose, the invocation of a property like `isSquare` stays the same:

```
fun main() {  
    val rectangle = Rectangle(41, 43)  
    println(rectangle.isSquare)  
    // false  
}
```

If you need to access this property from Java, you call the `isSquare` method as before.

You might ask whether it's better to declare a property with a custom getter, or define a function inside the class (referred to as a *member function* or *method* in Kotlin). Both options are similar: There is no difference in implementation or performance; they only differ in readability. Generally, if you describe the characteristic (the property) of a class, you should declare it as a property. If you are describing the behavior of a class, choose a member function instead.

In [4](#), you'll take a look at more examples that use classes, properties, and member functions, and also look at the syntax to explicitly declare constructors. If you're impatient and happen to know the equivalents of these topics in Java, you can always use the Java-to-Kotlin converter in the meantime to peek ahead.

Before we move on to discuss other language features, let's briefly examine how code in Kotlin projects is generally structured.

## 2.2.3 Kotlin source code layout: directories and packages

As your programs grow in complexity, consisting of more and more

functions, classes, and other language constructs, you'll inevitably need to start thinking about how to organize your source code in order for your project to stay maintainable and navigable. Let's examine how Kotlin projects are typically structured.

Kotlin has the concept of *packages* to organize classes (similar to what you may be familiar with from Java.) Every Kotlin file can have a package statement at the beginning, and all declarations (classes, functions, and properties) defined in the file will be placed in that package.

Here's an example of a source file showing the syntax for the package declaration statement.

**Listing 2.7. Putting a class and a function declaration in a package**

```
package geometry.shapes #1

class Rectangle(val height: Int, val width: Int) {
    val isSquare: Boolean
        get() = height == width
}

fun createUnitSquare(): Rectangle {
    return Rectangle(1, 1)
}
```

Declarations defined in other files can be used directly if they're in the same package; they need to be *imported* if they're in a different package. This happens using the `import` keyword at the beginning of the file, placed directly below the package directive.

Kotlin doesn't make a distinction between importing classes or functions, and it allows you to import any kind of declaration using the `import` keyword. If you are writing a demo project in the `geometry.example` package, then you can use the class `Rectangle` and the function `createUnitSquare` from the `geometry.shapes` package by simply importing them by name:

**Listing 2.8. Importing the function from another package**

```
package geometry.example
```

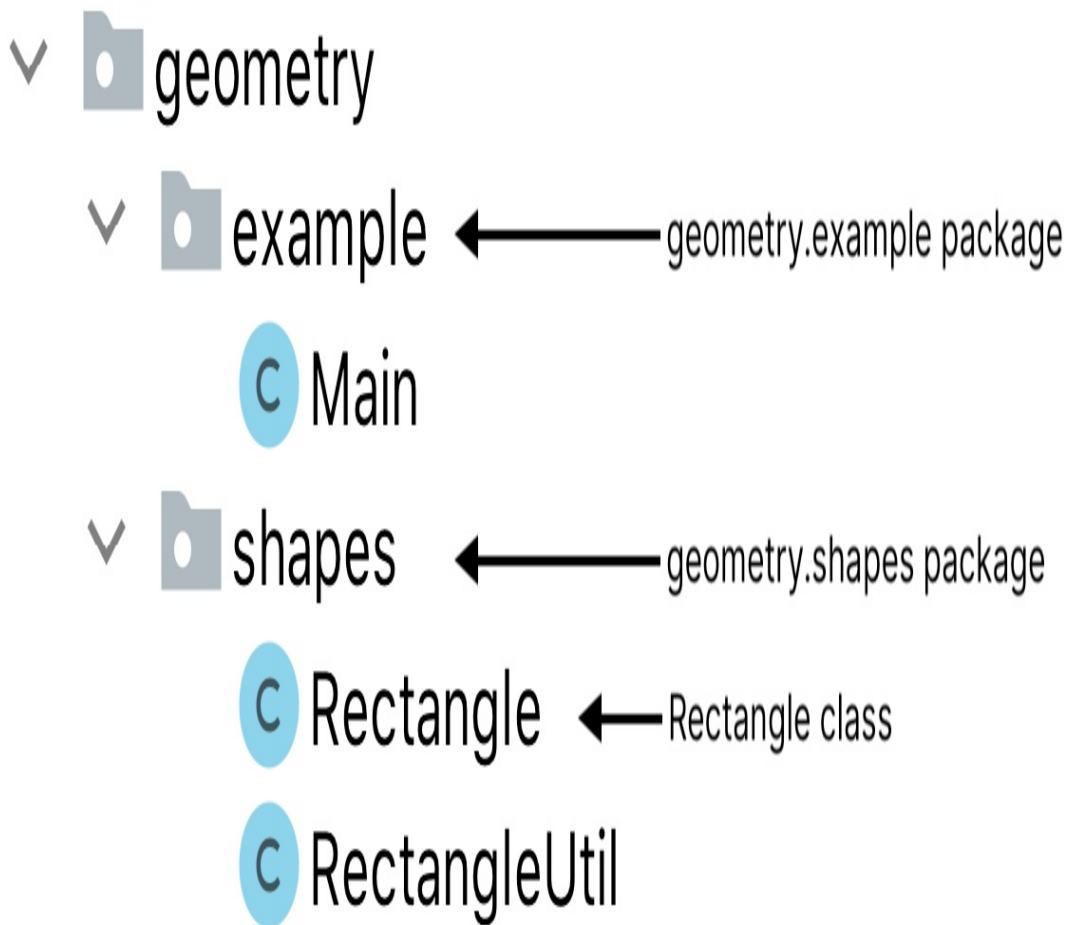
```
import geometry.shapes.Rectangle #1
import geometry.shapes.createUnitSquare #2

fun main() {
    println(Rectangle(3, 4).isSquare)
    // false
    println(createUnitSquare().isSquare)
    // true
}
```

You can also import all declarations defined in a particular package by putting `. *` after the package name. Note that this *star import* (also called *wildcard import*) will make everything defined in the package visible—not only classes, but also top-level functions and properties. In [2.8](#), writing `import geometry.shapes.*` instead of the explicit import makes the code compile correctly as well.

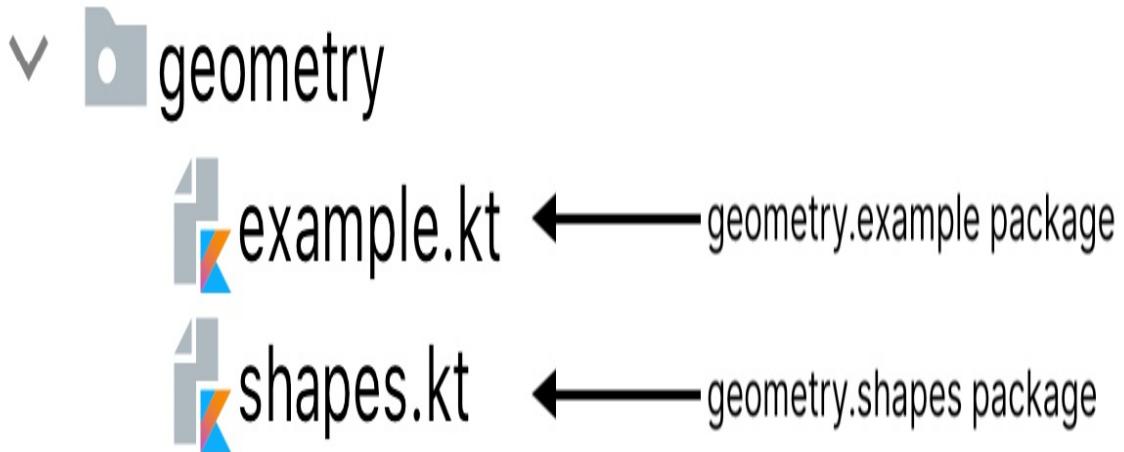
In Java, you put your classes into a structure of files and directories that matches the package structure. For example, if you have a package named `shapes` with several classes, you need to put every class into a separate file with a matching name and store those files in a directory also called `shapes`. [2.3](#) shows how the `geometry` package and its subpackages could be organized in Java. Assume that the `createUnitSquare` function is located in a separate file, `RectangleUtil`.

**Figure 2.3. In Java, the directory hierarchy duplicates the package hierarchy.**



In Kotlin, you can put multiple classes in the same file and choose any name for that file. Kotlin also doesn't impose any restrictions on the layout of source files on disk; you can use any directory structure to organize your files. For instance, you can define all the content of the package `geometry.shapes` in the file `shapes.kt` and place this file in the `geometry` folder without creating a separate `shapes` folder (see [2.4](#)).

**Figure 2.4.** Your package hierarchy doesn't need to follow the directory hierarchy.



In most cases, however, it's still a good practice to follow Java's directory layout and to organize source files into directories according to the package structure. Sticking to that structure is especially important in projects where Kotlin is mixed with Java, because doing so lets you migrate the code gradually without introducing any surprises. But you shouldn't hesitate to pull multiple classes into the same file, especially if the classes are small (and in Kotlin, they often are).

Now you know how programs are structured. Let's return to our journey through the basic concepts of Kotlin, and let's have a look at how to handle different choices in Kotlin beyond the `if` expression.

## 2.3 Representing and handling choices: enums and "when"

In this section, we'll look at an example of declaring enums in Kotlin, and talk about the `when` construct. The latter can be thought of a more powerful and often-used replacement for the `switch` construct in Java. We will also discuss the concept of *smart casts*, which combine type checks and casts.

### 2.3.1 Declaring enum classes and enum constants

It's time we add a splash of color to this book! Given manufacturing constraints, you'll have to rely on your imagination to visualise them in all

their glory—here, you'll have to realize them as Kotlin code, specifically, an enum of color constants:

**Listing 2.9. Declaring a simple enum class**

```
package ch02.colors

enum class Color {
    RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
}
```

This is a rare case when a Kotlin declaration uses more keywords than the corresponding Java one: `enum class` versus just `enum` in Java.

**enum is a soft keyword**

In Kotlin, `enum` is a so-called *soft keyword*: it has a special meaning when it comes before `class`, but you can use it as a regular name, e.g. for a function, variable name, or parameter, in other places. On the other hand, `class` is a *hard keyword*, meaning you can't use it as an identifier, and have to use an alternate spelling or phrasing, like `clazz` or `aClass`.

Having colors stored in an enum is already useful, but we can do better. Color values are often represented using their red, green, and blue components. Enum constants use the same constructor and property declaration syntax as you saw earlier for regular classes. You can use this to expand your `Color` enum: You can associate each enum constant with its `r`, `g`, and `b` values. You can also declare properties, like `rgb` that creates a combined numerical color value from the components, and methods, like `printColor`, using familiar syntax:

**Listing 2.10. Declaring an enum class with properties**

```
package ch02.colors

enum class Color(
    val r: Int, #1
    val g: Int, #1
    val b: Int #1
) {
```

```

    RED(255, 0, 0), #2
    ORANGE(255, 165, 0),
    YELLOW(255, 255, 0),
    GREEN(0, 255, 0),
    BLUE(0, 0, 255),
    INDIGO(75, 0, 130),
    VIOLET(238, 130, 238); #3
    val rgb = (r * 256 + g) * 256 + b #4
    fun printColor() = println("$this is $rgb") #5
}

fun main() {
    println(Color.BLUE.rgb)
    // 255
    Color.GREEN.printColor()
    // GREEN is 65280
}

```

Note that this example shows the only place in the Kotlin syntax where you're required to use semicolons: if you define any methods in the enum class, the semicolon separates the enum constant list from the method definitions.

Now that we have a fully fledged `Colors` enum, let's see how Kotlin lets you easily work with these constants.

### 2.3.2 Using "when" to deal with enum classes

Do you remember how children use mnemonic phrases to memorize the colors of the rainbow? Here's one: "Richard Of York Gave Battle In Vain!" Imagine you need a function that gives you a mnemonic for each color (and you don't want to store this information in the enum itself). In Java, you can use a `switch` statement or, since Java 13, a `switch` expression for this. The corresponding Kotlin construct is `when`.

Like `if`, `when` is an expression that returns a value, so you can write a function with an expression body, returning the `when` expression directly. When we talked about functions at the beginning of the chapter, we promised an example of a multiline function with an expression body. Here's such an example.

**Listing 2.11. Using when for choosing the right enum value**

```
fun getMnemonic(color: Color) = #1
    when (color) { #2
        Color.RED -> "Richard"
        Color.ORANGE -> "Of"
        Color.YELLOW -> "York"
        Color.GREEN -> "Gave"
        Color.BLUE -> "Battle"
        Color.INDIGO -> "In"
        Color.VIOLET -> "Vain"
    }

fun main() {
    println(getMnemonic(Color.BLUE))
    // Battle
}
```

The code finds the branch corresponding to the passed `color` value. Note that you don't need to write `break` statements for each branch. (In Java, a missing `break` in a `switch` statement is often a cause for bugs.) If a match is successful, only the corresponding branch is executed. You can also combine multiple values in the same branch if you separate them with commas.

So, to use different branches based on the "warmth" of the color, you could group your enum constants accordingly in your `when` expression:

**Listing 2.12. Combining options in one when branch**

```
fun measureColor() = Color.ORANGE
// as a stand-in for more complex measurement logic

fun getWarmthFromSensor(): String {
    val color = measureColor()
    return when(color) {
        Color.RED, Color.ORANGE, Color.YELLOW -> "warm (red = ${color.r})"
        Color.GREEN -> "neutral (green = ${color.g})"
        Color.BLUE, Color.INDIGO, Color.VIOLET -> "cold (blue = ${color.b})"
    }
}

fun main() {
    println(getWarmthFromSensor())
    // warm (red = 255)
```

```
}
```

These examples use enum constants by their full name, specifying the `Color` enum class name every time one of the enum's constants is referenced. By importing the constant values, you can simplify this code, and save yourself some repetition:

**Listing 2.13. Importing enum constants to access without qualifier**

```
import ch02.colors.Color #1
import ch02.colors.Color.* #2
fun measureColor() = ORANGE

fun getWarmthFromSensor(): String {
    val color = measureColor()
    return when (color) {
        RED, ORANGE, YELLOW ->
            "warm (red = ${color.r})" #3
        GREEN ->
            "neutral (green = ${color.g})" #3
        BLUE, INDIGO, VIOLET ->
            "cold (blue = ${color.b})" #3
    }
}
```

### 2.3.3 Capturing the subject of a "when" expression in a variable

In the previous examples, the subject of the `when` expression was the `color` variable, which you obtained by invoking the `measureColor()` function. To avoid cluttering the surrounding code with extraneous variables, like `color` in this case, the `when` expression can also capture its subject in a variable. In this case, the captured variable's scope is restricted to the body of the `when` expression, while still providing access to its properties inside the branches of the `when` expression:

**Listing 2.14. Assigning the subject of a when to a variable scoped to the body of the expression**

```
import ch02.colors.Color #1
import ch02.colors.Color.* #2
```

```

fun measureColor() = ORANGE

fun getWarmthFromSensor() =
    when (val color = measureColor()) { #3
        RED, ORANGE, YELLOW -> "warm (red = ${color.r})" #4
        GREEN -> "neutral (green = ${color.g})"
        BLUE, INDIGO, VIOLET -> "cold (blue = ${color.b})"
    }

```

In future examples we'll use the short enum names but omit the explicit imports for simplicity.

Note that whenever `when` is used as an expression (meaning that its result is used in an assignment or as a return value) the compiler enforces the construct to be *exhaustive*. This means that *all possible paths* must return a value.

In the previous example, we cover all enum constants, making the `when` construct exhaustive. Instead, we could also provide a default case using the `else` keyword. In cases where the compiler can't deduce whether all possible paths are covered, it forces us to provide a default case. We'll look at such an example in the next section.

### 2.3.4 Using "when" with arbitrary objects

The `when` construct in Kotlin is actually more flexible than you might be used to from other languages: you can use any kind of object as a branch condition. Let's write a function that mixes two colors if they can be mixed in this small palette. You don't have a lot of options, and you can easily enumerate them all.

**Listing 2.15. Using different objects in `when` branches**

```

fun mix(c1: Color, c2: Color) =
    when (setOf(c1, c2)) { #1
        setOf(RED, YELLOW) -> ORANGE #2
        setOf(YELLOW, BLUE) -> GREEN
        setOf(BLUE, VIOLET) -> INDIGO
        else -> throw Exception("Dirty color") #3
    }

```

```
fun main() {
    println(mix(BLUE, YELLOW))
    // GREEN
}
```

If colors `c1` and `c2` are `RED` and `YELLOW` (or vice versa), the result of mixing them is `ORANGE`, and so on. To implement this, you use set comparison. The Kotlin standard library contains a function `setOf` that creates a `Set` containing the objects specified as its arguments. A *set* is a collection for which the order of items doesn't matter; two sets are equal if they contain the same items. Thus, if the sets `setOf(c1, c2)` and `setOf(RED, YELLOW)` are equal, it means either `c1` is `RED` and `c2` is `YELLOW`, or vice versa. This is exactly what you want to check.

The `when` expression matches its argument against all branches in order until some branch condition is satisfied. Thus `setOf(c1, c2)` is checked for equality: first with `setOf(RED, YELLOW)` and then with other sets of colors, one after another. If none of the other branch conditions is satisfied, the `else` branch is evaluated.

Since the Kotlin compiler can't deduce that we have covered all possible combinations of color sets, and the result of the `when` expression is used as the return value for the `mix` function, we're forced to provide a default case to guarantee that the `when` expression is indeed exhaustive.

Being able to use any expression as a `when` branch condition lets you write concise and beautiful code in many cases. In this example, the condition is an equality check; next you'll see how the condition may be any Boolean expression.

### 2.3.5 Using "when" without an argument

You may have noticed that [2.15](#) is somewhat inefficient. Every time you call this function, it creates several `Set` instances that are used only to check whether two given colors match the other two colors. Normally this isn't an issue, but if the function is called often, it's worth rewriting the code in a different way to avoid creating many short-lived objects which need to be cleaned up by the garbage collector. You can do it by using the `when`

expression without an argument. The code is less readable, but that's the price you often have to pay to achieve better performance.

**Listing 2.16. Using when without an argument**

```
fun mixOptimized(c1: Color, c2: Color) =  
    when { #1  
        (c1 == RED && c2 == YELLOW) ||  
        (c1 == YELLOW && c2 == RED) ->  
            ORANGE  
  
        (c1 == YELLOW && c2 == BLUE) ||  
        (c1 == BLUE && c2 == YELLOW) ->  
            GREEN  
  
        (c1 == BLUE && c2 == VIOLET) ||  
        (c1 == VIOLET && c2 == BLUE) ->  
            INDIGO  
  
        else -> throw Exception("Dirty color") #2  
    }  
  
fun main() {  
    println(mixOptimized(BLUE, YELLOW))  
    // GREEN  
}
```

If no argument is supplied for the `when` expression, the branch condition is any Boolean expression. The `mixOptimized` function does the same thing as `mix` did earlier. Its advantage is that it doesn't create any extra objects, but the cost is that it's harder to read.

Let's move on and look at examples of the `when` construct in which *smart casts* come into play.

### 2.3.6 Smart casts: combining type checks and casts

Now that you've successfully mixed a few colors with Kotlin, let's move on to a bit more complex example. You'll write a function that evaluates simple arithmetic expressions like `(1 + 2) + 4`. The expressions will contain only one type of operation: the sum of two numbers. Other arithmetic operations (subtraction, multiplication, division) can be implemented in a similar way,

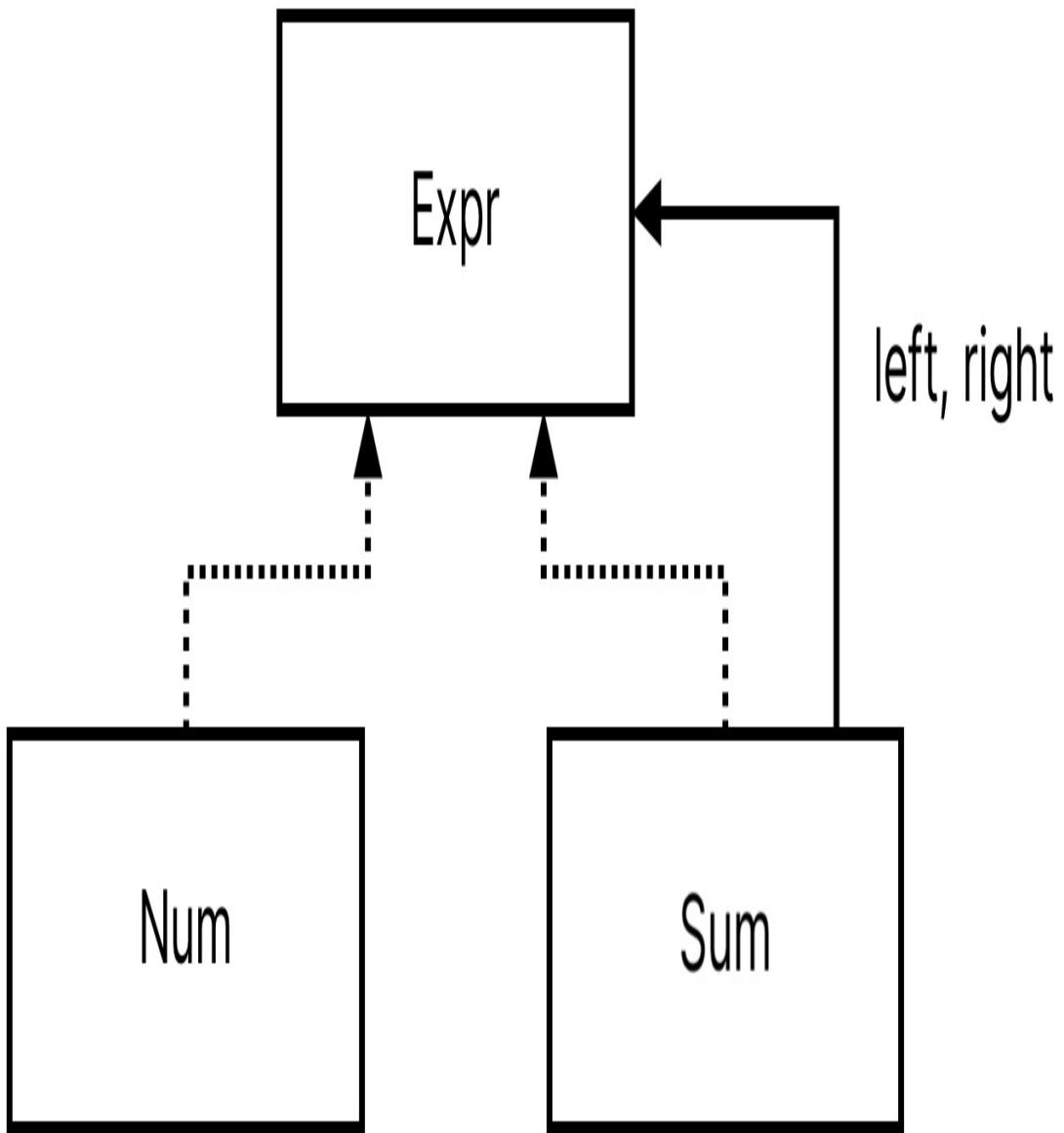
and you can do that as an exercise. In the process, you'll learn about how *smart casts* make it much easier to work with Kotlin objects of different types.

First, how do you encode the expressions? Traditionally, you store them in a tree-like structure, where each node is either a sum (`Sum`) or a number (`Num`). `Num` is always a leaf node, whereas a `Sum` node has two children: the arguments of the `sum` operation. [2.17](#) shows a simple structure of classes used to encode the expressions: an interface called `Expr` and two classes, `Num` and `Sum`, that implement it. Note that the `Expr` interface doesn't declare any methods; it's used as a *marker interface* to provide a common type for different kinds of expressions. To mark that a class implements an interface, you use a colon (:) followed by the interface name (you'll take a closer look at interfaces in [4.1.1](#)):

**Listing 2.17. Expression class hierarchy**

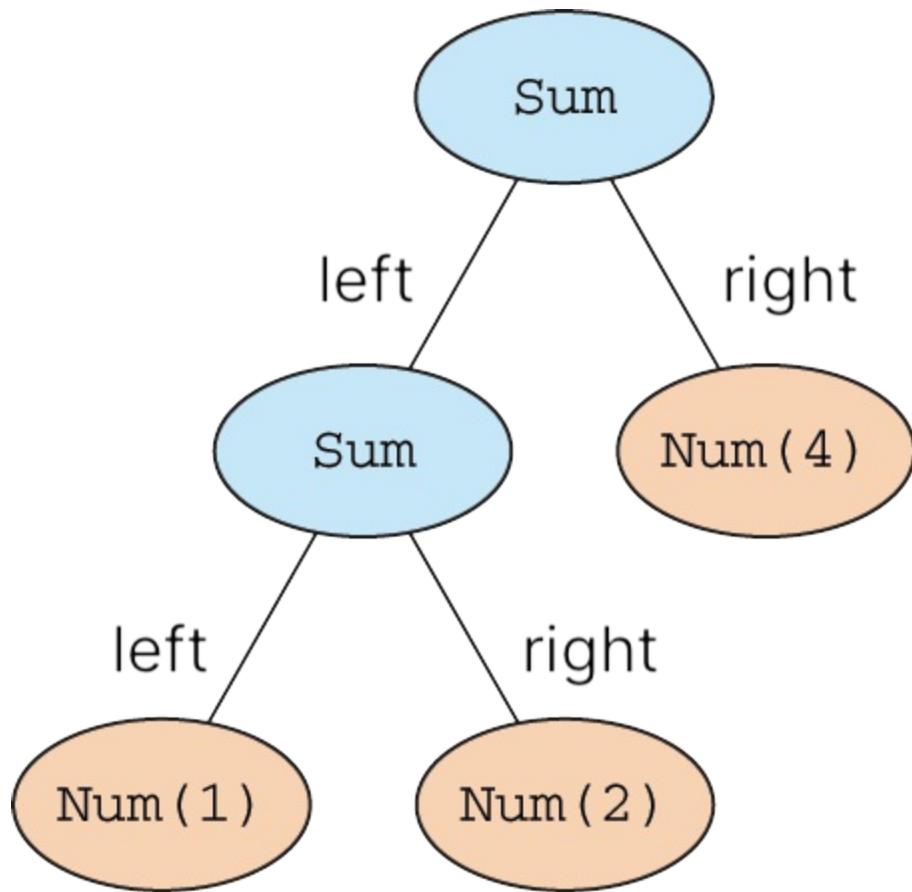
```
interface Expr
class Num(val value: Int) : Expr #1
class Sum(val left: Expr, val right: Expr) : Expr #2
```

**Figure 2.5. Class diagram showing the relationship between `Expr`, `Num` and `Sum`. `Num` and `Sum` both realize the marker interface `Expr`. `Sum` also has an association with the `left` and `right` operands, which are once again of type `Expr`.**



Sum stores references to the `left` and `right` arguments of type Expr. In the case of your example, that means they can be either Num or Sum. To store the expression  $(1 + 2) + 4$  mentioned earlier, you create a structure of Expr objects, specifically `Sum(Sum(Num(1), Num(2)), Num(4))`. [2.6](#) shows its tree representation.

**Figure 2.6. A representation of the expression `Sum(Sum(Num(1), Num(2)), Num(4))` describing the mathematical expression  $(1 + 2) + 4$ . We use this codification as the input for our evaluation function.**



Your goal is to evaluate this kind of expression consisting of `Sum` and `Num` objects, computing the resulting value. Let's take a look at that next.

The `Expr` interface has two implementations, so you have to try two options in order to evaluate a result value for an expression:

1. If an expression is a number, you return the corresponding value.
2. If it's a sum, you have to evaluate the left and right expressions recursively, and return their sum.

First we'll look at an implementation of this function written in a style similar to what you might see in Java code. Then, we'll refactor it to reflect idiomatic Kotlin.

At first, you might write a function reminiscent of the style you may see in other languages, using a sequence of `if` expressions to check the different subtypes of `Expr`. In Kotlin, you check whether a variable is of a certain type by using an `is` check, so an implementation might look like this:

**Listing 2.18. Evaluating expressions with an `if`-cascade**

```
fun eval(e: Expr): Int {  
    if (e is Num) {  
        val n = e as Num #1  
        return n.value  
    }  
    if (e is Sum) {  
        return eval(e.right) + eval(e.left) #2  
    }  
    throw IllegalArgumentException("Unknown expression")  
}  
  
fun main() {  
    println(eval(Sum(Sum(Num(1), Num(2)), Num(4))))  
    // 7  
}
```

The `is` syntax might be familiar to you if you've programmed in C#, and Java developers might recognize it as the equivalent of `instanceof`.

Kotlin's `is` check provides some additional convenience: If you check the variable for a certain type, you don't need to cast it afterward; you can use it as having the type you checked for. In effect, the compiler performs the cast for you: something we call a *smart cast*. (This is more ergonomic than in Java, where after checking the type of a variable, you still need to add an explicit cast.)

In the `eval` function, after you check whether the variable `e` has `Num` type, the compiler smartly interprets it as a variable of type `Num`. You can then access the `value` property of `Num` without an explicit cast: `e.value`. The same goes for the `right` and `left` properties of `Sum`: you write only `e.right` and `e.left` in the corresponding context. In IntelliJ IDEA and Android Studio, these smart-cast values are emphasized with a background color, so it's easy to grasp that this value was checked beforehand (see [2.7.](#))

Figure 2.7. The IDE highlights smart casts with a background color.

```
if (e is Sum) {  
    return eval(e.right) + eval(e.left)  
}
```

The smart cast works only if a variable couldn't have changed after the `is` check. When you're using a smart cast with a property of a class, as in this example, the property has to be a `val` and it can't have a custom accessor. Otherwise, it would not be possible to verify that every access to the property would return the same value.

An explicit cast to the specific type is expressed via the `as` keyword:

```
val n = e as Num
```

But, as you may have guessed, this implementation isn't yet considered idiomatic Kotlin. Let's look at how to refactor the `eval` function.

### 2.3.7 Refactoring: replacing "if" with "when"

In [2.1.2](#), you have already seen that `if` is an *expression* in Kotlin. This is also why there is no ternary operator in Kotlin—the `if` expression can already return a value.

That means you can rewrite the `eval` function to use the expression-body syntax, removing the `return` statement and the curly braces and using the `if` expression as the function body instead.

#### **Listing 2.19. Using if-expressions that return values**

```
fun eval(e: Expr): Int =  
    if (e is Num) {
```

```

        e.value
    } else if (e is Sum) {
        eval(e.right) + eval(e.left)
    } else {
        throw IllegalArgumentException("Unknown expression")
    }

fun main() {
    println(eval(Sum(Num(1), Num(2))))
    // 3
}

```

And you can make this code even more concise: The curly braces are optional if there's only one expression in an `if` branch—for an `if` branch with a block, the last expression is returned as a result. A shortened version of the `eval` expression using cascading `if` expressions could look like this:

```

fun eval(e: Expr): Int =
    if (e is Num) e.value
    else if (e is Sum) eval(e.right) + eval(e.left)
    else throw IllegalArgumentException("Unknown expression")

```

But you've already gotten to know an even better language construct for expressing multiple choices in [2.3.2](#)—let's polish this code even more and rewrite it using `when`.

The `when` expression isn't restricted to checking values for equality, which is what you saw earlier. Here you use a different form of `when` branches, allowing you to check the type of the `when` argument value. Just as in the `if` example in [2.19](#), the type check applies a smart cast, so you can access members of `Num` and `Sum` without extra casts:

#### **Listing 2.20. Using `when` instead of `if`-cascade**

```

fun eval(e: Expr): Int =
    when (e) {
        is Num -> e.value #1
        is Sum -> eval(e.right) + eval(e.left) #1
        else -> throw IllegalArgumentException("Unknown expr")
    }

```

Compare the last two Kotlin versions of the `eval` function, and think about

how you can apply when as a replacement for sequences of if expressions in your own code as well. For branch logic containing multiple operations, you can use a block expression as a branch body. Let's see how this works.

### 2.3.8 Blocks as branches of "if" and "when"

Both if and when can have blocks as branches. In this case, the last expression in the block is the result. Let's say you want to gain a deeper understanding of how your eval function computes the result. One way to do so is to add some println statements that log what the function is currently calculating. You can add them in the block for each branch in your when expression. The last expression in the block is what will be returned:

**Listing 2.21. Using when with compound actions in branches**

```
fun evalWithLogging(e: Expr): Int =  
    when (e) {  
        is Num -> {  
            println("num: ${e.value}")  
            e.value #1  
        }  
        is Sum -> {  
            val left = evalWithLogging(e.left)  
            val right = evalWithLogging(e.right)  
            println("sum: $left + $right")  
            left + right #2  
        }  
        else -> throw IllegalArgumentException("Unknown expression")  
    }
```

Now you can look at the logs printed by the evalWithLogging function and follow the order of computation:

```
fun main() {  
    println(evalWithLogging(Sum(Sum(Num(1), Num(2)), Num(4))))  
    // num: 1  
    // num: 2  
    // sum: 1 + 2  
    // num: 4  
    // sum: 3 + 4  
    // 7  
}
```

The rule "the last expression in a block is the result" holds in all cases where a block can be used and a result is expected. As you'll see in [2.5.2](#), the same rule works for the `try` body and `catch` clauses, and in [5.1.3](#), we'll discuss the application of this rule to lambda expressions. But as we mentioned in section [2.1.3](#), this rule doesn't hold for regular functions. A function can have either an expression body that can't be a block or a block body with explicit `return` statements inside.

By now, you've seen multiple ways of how you can choose the right things among many in your Kotlin code, so it seems now would be a good time to see how you can iterate over things.

## 2.4 Iterating over things: "while" and "for" loops

Iteration in Kotlin is very similar to what you are probably used to from Java, C#, or other languages. The `while` loop takes the same traditional form it does in other languages, so you'll only take a brief look at it. You'll also find the `for` loop, which is written `for (<item> in <elements>)`, to be reminiscent of Java's `for-each` loop, for example. Let's explore what kind of looping scenarios you can cover with these two forms of loops.

### 2.4.1 Repeating code while a condition is true: the "while" loop

Kotlin has `while` and `do-while` loops, and their syntax are probably familiar to you from other programming languages. Let's briefly review it:

```
while (condition) { #1
    /*...*/
    if(shouldExit) break #2
}

do {
    if(shouldSkip) continue #3
    /*...*/
} while (condition) #4
```

For nested loops, Kotlin allows you to specify a *label*, which you can then reference when using `break` or `continue`. A label is an identifier followed by the `@` sign:

```

outer@ while (outerCondition) { #1
    while (innerCondition) {
        if (shouldExitInner) break #2
        if (shouldSkipInner) continue #2
        if (shouldExit) break@outer #3
        if (shouldSkip) continue@outer #3
        // ...
    }
    // ...
}

```

Let's move on to discuss the various uses of the `for` loop, and see how it covers not just the iteration over collection items, but over *ranges* as well.

## 2.4.2 Iterating over numbers: ranges and progressions

As we just mentioned, in Kotlin there's no C-style `for` loop, where you initialize a variable, update its value on every step through the loop, and exit the loop when the value reaches a certain bound (the classical `int i = 0; i < 10; i++`). To replace the most common use cases of such loops, Kotlin uses the concepts of *ranges*.

A range is essentially just an interval between two values, usually numbers: a start and an end. You write it using the `..` operator:

```
val oneToTen = 1..10
```

Note that these ranges in Kotlin are *closed* or *inclusive*, meaning the second value is always a part of the range.

The most basic thing you can do with integer ranges is loop over all the values. If you can iterate over all the values in a range, such a range is called a *progression*.

Let's use integer ranges to play the Fizz-Buzz game. It's a nice way to survive a long trip in a car and remember your forgotten division skills. Implementing this game is also a popular task for programming interviews!

To play Fizz-Buzz, players take turns counting incrementally, replacing any number divisible by three with the word *fizz* and any number divisible by five

with the word *buzz*. If a number is a multiple of both three and five, you say "FizzBuzz."

[2.22](#) prints the right answers for the numbers from 1 to 100. Note how you check the possible conditions with a `when` expression without an argument.

#### **Listing 2.22. Using `when` to implement the Fizz-Buzz game**

```
fun fizzBuzz(i: Int) = when {
    i % 15 == 0 -> "FizzBuzz" #1
    i % 3 == 0 -> "Fizz" #2
    i % 5 == 0 -> "Buzz" #3
    else -> "$i" #4
}

fun main() {
    for (i in 1..100) { #5
        print(fizzBuzz(i))
    }
    // 1 2 Fizz 4 Buzz Fizz 7 ...
}
```

Suppose you get tired of these rules after an hour of driving and want to complicate things a bit. Let's start counting backward from 100 and include only even numbers.

#### **Listing 2.23. Iterating over a range with a step**

```
fun main() {
    for (i in 100 downTo 1 step 2) {
        print(fizzBuzz(i))
    }
    // Buzz 98 Fizz 94 92 FizzBuzz 88 ...
}
```

Now you're iterating over a progression that has a *step*, which allows it to skip some numbers. The step can also be negative, in which case the progression goes backward rather than forward. In this example, `100 downTo 1` is a progression that goes backward (with step `-1`). Then `step` changes the absolute value of the step to 2 while keeping the direction (in effect, setting the step to `-2`).

As we mentioned earlier, the `..` syntax always creates a range that includes the end point (the value to the right of `..`). In many cases, it's more convenient to iterate over half-closed ranges, which don't include the specified end point. To create such a range, use `... For example, the loop for (x in 0..<size) is equivalent to for (x in 0..size-1), but it expresses the idea somewhat more clearly. Later, in section 3.4, you'll learn more about the syntax for downTo, step in these examples.`

You can see how working with ranges and progressions helped you cope with the advanced rules for the FizzBuzz game. But the `for` loop in Kotlin can do more than that. Let's look at some other examples:

### 2.4.3 Iterating over maps

We've mentioned that the most common scenario of using a `for (x in y)` loop is iterating over a collection. You are most likely already familiar with its behavior—the loop is executed for each element in the input collection. In this case, you simply print each element from the collection of colors. Inside the loop, the individual colors can be addressed with `color`, since that is the name used in the `for` loop:

**Listing 2.24. Iterating over a range with a step**

```
fun main() {
    val collection = listOf("red", "green", "blue")
    for(color in collection) {
        print("$color ")
    }
    // red green blue
}
```

Instead of spending more time on this, let's see something more interesting instead: how you can iterate over a map.

As an example, we'll look at a small program that prints binary representations for characters—providing you with a simple look-up table that will help you decipher binary-encoded text like `1000100 1000101 1000011 1000001 1000110` by hand! You'll store these binary representations in a map (just for illustrative purposes).

The following code creates a map, fills it with binary representations of some letters, and then prints the map's contents. As you can see, the `..` syntax to create a range works not only for numbers, but also for characters. Here you use it to iterate over all characters from A up to and including F:

**Listing 2.25. Initializing and iterating over a map**

```
fun main() {
    val binaryReps = mutableMapOf<Char, String>() #1
    for (char in 'A'..'F') { #2
        val binary = char.code.toString(radix = 2) #3
        binaryReps[char] = binary #4
    }

    for ((letter, binary) in binaryReps) { #5
        println("$letter = $binary")
    }
    // A = 1000001      D = 1000100
    // B = 1000010      E = 1000101
    // C = 1000011      F = 1000110
    // (output split into columns for conciseness)
}
```

[2.25](#) shows that the `for` loop allows you to unpack an element of a collection you're iterating over (in this case, a collection of key/value pairs in the map). You store the result of the unpacking in two separate variables: `letter` receives the key, and `binary` receives the value. Later, in section [9.4](#), you'll find out more about this destructuring syntax.

Another nice trick used in [2.25](#) is the shorthand syntax for getting and updating the values of a map by key. Instead of having to call functions like `get` and `put`, you can use `map[key]` to read values and `map[key] = value` to set them. That means instead of having to use the Java-style version of `binaryReps.put(char, binary)`, you can use the equivalent, but more elegant `binaryReps[char] = binary`.

You can use the same unpacking syntax to iterate over a collection while keeping track of the index of the current item. This lets you avoid creating a separate variable to store the index and incrementing it by hand. In this case, you're printing the elements of a collection with their respective index using the `withIndex` function:

```

fun main() {
    val list = listOf("10", "11", "1001")
    for ((index, element) in list.withIndex()) { #1
        println("$index: $element")
    }
    // 0: 10
    // 1: 11
    // 2: 1001
}

```

We'll dig into the whereabouts of `withIndex` in [3.3](#).

You've seen how you can use the `in` keyword to iterate over a range or a collection. Beyond that, you can also use `in` to check whether a value belongs to the range or collection. Let's take a closer look.

## 2.4.4 Using "in" to check collection and range membership

You use the `in` operator to check whether a value is in a range, or its opposite, `!in`, to check whether a value isn't in a range. For example, when validating the input of a user, you often have to check that an input character is indeed a letter, or excludes digits. Here's how you could use `in` to write some small helper functions `isLetter` and `isNotDigit` that check whether a character belongs to a range of characters:

**Listing 2.26. Checking range membership using `in`**

```

fun isLetter(c: Char) = c in 'a'..'z' || c in 'A'..'Z'
fun isNotDigit(c: Char) = c !in '0'..'9'

fun main() {
    println(isLetter('q'))
    // true
    println(isNotDigit('x'))
    // true
}

```

This technique for checking whether a character is a letter looks simple. Under the hood, nothing tricky happens: you still check that the character's code is somewhere between the code of the first letter and the code of the last one. But this logic is concisely hidden in the implementation of the range

classes in the standard library:

```
c in 'a'..'z' #1
```

The `in` and `!in` operators also work in `when` expressions, which becomes extra convenient when you have a number of different ranges that you want to check:

**Listing 2.27. Using `in` checks as `when` branches**

```
fun recognize(c: Char) = when (c) {
    in '0'..'9' -> "It's a digit!" #1
    in 'a'..'z', in 'A'..'Z' -> "It's a letter!" #2
    else -> "I don't know..."
}

fun main() {
    println(recognize('8'))
    // It's a digit!
}
```

Ranges aren't restricted to characters, either. If you have any class that supports comparing instances (by implementing the `kotlin.Comparable` interface that you'll learn more about in [9.2.2](#)), you can create ranges of objects of that type. If you have such a range, you can't enumerate all objects in the range. Think about it: can you, for example, enumerate all strings between "Java" and "Kotlin"? No, you can't. But you can still check whether another object belongs to the range, using the `in` operator:

```
fun main() {
    println("Kotlin" in "Java".."Scala") #1
    // true
}
```

Note that the strings are compared alphabetically here, because that's how the `String` class implements the `Comparable` interface: In alphabetical sorting, "Java" comes before "Kotlin", and "Kotlin" comes before "Scala", so "Kotlin" is in the range between the two strings.

The same `in` check works with collections as well:

```
fun main() {
```

```
    println("Kotlin" in setOf("Java", "Scala")) #1
    // false
}
```

Later, in section [9.3.2](#), you'll see how to use ranges and progressions with your own data types and what objects in general you can use in checks with.

To round out our overview of basic building blocks of Kotlin programs, there's one more topic we want to look at in this chapter: dealing with exceptions.

## 2.5 Throwing and catching exceptions in Kotlin

Exception handling in Kotlin is similar to the way it's done in Java and many other languages. A function can complete in a normal way or throw an exception if an error occurs. The function caller can catch this exception and process it; if it doesn't, the exception is rethrown further up the stack.

You throw an exception using the `throw` keyword—in this case, to indicate that the calling function has provided an invalid percentage value:

```
if (percentage !in 0..100) {
    throw IllegalArgumentException(
        "A percentage value must be between 0 and 100: $percentag
    )
}
```

This is a good point to remind yourself that Kotlin doesn't have a new keyword. Creating an exception instance is no different.

In Kotlin the `throw` construct is an *expression*, and can be used as a part of other expressions:

```
val percentage =
    if (number in 0..100)
        number
    else
        throw IllegalArgumentException( #1
            "A percentage value must be between 0 and 100: $numbe
        )
```

In this example, if the condition is satisfied, the program behaves correctly, and the percentage variable is initialized with number. Otherwise, an exception is thrown, and the variable isn't initialized. We'll discuss the technical details of throw as a part of other expressions, in section [8.1.7](#), where we'll also discover more about its return type, among other things.

## 2.5.1 Handling exceptions and recovering from errors: "try", "catch", and "finally"

If you're on the other side—trying to recover from errors, rather than throw them—you use the try construct with catch and finally clauses to handle exceptions. You can see it in [2.28](#), which reads a line from the given file, tries to parse it as a number, and returns either the number or null if the line isn't a valid number.

**Listing 2.28. Using try the same way you would in Java**

```
import java.io.BufferedReader
import java.io.StringReader

fun readNumber(reader: BufferedReader): Int? { #1
    try {
        val line = reader.readLine()
        return Integer.parseInt(line)
    } catch (e: NumberFormatException) { #2
        return null
    } finally { #3
        reader.close()
    }
}

fun main() {
    val reader = BufferedReader(StringReader("239"))
    println(readNumber(reader))
    // 239
}
```

An important difference from Java is Kotlin doesn't have a throws clause: if you wrote this function in Java, you'd explicitly write throws IOException after the function declaration:

**Listing 2.29. In Java, checked exceptions are part of the method signature.**

```
Integer readNumber(BufferedReader reader) throws IOException
```

You'd need to do this because Java's `readLine` and `close` may throw an `IOException`, which is a *checked exception*. In the Java world, this describes a type of exception that needs to be handled explicitly. You have to declare all checked exceptions that your function can throw, and if you call another function, you need to handle its checked exceptions or declare that your function can throw them, too.

Just like many other modern JVM languages, Kotlin doesn't differentiate between checked and unchecked exceptions. You don't specify the exceptions thrown by a function, and you may or may not handle any exceptions.

This design decision is based on the practice of using checked exceptions in Java. Experience has shown that the Java rules often require a lot of meaningless code to rethrow or ignore exceptions, and the rules don't consistently protect you from the errors that can happen.

For example, in [2.28](#), `NumberFormatException` isn't a checked exception. Therefore, the Java compiler doesn't force you to catch it, and you can easily see the exception happen at runtime. This is unfortunate, because invalid input data is a common situation and should be handled gracefully. At the same time, the `BufferedReader.close` method can throw an `IOException`, which is a checked exception and needs to be handled. Most programs can't take any meaningful action if closing a stream fails, so the code required to catch the exception from the `close` method is boilerplate.

As a result of this design decision, you get to decide yourself which exceptions you want and don't want to handle. If you wanted to, you could implement the `readNumber` function without any `try-catch` constructs at all:

**Listing 2.30. In Kotlin, the compiler does not force you to handle exceptions:**

```
fun readNumber(reader: BufferedReader): Int {  
    val line = reader.readLine()  
    reader.close()
```

```
        return Integer.parseInt(line)
    }
```

What about Java 7's try-with-resources? Kotlin doesn't have any special syntax for this; it's implemented as a library function. In **Chapter 10**, you'll see how this is possible.

## 2.5.2 "try" as an expression

So far, you've only seen the `try` construct used as a statement. But since `try` is an *expression* (just like `if` and `when`), you can modify your example a little to take advantage of that, and assign the value of your `try` expression to a variable. For brevity, let's remove the `finally` section (only because you've already seen how this works—don't use it as an excuse to not close your streams!) and add some code to print the number you read from the file.

**Listing 2.31. Using try as an expression**

```
fun readNumber(reader: BufferedReader) {
    val number = try {
        Integer.parseInt(reader.readLine()) #1
    } catch (e: NumberFormatException) {
        return
    }
    println(number)
}

fun main() {
    val reader = BufferedReader(StringReader("not a number"))
    readNumber(reader) #2
}
```

It's worth pointing out that unlike with `if`, you always need to enclose the statement body in curly braces. Just as in other statements, if the body contains multiple expressions, the value of the `try` expression as a whole is the value of the last expression.

This example puts a `return` statement in the `catch` block, so the execution of the function doesn't continue after the `catch` block. If you want to continue execution, the `catch` clause also needs to have a value, which will be the

value of the last expression in it. Here's how this works.

#### **Listing 2.32. Returning a value in catch**

```
fun readNumber(reader: BufferedReader) {
    val number = try {
        Integer.parseInt(reader.readLine()) #1
    } catch (e: NumberFormatException) {
        null #2
    }

    println(number)
}

fun main() {
    val reader = BufferedReader(StringReader("not a number"))
    readNumber(reader)
    // null #3
}
```

If the execution of a `try` code block behaves normally, the last expression in the block is the result. If an exception is caught, the last expression in a corresponding `catch` block is the result. In [2.32](#), the result value is `null` if a `NumberFormatException` is caught.

Using `try` as an expression can help you make your code a bit more concise by avoiding the introduction of additional intermediate variables, and allows you to easily assign fallback values or return from the enclosing function outright.

At this point, if you're impatient, you can start writing programs in Kotlin by combining the basic building blocks you've seen so far. As you read this book, you'll continue to learn how to change your habitual ways of thinking and use the full power of the Kotlin!

## **2.6 Summary**

- The `fun` keyword is used to declare a function. The `val` and `var` keywords declare read-only and mutable variables, respectively.
- String templates help you avoid noisy string concatenation. Prefix a

variable name with \$ or surround an expression with \${} to have its value injected into the string.

- Classes can be expressed in a concise way in Kotlin.
- The familiar if is now an expression with a return value.
- The when expression is analogous to switch in Java but is more powerful.
- You don't have to cast a variable explicitly after checking that it has a certain type: the compiler casts it for you automatically using a smart cast.
- The for, while, and do-while loops are similar to their counterparts in Java, but the for loop is now more convenient, especially when you need to iterate over a map or a collection with an index.
- The concise syntax 1..5 creates a range. Ranges and progressions allow Kotlin to use a uniform syntax and set of abstractions in for loops and also work with the in and !in operators that check whether a value belongs to a range.
- Exception handling in Kotlin is very similar to that in Java, except that Kotlin doesn't require you to declare the exceptions that can be thrown by a function.

# 3 Defining and calling functions

## This chapter covers

- Functions for working with collections, strings, and regular expressions
- Using named arguments, default parameter values, and the infix call syntax
- Adapting Java libraries to Kotlin through extension functions and properties
- Structuring code with top-level and local functions and properties

By now, you should be fairly comfortable with using Kotlin on a basic level, the same way you might have used other object-oriented languages like Java before. You've seen how the concepts familiar to you from Java translate to Kotlin, and how Kotlin often makes them more concise and readable.

In this chapter, you'll see how Kotlin improves on one of the key elements of every program: declaring and calling functions. We'll also look into the possibilities for adapting Java libraries to the Kotlin style through the use of extension functions, allowing you to gain the full benefits of Kotlin in mixed-language projects.

To make our discussion more useful and less abstract, we'll focus on Kotlin collections, strings, and regular expressions as our problem domain. As an introduction, let's look at how to create collections in Kotlin.

## 3.1 Creating collections in Kotlin

Before you can do interesting things with collections, you need to learn how to create them. In [2.3.4](#), you bumped into the way to create a new set: the `setOf` function. You created a set of colors then, but for now, let's keep it simple and work with numbers:

```
val set = setOf(1, 7, 53)
```

You create a list or a map in a similar way:

```
val list = listOf(1, 7, 53)
val map = mapOf(1 to "one", 7 to "seven", 53 to "fifty-three")
```

Note that `to` isn't a special construct, but a normal function. We'll return to it later in the chapter, in [8.2](#).

Can you guess the classes of objects that are created here? Run the following example to see this for yourself:

```
fun main() {
    println(set.javaClass) #1
    // class java.util.LinkedHashSet

    println(list.javaClass)
    // class java.util.Arrays$ArrayList

    println(map.javaClass)
    // class java.util.LinkedHashMap
}
```

As you can see, Kotlin uses the standard Java collection classes. This is good news for Java developers: Kotlin doesn't re-implement collection classes. All of your existing knowledge about Java collections still applies here. It is worth noting however that unlike in Java, Kotlin's collection interfaces are read-only by default. We will further details on this topic, as well as the mutable counterparts for these interfaces, in [8.2](#).

Using the standard Java collections makes it much easier to interact with Java code. You don't need to convert collections one way or the other when you call Java functions from Kotlin or vice versa.

Even though Kotlin's collections are exactly the same classes as Java collections, you can do much more with them in Kotlin. For example, you can get the last element in a list or sum up a collection (given that it's a collection of numbers):

```
fun main() {
    val strings = listOf("first", "second", "fourteenth")
    println(strings.last())
    // fourteenth
```

```
val numbers = setOf(1, 14, 2)
println(numbers.sum())
// 17
}
```

In this chapter, we'll explore in detail how this works and where all the new methods on the Java classes come from.

In future chapters, when we start talking about lambdas, you'll see much more that you can do with collections, but we'll keep using the same standard Java collection classes. And in [8.2](#), you'll learn how the Java collection classes are represented in the Kotlin type system.

Before discussing how the magic functions `last` and `sum` work on Java collections, let's learn some new concepts for declaring a function.

## 3.2 Making functions easier to call

Now that you know how to create a collection of elements, let's do something straightforward: print its contents. Don't worry if this seems overly simple; along the way, you'll meet a bunch of important concepts.

Java collections have a default `toString` implementation, but the formatting of the output is fixed and not always what you need:

```
fun main() {
    val list = listOf(1, 2, 3)
    println(list) #1
    // [1, 2, 3]
}
```

Imagine that you need the elements to be separated by semicolons and surrounded by parentheses, instead of the brackets used by the default implementation: `(1; 2; 3)`. To solve this, Java projects use third-party libraries such as Guava and Apache Commons, or reimplement the logic inside the project. In Kotlin, a function to handle this is part of the standard library.

In this section, you'll implement this function yourself. You'll begin with a

straightforward implementation that doesn't use Kotlin's facilities for simplifying function declarations, and then you'll rewrite it in a more idiomatic style.

The `joinToString` function shown next appends the elements of the collection to a `StringBuilder`, with a separator between them, a prefix at the beginning, and a postfix at the end. The function is generic: it works on collections that contain elements of any type. The syntax for generics is similar to Java. (A more detailed discussion of generics will be the subject of [Chapter 11](#).)

**Listing 3.1. Initial implementation of `joinToString()`**

```
fun <T> joinToString(
    collection: Collection<T>,
    separator: String,
    prefix: String,
    postfix: String
): String {
    val result = StringBuilder(prefix)
    for ((index, element) in collection.withIndex()) {
        if (index > 0) result.append(separator) #1
        result.append(element)
    }
    result.append(postfix)
    return result.toString()
}
```

Let's verify that the function works as intended:

```
fun main() {
    val list = listOf(1, 2, 3)
    println(joinToString(list, "; ", "(", ")"))
    // (1; 2; 3)
}
```

The implementation is fine, and you'll mostly leave it as is. What we'll focus on is the declaration: how can you change it to make calls of this function less verbose? Maybe you could avoid having to pass four arguments every

time you call the function. Let's see what you can do.

### 3.2.1 Named arguments

The first problem we'll address concerns the readability of function calls. For example, look at the following call of `joinToString`:

```
joinToString(collection, " ", " ", ".")
```

Can you tell what parameters all these `String`s correspond to? Are the elements separated by the whitespace or the dot? These questions are hard to answer without looking at the signature of the function. Maybe you remember it, or maybe your IDE can help you, but it's not obvious from the calling code.

This problem is especially common with Boolean flags. To solve it, some Java coding styles recommend creating enum types instead of using Booleans. Others even require you to specify the parameter names explicitly in a comment, as in the following example:

```
/* Java */
joinToString(collection, /* separator */ " ", /* prefix */ " ",
             /* postfix */ ".");
```

With Kotlin, you can do better:

```
joinToString(collection, separator = " ", prefix = " ", postfix =
```

When calling a function written in Kotlin, you can specify the names of some of the arguments that you're passing to the function. If you specify the names of all arguments passed to the function, you can even change their order:

```
joinToString(
    postfix = ".",
    separator = " ",
    collection = collection,
    prefix = " "
)
```



Tip

IntelliJ IDEA and Android Studio can keep explicitly written argument names up to date if you rename the parameter of the function being called. Just ensure that you use the "Rename" or "Change Signature" action instead of editing the parameter names by hand. Both actions can be found by right-clicking on the function name and choosing the "Refactor" option.

Named arguments work especially well with default parameter values, which we'll look at next.

### 3.2.2 Default parameter values

Another common Java problem is the overabundance of overloaded methods in some classes. Just look at `java.lang.Thread` (<http://mng.bz/4KZC>) and its eight constructors! The overloads can be provided for the sake of backward compatibility, for convenience of API users, or for other reasons, but the end result is the same: duplication. The parameter names and types are repeated over and over, and if you want to be thorough, you also have to repeat most of the documentation in every overload. At the same time, if you call an overload that omits some parameters, it's not always clear which values are used for them.

In Kotlin, you can often avoid creating overloads because you can specify default values for parameters in a function declaration. Let's use that to improve the `joinToString` function. For most cases, the strings can be separated by commas without any prefix or postfix. So, let's make these values the defaults.

**Listing 3.2. Declaring `joinToString()` with default parameter values**

```
fun <T> joinToString(
    collection: Collection<T>,
    separator: String = ", ", #1
    prefix: String = "", #1
    postfix: String = "" #1
): String
```

Now you can either invoke the function with all the arguments or omit some of them:

```
fun main() {
    joinToString(list, ", ", "", "")
    // 1, 2, 3
    joinToString(list)
    // 1, 2, 3
    joinToString(list, "; ")
    // 1; 2; 3
}
```

When using the regular call syntax, you have to specify the arguments in the same order as in the function declaration, and you can omit only trailing arguments. If you use named arguments, you can omit some arguments from the middle of the list and specify only the ones you need, in any order you want:

```
fun main() {
    joinToString(list, suffix = ";", prefix = "# ")
    // # 1, 2, 3;
}
```

Note that the default values of the parameters are encoded in the function being called, not at the call site. If you change the default value and recompile the class containing the function, the callers that haven't specified a value for the parameter will start using the new default value.

### Default values and Java

Given that Java doesn't have the concept of default parameter values, you have to specify all the parameter values explicitly when you call a Kotlin function with default parameter values from Java. If you frequently need to call a function from Java and want to make it easier to use for Java callers, you can annotate it with `@JvmOverloads`. This instructs the compiler to generate Java overloaded methods, omitting each of the parameters one by one, starting from the last one.

For example, you may annotate your `joinToString` function with `@JvmOverloads`:

#### **Listing 3.3. Declaring `joinToString()` with default parameter values**

`@JvmOverloads`

```
fun <T> joinToString(  
    collection: Collection<T>,  
    separator: String = ", ",  
    prefix: String = "",  
    postfix: String = ""  
) : String { /* ... */ }
```

This means the following overloads are generated:

```
/* Java */  
String joinToString(Collection<T> collection, String separator,  
    String prefix, String postfix);  
  
String joinToString(Collection<T> collection, String separator,  
    String prefix);  
  
String joinToString(Collection<T> collection, String separator);  
  
String joinToString(Collection<T> collection);
```

Each overload uses the default values for the parameters that have been omitted from the signature.

So far, you've been working on your utility function without paying much attention to the surrounding context. Surely it must have been a method of some class that wasn't shown in the example listings, right? In fact, Kotlin makes this unnecessary.

### 3.2.3 Getting rid of static utility classes: top-level functions and properties

We all know that Java, as an object-oriented language, requires all code to be written as methods of classes. Usually, this works out nicely; but in reality, almost every large project ends up with a lot of code that doesn't clearly belong to any single class. Sometimes an operation works with objects of two different classes that play an equally important role for it. Sometimes there is one primary object, but you don't want to bloat its API by adding the operation as an instance method.

As a result, you end up with classes that don't contain any state or any instance methods. Such classes only act as containers for a bunch of static

methods. A perfect example is the `Collections` class in the JDK. To find other examples in your own code, look for classes that have `Util` as part of the name.

In Kotlin, you don't need to create all those meaningless classes. Instead, you can place functions directly at the top level of a source file, outside of any class. Such functions are still members of the package declared at the top of the file, and you still need to import them if you want to call them from other packages, but the unnecessary extra level of nesting no longer exists.

Let's put the `joinToString` function into the `strings` package directly. Create a file called `join.kt` with the following contents.

**Listing 3.4. Declaring `joinToString()` as a top-level function**

```
package strings

fun joinToString(...): String { ... }
```

How does this run? When you compile the file, some classes will be produced, because the JVM can only execute code in classes. When you work only with Kotlin, that's all you need to know. But if you need to call such a function from Java, you have to understand how it will be compiled. To make this clear, let's look at the Java code that would compile to the same class:

```
/* Java */
package strings;

public class JoinKt { #1
    public static String joinToString(...) { ... }
}
```

You can see that the name of the class generated by the Kotlin compiler corresponds to the name of the file containing the function—capitalized to match Java's naming scheme, and suffixed with `Kt`. All top-level functions in the file are compiled to static methods of that class. Therefore, calling this function from Java is as easy as calling any other static method:

```
/* Java */
```

```
import strings.JoinKt;  
...  
JoinKt.joinToString(list, ", ", "", "");
```

### Changing the file class name

By default, the class name generated by the compiler corresponds to the file name, together with a "Kt" suffix. To change the name of the generated class that contains Kotlin top-level functions, you add a `@JvmName` annotation to the file. Place it at the beginning of the file, before the package name:

```
@file:JvmName("StringFunctions") #1  
package strings #2  
fun joinToString(...): String { ... }
```

Now the function can be called as follows:

```
/* Java */  
import strings.StringFunctions;  
StringFunctions.joinToString(list, ", ", "", "");
```

A detailed discussion of the annotation syntax comes later, in [Chapter 12](#).

## Top-level properties

Just like functions, properties can be placed at the top level of a file. Storing individual pieces of data outside of a class isn't needed as often but is still useful.

For example, you can use a `var` property to count the number of times some operation has been performed:

```
var opCount = 0 #1  
  
fun performOperation() {  
    opCount++ #2  
    // ...  
}
```

```
fun reportOperationCount() {  
    println("Operation performed $opCount times") #3  
}
```

The value of such a property will be stored in a static field.

Top-level properties also allow you to define constants in your code:

```
val UNIX_LINE_SEPARATOR = "\n"
```

By default, top-level properties, just like any other properties, are exposed to Java code as accessor methods (a getter for a `val` property and a getter/setter pair for a `var` property). If you want to expose a constant to Java code as a `public static final` field, to make its use more natural, you can mark it with the `const` modifier (this is allowed for properties of primitive types, as well as `String`):

```
const val UNIX_LINE_SEPARATOR = "\n"
```

This gets you the equivalent of the following Java code:

```
/* Java */  
public static final String UNIX_LINE_SEPARATOR = "\n";
```

You've improved the initial `joinToString` utility function quite a lot. Now let's look at how to make it even handier.



#### Note

The Kotlin standard library also contains a number of useful top-level functions and properties. An example for this is the `kotlin.math` package. It provides useful functions for typical mathematical and trigonometric operations, such as the `max` function to compute the maximum of two numbers. It also comes with a number of mathematical constants, like Euler's number, or Pi:

```
fun main() {  
    println(max(PI, E))  
    // 3.141592653589793
```

```
}
```

### 3.3 Adding methods to other people's classes: extension functions and properties

One of the main themes of Kotlin is smooth integration with existing code. Even pure Kotlin projects are built on top of Java libraries such as the JDK, the Android framework, and other third-party frameworks. And when you integrate Kotlin into a Java project, you're also dealing with the existing code that hasn't been or won't be converted to Kotlin. Wouldn't it be nice to be able to use all the niceties of Kotlin when working with those APIs, without having to rewrite them? That's what extension functions allow you to do.

Conceptually, an *extension function* is a simple thing: it's a function that can be called as a member of a class but is defined outside of it. To demonstrate that, let's add a method for computing the last character of a string:

```
package strings

fun String.lastChar(): Char = this.get(this.length - 1)
```

All you need to do is put the name of the class or interface that you're extending before the name of the function you're adding. This class name is called the *receiver type*; the value on which you're calling the extension function is called the *receiver object*. This is illustrated in [3.1](#).

**Figure 3.1.** In an extension function declaration, the *receiver type* is the type on which the extension is defined. You use it to specify the type your function extends. The *receiver object* is the instance of that type. You use it to access properties and methods of the type you're extending.

Receiver type

The diagram shows two arrows pointing from the text "Receiver type" and "Receiver object" to the "String" and "this" in the code respectively. The "String" arrow points to the first "String" in "String.lastChar()", and the "this" arrow points to the "this" in "this.get(this.length - 1)".

```
fun String.lastChar(): Char = this.get(this.length - 1)
```

Receiver object

You can call the function using the same syntax you use for ordinary class members:

```
fun main() {
    println("Kotlin".lastChar())
    // n
}
```

In this example, `String` is the receiver type, and "Kotlin" is the receiver object.

In a sense, you've added your own method to the `String` class. Even though `String` isn't part of your code, and you may not even have the source code to that class, you can still extend it with the methods you need in your project. It doesn't even matter whether `String` is written in Java, Kotlin, or some other JVM language, such as Groovy, or even whether it is marked as `final`, preventing subclassing. As long as it's compiled to a Java class, you can add your own extensions to that class.

In the body of an extension function, you use `this` the same way you would use it in a method. And, as in a regular method, you can omit it:

```
package strings

fun String.lastChar(): Char = get(length - 1) #1
```

In the extension function, you can directly access the methods and properties of the class you're extending, as in methods defined in the class itself. Note that extension functions don't allow you to break encapsulation. Unlike methods defined in the class, extension functions don't have access to private or protected members of the class.

Later we'll use the term *method* for both members of the class and extensions functions. For instance, we can say that in the body of the extension function you can call any method on the receiver, meaning you can call both members and extension functions. On the call site, extension functions are indistinguishable from members, and often it doesn't matter whether the particular method is a member or an extension.

### 3.3.1 Imports and extension functions

When you define an extension function, it doesn't automatically become available across your entire project. Instead, it needs to be imported, just like any other class or function. This helps avoid accidental name conflicts. Kotlin allows you to import individual functions using the same syntax you use for classes:

```
import strings.lastChar  
  
val c = "Kotlin".lastChar()
```

Of course, `*` imports work as well:

```
import strings.*  
  
val c = "Kotlin".lastChar()
```

You can change the name of the class or function you're importing using the `as` keyword:

```
import strings.lastChar as last  
  
val c = "Kotlin".last()
```

Changing a name on import is useful when you have several functions with the same name in different packages and you want to use them in the same file. For regular classes or functions, you have another choice in this situation: You can use a fully qualified name to refer to the class or function (and whether you can import a class or function at all also depends on its visibility modifier, as you'll see in [4.1.3](#).) For extension functions, the syntax requires you to use the short name, so the `as` keyword in an import statement is the only way to resolve the conflict.

### 3.3.2 Calling extension functions from Java

Under the hood, an extension function is a static method that accepts the receiver object as its first argument. Calling it doesn't involve creating adapter objects or any other runtime overhead.

That makes using extension functions from Java pretty easy: you call the static method and pass the receiver object instance. Just as with other top-level functions, the name of the Java class containing the method is determined from the name of the file where the function is declared. Let's say it was declared in a `StringUtil.kt` file:

```
/* Java */
char c = StringUtilKt.lastChar("Java");
```

This extension function is declared as a top-level function, so it's compiled to a static method. You can import the `lastChar` method statically from Java, simplifying the use to just `lastChar("Java")`. This code is somewhat less readable than the Kotlin version, but it's idiomatic from the Java point of view.

### 3.3.3 Utility functions as extensions

Now you can write the final version of the `joinToString` function. This is almost exactly what you'll find in the Kotlin standard library.

**Listing 3.5. Declaring `joinToString()` as an extension**

```
fun <T> Collection<T>.joinToString( #1
    separator: String = ", ", #2
    prefix: String = "", #2
    postfix: String = "" #2
): String {
    val result = StringBuilder(prefix)

    for ((index, element) in this.withIndex()) { #3
        if (index > 0) result.append(separator)
        result.append(element)
    }

    result.append(postfix)
    return result.toString()
}

fun main() {
    val list = listOf(1, 2, 3)
    println(
        list.joinToString(
```

```

        separator = "; ",
        prefix = "(",
        postfix = ")"
    )
}
// (1; 2; 3)
}

```

You make it an extension to a collection of elements, and you provide default values for all the arguments. Now you can invoke `joinToString` like a member of a class:

```

fun main() {
    val list = listOf(1, 2, 3)
    println(list.joinToString(" "))
    // 1 2 3
}

```

Because extension functions are effectively syntactic sugar over static method calls, you can use a more specific type as a receiver type, not only a class. Let's say you want to have a `join` function that can be invoked only on collections of strings.

```

fun Collection<String>.join(
    separator: String = ", ",
    prefix: String = "",
    postfix: String = ""
) = joinToString(separator, prefix, postfix)

fun main() {
    println(listOf("one", "two", "eight").join(" "))
    // one two eight
}

```

Calling this function with a list of objects of another type won't work:

```

fun main() {
    listOf(1, 2, 8).join()
    // Error: None of the following candidates is applicable because
    // receiver type mismatch:
    // public fun Collection<String>.join(...): String
    // defined in root package
}

```

The static nature of extensions also means that extension functions can't be overridden in subclasses. Let's look at an example.

### 3.3.4 No overriding for extension functions

Method overriding in Kotlin works as usual for member functions, but you can't override an extension function. Let's say you have two classes, `View` and `Button`. `Button` is a subclass of `View`, and overrides the `click` function from the superclass. To implement this, you mark `View` and `click` with the `open` modifier to allow overriding, and use the `override` modifier to provide an implementation in the subclass (we'll take a closer look at this syntax in [4.1.1](#), and learn more about the syntax for instantiating subclasses in [4.2.1](#)).

**Listing 3.6. Overriding a member function**

```
open class View { #1
    open fun click() = println("View clicked")
}

class Button: View() { #2
    override fun click() = println("Button clicked")
}
```

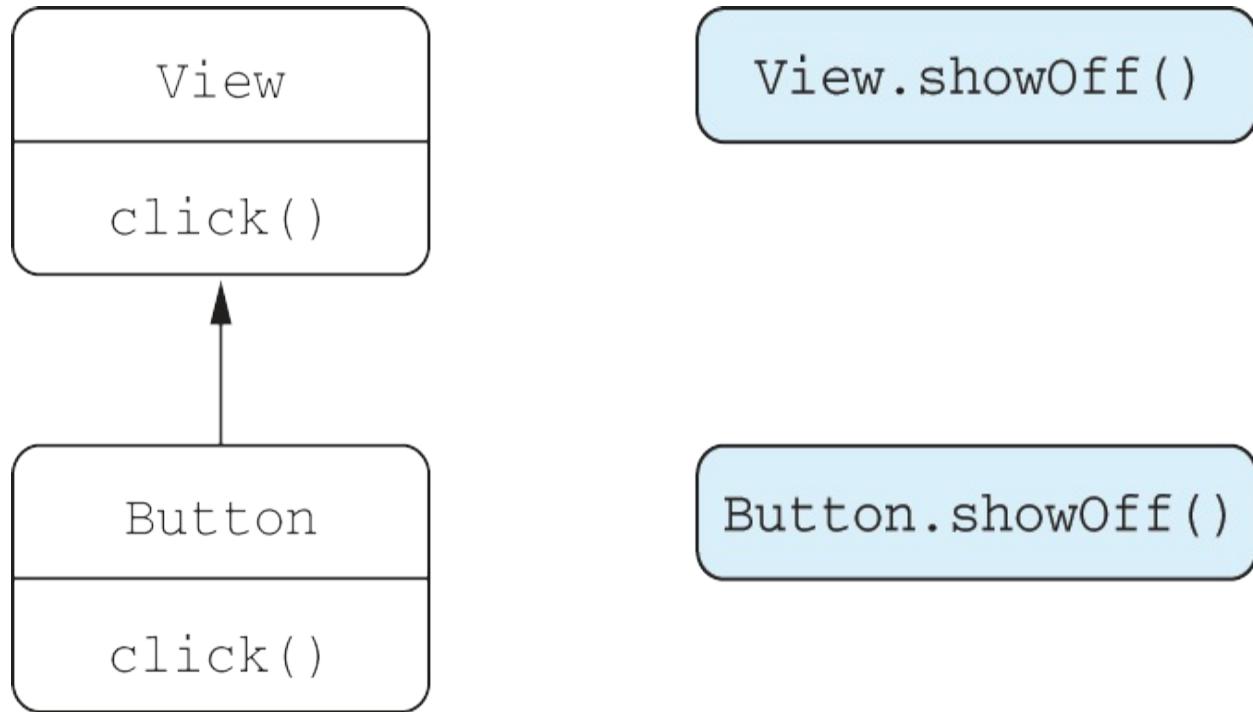
If you declare a variable of type `View`, you can store a value of type `Button` in that variable, because `Button` is a subtype of `View`. If you call a regular method, such as `click`, on this variable, and that method is overridden in the `Button` class, the overridden implementation from the `Button` class will be used:

```
fun main() {
    val view: View = Button()
    view.click() #1
    // Button clicked
}
```

But it doesn't work that way for extensions. Extension functions aren't a part of the class; they're declared externally to it, as shown in [3.2](#).

**Figure 3.2. The `view.showOff()` and `button.showOff()` extension functions are defined outside the**

## **View and Button classes.**



Even though you can define extension functions with the same name and parameter types for a base class and its subclass, the function that's called depends on the declared static type of the variable, determined at compile time, not on the runtime type of the value stored in that variable.

The following example shows two `showOff` extension functions declared on the `View` and `Button` classes. When you call `showOff` on a variable of type `View`, the corresponding extension is called, even though the actual type of the value is `Button`:

### **Listing 3.7. No overriding for extension functions**

```
fun View.showOff() = println("I'm a view!")
fun Button.showOff() = println("I'm a button!")

fun main() {
    val view: View = Button()
    view.showOff() #1
    // I'm a view!
}
```

It might help to recall that an extension function is compiled to a static function in Java with the receiver as the first argument. Java would choose the function the same way:

```
/* Java */
class Demo {
    public static void main(String[] args) {
        View view = new Button();
        ExtensionsKt.showOff(view); #1
        // I'm a view!
    }
}
```

As you can see, overriding doesn't apply to extension functions: Kotlin resolves them statically.



#### Note

If the class has a member function with the same signature as an extension function, the member function always takes precedence. You should keep this in mind when extending the API of classes: if you add a member function with the same signature as an extension function that a client of your class has defined, and they then recompile their code, it will change its meaning and start referring to the new member function. Your IDE will also warn you that the extension function is shadowed by a member function.

We've discussed how to provide additional methods for external classes. Now let's see how to do the same with properties.

### 3.3.5 Extension properties

You've already gotten to know the syntax for declaring Kotlin properties in [2.2.1](#), and just like extension *functions*, you can also specify *extension properties*. These allow you to extend classes with APIs that can be accessed using the property syntax, rather than the function syntax. Even though they're called *properties*, they can't have any state, because there's no proper place to store it: it's not possible to add extra fields to existing instances of Java objects. As a result, extension properties always have to define custom

accessors like the ones you learned about in [2.2.2](#). Still, they provide a shorter, more concise calling convention, which can still come in handy sometimes.

In the previous section, you defined a function `lastChar()`. Now let's convert it into a property—allowing you to call `"myText".lastChar` instead of `"myText.lastChar()"`.

#### **Listing 3.8. Declaring an extension property**

```
val String.lastChar: Char
    get() = this.get(length - 1)
```

You can see that, just as with functions, an extension property looks like a regular property with a receiver type added. The getter must always be defined, because there's no backing field and therefore no default getter implementation. Initializers aren't allowed for the same reason: there's nowhere to store the value specified as the initializer.

If you define the same property on a `StringBuilder`, you can make it a `var`, because the contents of a `StringBuilder` can be modified.

#### **Listing 3.9. Declaring a mutable extension property**

```
var StringBuilder.lastChar: Char
    get() = this.get(length - 1) #1
    set(value) { #2
        this.setCharAt(length - 1, value)
    }
```

You access extension properties exactly like member properties:

```
fun main() {
    val sb = StringBuilder("Kotlin?")
    println(sb.lastChar)
    // ?
    sb.lastChar = '!'
    println(sb)
    // Kotlin!
}
```

Note that when you need to access an extension property from Java, you have

to invoke its getter explicitly: `StringUtilKt.getLastChar("Java")`.

We've discussed the concept of extensions in general. Now let's return to the topic of collections and look at a few more library functions that help you handle them, as well as language features that come up in those functions.

## 3.4 Working with collections: varargs, infix calls, and library support

This section shows some of the functions from the Kotlin standard library for working with collections. Along the way, it describes a few related language features:

- The `vararg` keyword, which allows you to declare a function taking an arbitrary number of arguments
- An *infix* notation that lets you call some one-argument functions without ceremony
- *Destructuring declarations* that allow you to unpack a single composite value into multiple variables

### 3.4.1 Extending the Java Collections API

We started this chapter with the idea that collections in Kotlin are the same classes as in Java, but with an extended API. You saw examples of getting the last element in a list and finding the maximum in a collection of numbers:

```
fun main() {  
    val strings: List<String> = listOf("first", "second", "fourte  
    strings.last()  
    // fourteenth  
  
    val numbers: Collection<Int> = setOf(1, 14, 2)  
    numbers.sum()  
    // 17  
}
```

We were interested in how it works: why it's possible to do so many things with collections in Kotlin out of the box, even though they're instances of the

Java library classes. Now the answer should be clear: the `last` and `sum` functions are declared as extension functions, and are always imported by default in your Kotlin files!

The `last` function is no more complex than `lastChar` for `String`, discussed in the previous section: it's an extension on the `List` class. For `sum`, we show a simplified declaration (the real library function works not only for `Int` numbers, but for any number types):

```
fun <T> List<T>.last(): T { /* returns the last element */ }
fun Collection<Int>.sum(): Int { /* sum up all elements */ }
```

Many extension functions are declared in the Kotlin standard library, and we won't list all of them here. You may wonder about the best way to learn everything in the Kotlin standard library. You don't have to—any time you need to do something with a collection or any other object, the code completion in the IDE will show you all the possible functions available for that type of object. The list includes both regular methods and extension functions; you can choose the function you need. In addition to that, the standard library reference (<https://kotlinlang.org/api/latest/jvm/stdlib/>) lists all the methods available for each library class—members as well as extensions.

At the beginning of the chapter, you also saw functions for creating collections. A common trait of those functions is that they can be called with an arbitrary number of arguments. In the following section, you'll see the syntax for declaring such functions.

### 3.4.2 Varargs: functions that accept an arbitrary number of arguments

When you call a function to create a list, you can pass any number of arguments to it:

```
val list = listOf(2, 3, 5, 7, 11)
```

If you look up how this function is declared in the standard library, you'll find the following signature:

```
fun listOf<T>(vararg values: T): List<T> { /* implementation */ }
```

This method makes use of a language feature that allows you to pass an arbitrary number of values to a method by packing them in an array: varargs. Kotlin's varargs are similar to those in Java, but the syntax is slightly different: instead of three dots after the type, Kotlin uses the `vararg` modifier on the parameter.

One other difference between Kotlin and Java is the syntax of calling the function when the arguments you need to pass are already packed in an array. In Java, you pass the array as is, whereas Kotlin requires you to explicitly unpack the array, so that every array element becomes a separate argument to the function being called. This feature is called a *spread operator*, and using it is as simple as putting the `*` character before the corresponding argument. In this snippet, you're "spreading" the `args` array received by the `main` function to be used as variable arguments for the `listOf` function:

```
fun main(args: Array<String>) {  
    val list = listOf("args: ", *args) #1  
    println(list)  
}
```

This example shows that the spread operator lets you combine the values from an array and some fixed values in a single call. This isn't supported in Java.

Now let's move on to maps. We'll briefly discuss another way to improve the readability of Kotlin function invocations: the *infix call*.

### 3.4.3 Working with pairs: infix calls and destructuring declarations

To create maps, you use the `mapOf` function:

```
val map = mapOf(1 to "one", 7 to "seven", 53 to "fifty-three")
```

This is a good time to provide another explanation we promised you at the beginning of the chapter. The word `to` in this line of code isn't a built-in construct, but rather a method invocation of a special kind, called an *infix call*.

In an infix call, the method name is placed immediately between the target object name and the parameter, with no extra separators. The following two calls are equivalent:

```
1.to("one") #1  
1 to "one" #2
```

Infix calls can be used with regular methods and extension functions that have exactly one required parameter. To allow a function to be called using the infix notation, you need to mark it with the `infix` modifier. Here's a simplified version of the declaration of the `to` function:

```
infix fun Any.to(other: Any) = Pair(this, other)
```

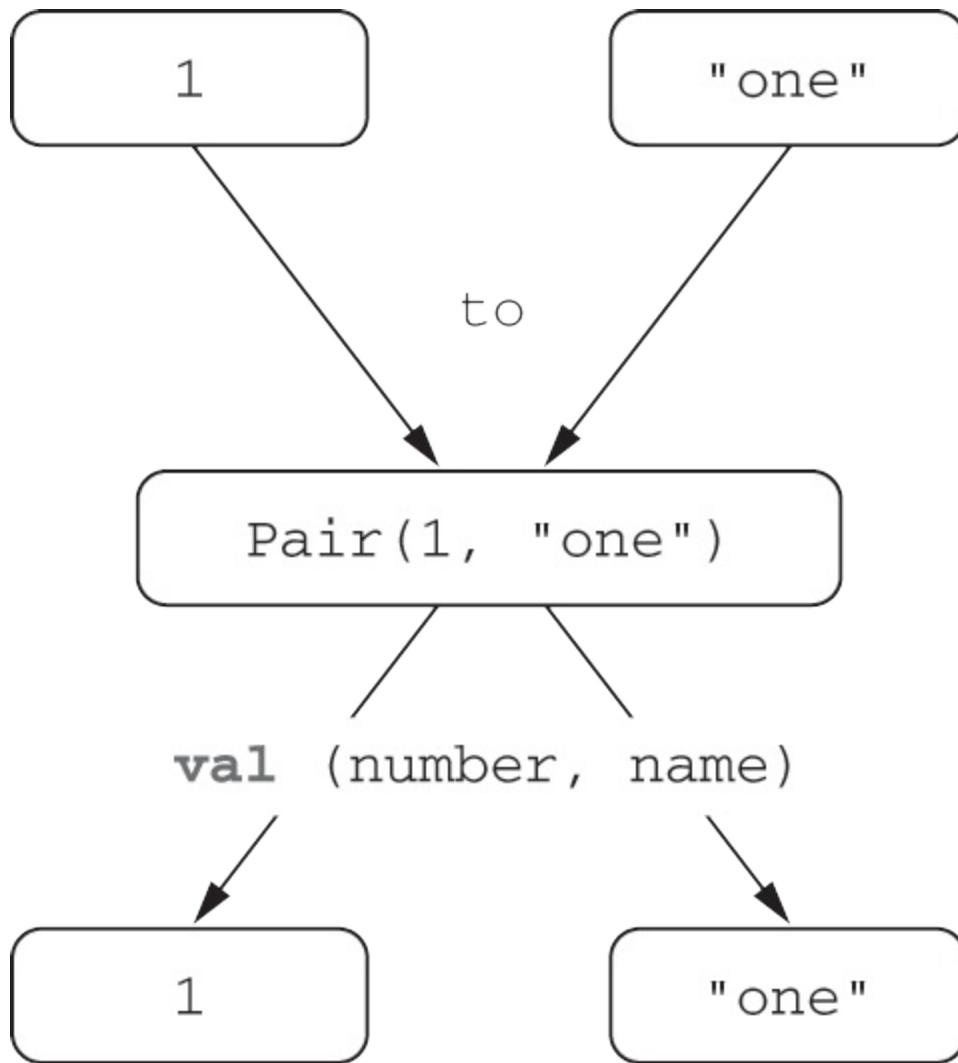
The `to` function returns an instance of `Pair`, which is a Kotlin standard library class that, unsurprisingly, represents a pair of elements. The actual declarations of `Pair` and `to` use generics, but we're omitting them here to keep things simple.

Note that you can initialize two variables with the contents of a `Pair` directly:

```
val (number, name) = 1 to "one"
```

This feature is called a *destructuring declaration*. [3.3](#) illustrates how it works with pairs.

**Figure 3.3. You create a pair using the `to` function and unpack it with a destructuring declaration.**



The destructuring declaration feature isn't limited to pairs. For example, you can also initialize two variables, `key` and `value`, with the contents of a map entry (something you've already seen very briefly in [2.25](#)).

This also works with loops, as you've seen in the implementation of `joinToString`, which uses the `withIndex` function:

```
for ((index, element) in collection.withIndex()) {
    println("$index: $element")
}
```

[9.4](#) will describe the general rules for destructuring an expression and using it to initialize several variables.

The `to` function is an extension function. You can create a pair of any elements, which means it's an extension to a generic receiver: you can write `1 to "one", "one" to 1, list to list.size()`, and so on. Let's look at the signature of the `mapOf` function:

```
fun <K, V> mapOf(vararg values: Pair<K, V>): Map<K, V>
```

Like `listOf`, `mapOf` accepts a variable number of arguments, but this time they should be pairs of keys and values. Even though the creation of a new map may look like a special construct in Kotlin, it's a regular function with a concise syntax.

Next, let's discuss how extensions simplify dealing with strings and regular expressions.

## 3.5 Working with strings and regular expressions

Kotlin strings are exactly the same things as Java strings. You can pass a string created in Kotlin code to any Java method, and you can use any Kotlin standard library methods on strings that you receive from Java code. No conversion is involved, and no additional wrapper objects are created.

Kotlin makes working with standard Java strings more enjoyable by providing a bunch of useful extension functions. Also, it hides some confusing methods, adding extensions that are clearer. As our first example of the API differences, let's look at how Kotlin handles splitting strings.

### 3.5.1 Splitting strings

You're probably familiar with the `split` method on `String`. Everyone uses it, but sometimes people complain about it on Stack Overflow (<http://stackoverflow.com>): "The `split` method in Java doesn't work with a dot." It's a common trap to write `"12.345-6.A".split(".")` and to expect an array `[12, 345-6, A]` as a result. But Java's `split` method returns an empty array! That happens because it takes a regular expression as a parameter, and it splits a string into several strings according to the expression. Here, the dot `(.)` is a regular expression that denotes any character.

Kotlin hides the confusing method and provides as replacements several overloaded extensions named `split` that have different arguments. The one that takes a regular expression requires a value of type `Regex` or `Pattern`, not `String`. This ensures that it's always clear whether a string passed to a method is interpreted as plain text or a regular expression.

Here's how you'd split the string with either a dot or a dash:

```
fun main() {  
    println("12.345-6.A".split("\\.|-".toRegex())) #1  
    // [12, 345, 6, A]  
}
```

Kotlin uses exactly the same regular expression syntax as in Java. The pattern here matches a dot (we escaped it to indicate that we mean a literal character, not a wildcard) or a dash. The APIs for working with regular expressions are also similar to the standard Java library APIs, but they're more idiomatic. For instance, in Kotlin you use an extension function `toRegex` to convert a string into a regular expression.

But for such a simple case, you don't need to use regular expressions. The other overload of the `split` extension function in Kotlin takes an arbitrary number of delimiters as plain-text strings:

```
fun main() {  
    println("12.345-6.A".split(".", "-")) #1  
    // [12, 345, 6, A]  
}
```

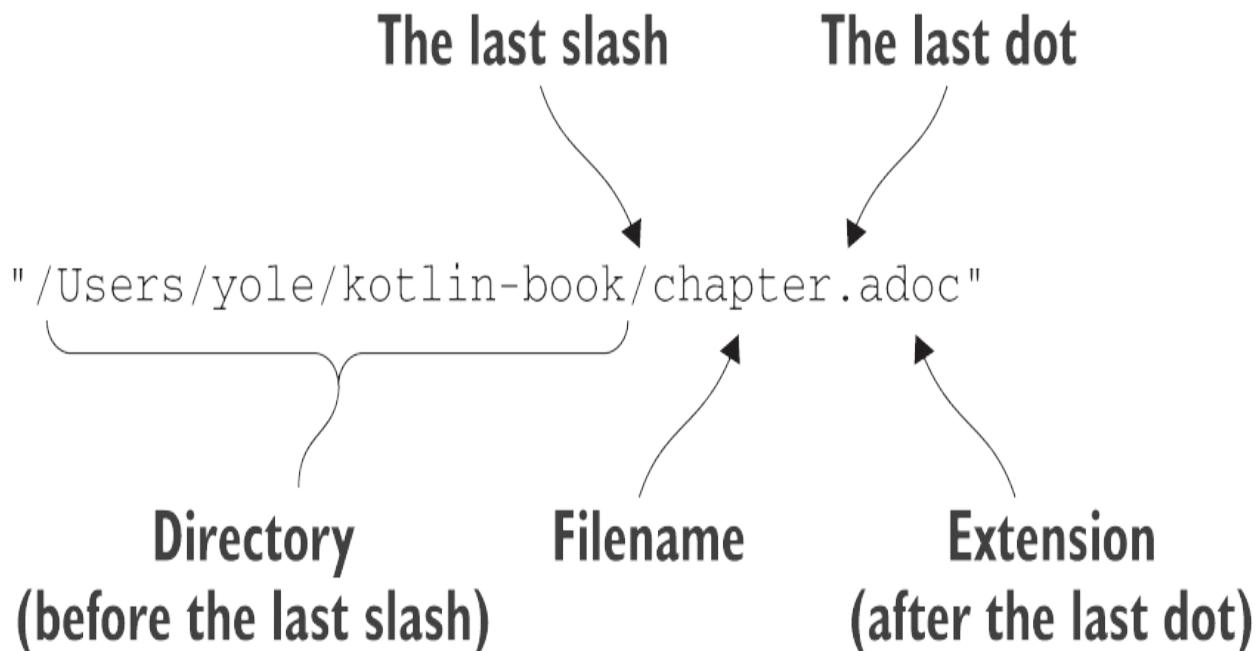
Note that you can specify character arguments instead and write `"12.345-6.A".split('.','-')`, which will lead to the same result. This method replaces the similar Java method that can take only one character as a delimiter.

### 3.5.2 Regular expressions and triple-quoted strings

Let's look at another example with two different implementations: the first one will use extensions on `String`, and the second will work with regular expressions. Your task will be to parse a file's full path name into its

components: a directory, a filename, and an extension. The Kotlin standard library contains functions to get the substring before (or after) the first (or the last) occurrence of the given delimiter. Here's how you can use them to solve this task (also see [3.4](#)).

**Figure 3.4.** Splitting a path into a directory, a filename, and a file extension by using the `substringBeforeLast` and `substringAfterLast` functions



**Listing 3.10.** Using string extensions for parsing paths

```
fun parsePath(path: String) {
    val directory = path.substringBeforeLast("/")
    val fullName = path.substringAfterLast("/")

    val fileName = fullName.substringBeforeLast(".")
    val extension = fullName.substringAfterLast(".")

    println("Dir: $directory, name: $fileName, ext: $extension")
}

fun main() {
    parsePath("/Users/yole/kotlin-book/chapter.adoc")
    // Dir: /Users/yole/kotlin-book, name: chapter, ext: adoc
}
```

The substring before the last slash symbol of the file path is the path to an enclosing directory, the substring after the last dot is a file extension, and the filename goes between them.

Kotlin makes it easier to work with strings without resorting to regular expressions, which are powerful but also sometimes hard to understand after they've been written. If you do want to use regular expressions, the Kotlin standard library can help. Here's how the same task can be done using regular expressions:

**Listing 3.11. Using regular expressions for parsing paths**

```
fun parsePathRegex(path: String) {
    val regex = """(.+)/(.+)\.(.+)""".toRegex()
    val matchResult = regex.matchEntire(path)
    if (matchResult != null) {
        val (directory, filename, extension) = matchResult.destructured
        println("Dir: $directory, name: $filename, ext: $extension")
    }
}

fun main() {
    parsePathRegex("/Users/yole/kotlin-book/chapter.adoc")
    // Dir: /Users/yole/kotlin-book, name: chapter, ext: adoc
}
```

In this example, the regular expression is written in a *triple-quoted string*. In such a string, you don't need to escape any characters, including the backslash, so you can encode a regular expression matching a literal dot with \. rather than \\. as you'd write in an ordinary string literal (see [3.5](#)).

**Figure 3.5. The regular expression for splitting a path into a directory, a filename, and a file extension**

## The last slash      The last dot

A diagram illustrating a regular expression pattern for path parsing. The pattern is triple-quoted: " " "( .+) / ( .+) \ . ( .+) " " ". Two arrows point from the labels "The last slash" and "The last dot" above to the backslash character '\' and the dot character '.' respectively in the pattern. Below the pattern, three curly braces group the first two capture groups as "Directory", the third as "Filename", and the fourth as "Extension".

## Directory      Filename      Extension

This regular expression divides a path into three groups separated by a slash and a dot. The pattern `.` matches any character from the beginning, so the first group `(.+)` contains the substring before the last slash. This substring includes all the previous slashes, because they match the pattern "any character". Similarly, the second group contains the substring before the last dot, and the third group contains the remaining part.

Now let's discuss the implementation of the `parsePathRegex` function from the previous example. You create a regular expression and match it against an input path. If the match is successful (the result isn't `null`), you assign the value of its `destructured` property to the corresponding variables. This is the same syntax used when you initialized two variables with a `Pair`; [9.4](#) will cover the details.

### 3.5.3 Multiline triple-quoted strings

The purpose of triple-quoted strings is not only to avoid escaping characters. Such a string literal can contain any characters, including line breaks. That gives you an easy way to embed in your programs text containing line breaks. As an example, let's draw some ASCII art:

```
val kotlinLogo =  
    """  
    | //  
    | //
```

```
    |/ \
""".trimIndent()

fun main() {
    println(kotlinLogo)
    // | //
    // | //
    // | / \
}
```

The multiline string contains all the characters between the triple quotes. That includes the line breaks<sup>[2]</sup> and indents used to format the code. In cases like this, you are most likely only interested in the actual content of your string. By calling `trimIndent`, you can remove that common minimal indent of all the lines of your string, and remove the first and last lines of the string, given that they are blank.

As you saw in the previous example, a triple-quoted string can contain line breaks. However, you can't use special characters like `\n`. On the other hand, you don't have to escape `\`, so the Windows-style path  
"C:\\Users\\yole\\kotlin-book" can be written as  
"""C:\\Users\\yole\\kotlin-book""".

You can also use string templates in multiline strings. Because multiline strings don't support escape sequences, you have to use an embedded expression if you need to use a literal dollar sign or an escaped Unicode symbol in the contents of your string. So, rather than write `val think = """Hmm \uD83E\uDD14"""`, you'll have to write the following to properly interpret the escaped symbol encoded in this string: `val think = """Hmm ${"\uD83E\uDD14"}"""`

One of the areas where multiline strings can be useful in your programs (besides games that use ASCII art) is tests. In tests, it's fairly common to execute an operation that produces multiline text (for example, a web page fragment, or other structured text) and to compare the result with the expected output. Multiline strings give you a perfect solution for including the expected output as part of your tests. No need for clumsy escaping or loading the text from external files—just put in some quotation marks and place the expected HTML, XML, JSON, or other output between them. And for better formatting, use the aforementioned `trimIndent` function, which is

another example of an extension function:

```
val expectedPage = """
    <html lang="en">
        <head>
            <title>A page</title>
        </head>
        <body>
            <p>Hello, Kotlin!</p>
        </body>
    </html>
""".trimIndent()

val expectedObject = """
{
    "name": "Sebastian",
    "age": 27,
    "homeTown": "Munich"
}
""".trimIndent()
```



#### Syntax highlighting inside triple-quoted strings in IntelliJ IDEA and Android Studio

Using triple-quoted strings for formatted text like HTML or JSON gives you an additional benefit: IntelliJ IDEA and Android Studio can provide you with syntax highlighting inside those string literals. To enable highlighting, place your cursor inside the string, and press Alt + Enter (or Option + Return on macOS) or click the floating yellow lightbulb icon, and select "Inject language or reference". Next, select the type of language you're using in the string, e.g. JSON.

Your multiline string then becomes properly syntax-highlighted JSON. In case your text snippet happens to be malformed JSON, you'll even get warnings and descriptive error messages, all within your Kotlin string.

By default, this highlighting is temporary. To instruct your IDE to always inject a string literal with a given language, you can use the `@Language("JSON")` annotation:

For more information on language injections in IntelliJ IDEA and Android Studio, take a look at <https://www.jetbrains.com/help/idea/using-language-injections.html>

[injections.html](#).

You can now see that extension functions are a powerful way to extend the APIs of existing libraries and to adapt them to the idioms of the Kotlin language. And indeed, a large portion of the Kotlin standard library is made up of extension functions for standard Java classes. You can also find many community-developed libraries that provide Kotlin-friendly extensions for third-party libraries.

Now that you can see how Kotlin gives you better APIs for the libraries you use, let's turn our attention back to your code. You'll see some new uses for extension functions, and we'll also discuss a new concept: *local functions*.

[2] Different operating systems use different characters to mark the end of a line in a file: Windows uses CRLF (Carriage Return Line Feed), Linux and macOS uses LF (Line Feed). Regardless of the used operating system, Kotlin interprets CRLF, LF, and CR as line breaks.

## 3.6 Making your code tidy: local functions and extensions

Many developers believe that one of the most important qualities of good code is the lack of duplication. There's even a special name for this principle: Don't Repeat Yourself (DRY). But when you write in Java, following this principle isn't always trivial. In many cases, it's possible to use the Extract Method refactoring feature of your IDE to break longer methods into smaller chunks, and then to reuse those chunks. But this can make code more difficult to understand, because you end up with a class with many small methods and no clear relationship between them. You can go even further and group the extracted methods into an inner class, which lets you maintain the structure, but this approach requires a significant amount of boilerplate.

Kotlin gives you a cleaner solution: you can nest the functions you've extracted in the containing function. This way, you have the structure you need without any extra syntactic overhead.

Let's see how to use local functions to fix a fairly common case of code

duplication. In [3.12](#), a function saves a user to a database, and you need to make sure the user object contains valid data.

**Listing 3.12. A function with repetitive code**

```
class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {
    if (user.name.isEmpty()) { #1
        throw IllegalArgumentException(
            "Can't save user ${user.id}: empty Name")
    }

    if (user.address.isEmpty()) { #1
        throw IllegalArgumentException(
            "Can't save user ${user.id}: empty Address")
    }

    // Save user to the database
}

fun main() {
    saveUser(User(1, "", ""))
    // java.lang.IllegalArgumentException: Can't save user 1: emp
}
```

The amount of duplicated code here is fairly small, and you probably won't want to have a full-blown method in your class that handles one special case of validating a user. But if you put the validation code into a local function, you can get rid of the duplication and still maintain a clear code structure. Here's how it works.

**Listing 3.13. Extracting a local function to avoid repetition**

```
class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {

    fun validate(user: User, #1
                value: String,
                fieldName: String) {
        if (value.isEmpty()) {
            throw IllegalArgumentException(
                "Can't save user ${user.id}: empty $fieldName")
    }
}
```

```

        }
    }

    validate(user, user.name, "Name") #2
    validate(user, user.address, "Address")

    // Save user to the database
}

```

This looks better. The validation logic isn't duplicated, but it's still confined to the scope of the `validate` function. As the project evolves, you can easily add more validations if you need to add other fields to `User`. But having to pass the `User` object to the validation function is somewhat ugly. The good news is that it's entirely unnecessary, because local functions have access to all parameters and variables of the enclosing function. Let's take advantage of that and get rid of the extra `User` parameter.

**Listing 3.14. Accessing outer function parameters in a local function**

```

class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {
    fun validate(value: String, fieldName: String) { #1
        if (value.isEmpty()) {
            throw IllegalArgumentException(
                "Can't save user ${user.id}: " + #2
                "empty $fieldName")
        }
    }

    validate(user.name, "Name")
    validate(user.address, "Address")

    // Save user to the database
}

```

To improve this example even further, you can move the validation logic into an extension function of the `User` class.

**Listing 3.15. Extracting the logic into an extension function**

```

class User(val id: Int, val name: String, val address: String)

```

```

fun User.validateBeforeSave() {
    fun validate(value: String, fieldName: String) {
        if (value.isEmpty()) {
            throw IllegalArgumentException(
                "Can't save user $id: empty $fieldName") #1
        }
    }
    validate(name, "Name")
    validate(address, "Address")
}

fun saveUser(user: User) {
    user.validateBeforeSave() #2
    // Save user to the database
}

```

Extracting a piece of code into an extension function turns out to be surprisingly useful. Even though `User` is a part of your codebase and not a library class, you don't want to put this logic into a method of `User`, because it's not relevant to any other places where `User` is used. If you follow this approach the API of the class contains only the essential methods used everywhere, so the class remains small and easy to wrap your head around. On the other hand, functions that primarily deal with a single object and don't need access to its private data can access its members without extra qualification, as in [3.15](#).

Extension functions can also be declared as local functions, so you could go even further and put `User.validateBeforeSave` as a local function in `saveUser`. But deeply nested local functions are usually fairly hard to read; so, as a general rule, we don't recommend using more than one level of nesting.

Having looked at all the cool things you can do with functions, in the next chapter we'll look at what you can do with classes.

## 3.7 Summary

- Kotlin enhances the Java collection classes with a richer API.
- Defining default values for function parameters greatly reduces the need

to define overloaded functions, and the named-argument syntax makes calls to functions with many parameters much more readable.

- Functions and properties can be declared directly in a file, not just as members of a class, allowing for a more flexible code structure.
- Extension functions and properties let you extend the API of any class, including classes defined in external libraries, without modifying its source code and with no runtime overhead.
- Infix calls provide a clean syntax for calling operator-like methods with a single argument.
- Kotlin provides a large number of convenient string-handling functions for both regular expressions and plain strings.
- Triple-quoted strings provide a clean way to write expressions that would require a lot of noisy escaping and string concatenation in Java.
- Local functions help you structure your code more cleanly and eliminate duplication.

# 4 Classes, objects, and interfaces

## This chapter covers

- Classes and interfaces
- Nontrivial properties and constructors
- Data classes
- Class delegation
- Using the `object` keyword

In this chapter, you'll get a deeper understanding of working with classes and interfaces in Kotlin. You already saw the basic syntax for declaring a class in [2.2](#). You know how to declare methods and properties, use simple primary constructors (aren't they nice?), and work with enums. But there's a lot more to see and learn on the topic!

Kotlin's classes and interfaces differ a bit from what you might be used to from Java: for example, interfaces can contain property declarations. Kotlin's declarations are `final` and `public` by default. In addition, nested classes aren't inner by default: they don't contain an implicit reference to their outer class.

For constructors, the short primary constructor syntax works great for the majority of cases, but Kotlin also comes with full syntax that lets you declare constructors with nontrivial initialization logic. The same works for properties: the concise syntax is nice, but you can easily define your own implementations of accessors.

The Kotlin compiler can generate useful methods to avoid verbosity. Declaring a class as a data class instructs the compiler to generate several standard methods for this class. You can also avoid writing delegating methods by hand, because the delegation pattern is supported natively in Kotlin.

You'll also get to see Kotlin's `object` keyword, that declares a class and also

creates an instance of the class. The keyword is used to express singleton objects, companion objects, and object expressions (analogous to Java anonymous classes). Let's start by talking about classes and interfaces and the details of defining class hierarchies in Kotlin.

## 4.1 Defining class hierarchies

In this section, you'll take a look at how class hierarchies are defined in Kotlin. We'll look at Kotlin's visibility and access modifiers, and which defaults Kotlin chooses for them. You'll also learn about the sealed modifier, which restricts the possible subclasses of a class, or implementations of an interface.

### 4.1.1 Interfaces in Kotlin

We'll begin with a look at defining and implementing interfaces. Kotlin interfaces can contain definitions of abstract methods as well as implementations of non-abstract methods. However, they can't contain any state.

To declare an interface in Kotlin, use the `interface` keyword instead of `class`. An interface that indicates that an element is clickable—like a button, or a hyperlink—could look like this:

**Listing 4.1. Declaring a simple interface**

```
interface Clickable {  
    fun click()  
}
```

This declares an interface with a single abstract method named `click` that doesn't return any value<sup>[3]</sup>. All non-abstract classes implementing the interface need to provide an implementation of this method.

To mark a button as `Clickable`, you put the interface name behind a colon after the class name, and provide an implementation for the `click` function:

**Listing 4.2. Implementing a simple interface**

```
class Button : Clickable {
    override fun click() = println("I was clicked")
}

fun main() {
    Button().click()
    // I was clicked
}
```

Kotlin uses the colon after the class name for both *composition* (that is, implementing interfaces) and *inheritance* (that is, subclassing, as you'll see it in [4.1.2](#)). A class can implement as many interfaces as it wants, but it can extend only one class.

The `override` modifier is used to mark methods and properties that override those from the superclass or interface. Unlike Java, which uses the `@Override` annotation, *using the `override` modifier is mandatory* in Kotlin. This saves you from accidentally overriding a method if it's added after you wrote your implementation; your code won't compile unless you explicitly mark the method as `override` or rename it.

An interface method can have a default implementation. To do so, you just provide a method body. In this case, you could add a function `showOff` with a default implementation to the `Clickable` interface that simply prints some text:

**Listing 4.3. Defining a method with a body in an interface**

```
interface Clickable {
    fun click() #1
    fun showOff() = println("I'm clickable!") #2
}
```

If you implement this interface, you are forced to provide an implementation for `click`. You can redefine the behavior of the `showOff` method, or you can omit it if you're fine with the default behavior.

Let's suppose now that another interface also defines a `showOff` method and has the following implementation for it.

**Listing 4.4. Defining another interface implementing the same method**

```
interface Focusable {  
    fun setFocus(b: Boolean) =  
        println("I ${if (b) "got" else "lost"} focus.")  
  
    fun showOff() = println("I'm focusable!")  
}
```

What happens if you need to implement both interfaces in your class? Each of them contains a `showOff` method with a default implementation; which implementation wins? Neither one wins. Instead, you get the following compiler error if you don't implement `showOff` explicitly:

```
The class 'Button' must override public open fun showOff()  
because it inherits many implementations of it.
```

The Kotlin compiler forces you to provide your own implementation.

#### **Listing 4.5. Invoking an inherited interface method implementation**

```
class Button : Clickable, Focusable {  
    override fun click() = println("I was clicked")  
  
    override fun showOff() { #1  
        super<Clickable>.showOff() #2  
        super<Focusable>.showOff() #2  
    }  
}
```

The `Button` class now implements two interfaces. You implement `showOff()` by calling both implementations that you inherited from supertypes. To invoke an inherited implementation, you use the `super` keyword together with the base type name in angle brackets: `super<Clickable>.showOff()` (a different syntax from Java's `Clickable.super.showOff()`)

If you only need to invoke one inherited implementation, you can use the expression body syntax, and write this:

```
override fun showOff() = super<Clickable>.showOff()
```

To verify that everything you've read so far is actually true, you can create an instance of your `Button` class and invoke all the inherited methods—the overridden `showOff` and `click` functions, as well as the `setFocus` function

from the `Focusable` interface, which provided a default implementation:

**Listing 4.6. Calling inherited and overridden methods**

```
fun main() {
    val button = Button()
    button.showOff()
    // I'm clickable!
    // I'm focusable!
    button.setFocus(true)
    // I got focus.
    button.click()
    // I was clicked.
}
```

**Implementing interfaces with method bodies in Java**

Kotlin compiles each interface with default methods to a combination of a regular interface and a class containing the method bodies as static methods. The interface contains only declarations, and the class contains all the implementations as static methods. Therefore, if you need to implement such an interface in a Java class, you have to define your own implementations of all methods, including those that have method bodies in Kotlin. For example, a `JavaButton` that implements `Clickable` needs to provide implementations for both `click` and `showOff`, even though Kotlin provides a default implementation for the latter:

**Listing 4.7. Calling inherited and overridden methods**

```
class JavaButton implements Clickable {
    @Override
    public void click() {
        System.out.println("I was clicked");
    }

    @Override
    public void showOff() {
        System.out.println("I'm showing off"); #1
    }
}
```

Now that you've seen how Kotlin allows you to implement methods defined

in interfaces, let's look at the second half of that story: overriding members defined in base classes.

### 4.1.2 Open, final, and abstract modifiers: final by default

By default, you can't create a subclass for a Kotlin class or override any methods from a base class—all classes and methods are *final* by default.

This sets it apart from Java, where you are allowed to create subclasses of any class and can override any method, unless it has been explicitly marked with the `final` keyword. Why didn't Kotlin follow this approach? Because while this is often convenient, it's also problematic.

The so-called *fragile base class* problem occurs when modifications of a base class can cause incorrect behavior of subclasses because the changed code of the base class no longer matches the assumptions in its subclasses.

If the class doesn't provide exact rules for how it should be subclassed (which methods are supposed to be overridden and how), the clients are at risk of overriding the methods in a way the author of the base class didn't expect. Because it's impossible to analyze all the subclasses, the base class is "fragile" in the sense that any change in it may lead to unexpected changes of behavior in subclasses.

To protect against this problem, *Effective Java* by Joshua Bloch (Addison-Wesley, 2008), one of the best-known books on good Java programming style, recommends that you "design and document for inheritance or else prohibit it." This means all classes and methods that aren't specifically intended to be overridden in subclasses have to be explicitly marked as `final`.

Kotlin follows this philosophy, making its classes, methods, and properties are `final` by default.

If you want to allow the creation of subclasses of a class, you need to mark the class with the `open` modifier. In addition, you need to add the `open` modifier to every property or method that can be overridden.

Let's say you want to spice up your user interface beyond a simple button, and create a clickable `RichButton`. Subclasses of this class should be able to provide their own animations, but shouldn't be able to break basic behavior such as disabling the button. You could declare the class as follows:

**Listing 4.8. Declaring an open class with an open method**

```
open class RichButton : Clickable { #1
    fun disable() {} #2
    open fun animate() {} #3
    override fun click() {} #4
}
```

This means a subclass of `RichButton` could in turn look like this:

**Listing 4.9. Declaring an open class with an open method**

```
class ThemedButton : RichButton() { #1
    override fun animate() {} #2
    override fun click() {} #3
    override fun showOff() {} #4
}
```

Note that if you override a member of a base class or interface, the overriding member will also be `open` by default. If you want to change this and forbid the subclasses of your class from overriding your implementation, you can explicitly mark the overriding member as `final`.

**Listing 4.10. Forbidding an override**

```
open class RichButton : Clickable {
    final override fun click() {} #1
}
```

**Open classes and smart casts**

One significant benefit of classes that are `final` by default is that they enable smart casts in a larger variety of scenarios. As we mentioned in [2.3.6](#), the compiler can only perform a smart cast (an automatic cast that allows you to access members without further manual casting) for variables which couldn't

have changed after the type check.

For a class, this means smart casts can only be used with a class property that is a `val` and that doesn't have a custom accessor. This requirement means the property has to be `final`, because otherwise a subclass could override the property and define a custom accessor, breaking the key requirement of smart casts.

Because properties are `final` by default, you can use smart casts with most properties without thinking about it explicitly, which improves the expressiveness of your code.

You can also declare a class as `abstract`, making it so the class can't be instantiated. An abstract class usually contains abstract members that don't have implementations and must be overridden in subclasses. Abstract members are always open, so you don't need to use an explicit `open` modifier (just like you don't need an explicit `open` modifier in an `interface`).

An example for an abstract class would be class which defines the properties of an animation, like the animation speed and number of frames, as well as behavior for running the animation. Since these properties and methods only make sense when implemented by another object, `Animated` is marked as `abstract`:

#### **Listing 4.11. Declaring an abstract class**

```
abstract class Animated { #1
    abstract val animationSpeed: Double #2
    val keyframes: Int = 20 #3
    open val frames: Int = 60 #3

    abstract fun animate() #4
    open fun stopAnimating() { } #5
    fun animateTwice() { } #5
}
```

[4.1](#) lists the access modifiers in Kotlin. The comments in the table are applicable to modifiers in classes; in interfaces, you don't use `final`, `open`, or `abstract`. A member in an interface is always `open`; you can't declare it as `final`. It's `abstract` if it has no body, but the keyword isn't required.

**Table 4.1. The meaning of access modifiers in a class**

Modifier	Corresponding member	Comments
final	Can't be overridden	Used by default for class members
open	Can be overridden	Should be specified explicitly
abstract	Must be overridden	Can be used only in abstract classes; abstract members can't have an implementation
override	Overrides a member in a superclass or interface	Overridden member is open by default, if not marked final

Having discussed the modifiers that control inheritance, let's now take a look at the role visibility modifiers play in Kotlin.

### 4.1.3 Visibility modifiers: public by default

Visibility modifiers help to control access to declarations in your code base. By restricting the visibility of a class's implementation details, you ensure that you can change them without the risk of breaking code that depends on the class.

Kotlin provides `public`, `protected`, and `private` modifiers which are analogous to their Java counterpart: `public` declarations are visible everywhere, `protected` declarations are visible in subclasses, and `private` declarations are visible inside a class, or, in the case of top-level declarations,

visible inside a file.

In Kotlin, not specifying a modifier means the declaration is `public`.



**Note**

Public-by-default differs from what you might know from Java, but it is a convenient convention for application developers. Library authors, on the other hand, usually want to make sure that no parts of their API are accidentally exposed. Hiding a previously public API after the fact would be a breaking change, after all. For this case, Kotlin provides an *explicit API mode*. In explicit mode, you need to specify the visibility for declarations that would be exposed as a public API, and explicitly specify the type for properties and functions that are part of the public API, as well. Explicit API mode can be enabled using the `-Xexplicit-api={strict|warning}` compiler option or via your build system.

For restricting visibility inside a module, Kotlin provides the visibility modifier `internal`. A *module* is a set of Kotlin files compiled together. That could be a Gradle source set, a Maven project, or an IntelliJ IDEA module.



**Note**

When using Gradle, the test source set can access declarations marked as `internal` from your main source set.

### No package-private in Kotlin

Kotlin doesn't have the concept of package-private visibility, which is the default visibility in Java. Kotlin uses packages only as a way of organizing code in namespaces; it doesn't use them for visibility control.

The advantage of `internal` visibility is that it provides real encapsulation for the implementation details of your module. In Java, the encapsulation can be easily broken, because external code can define classes in the same packages used by your code and thus get access to your package-private declarations.

Kotlin also allows you to use `private` visibility for top-level declarations, including classes, functions, and properties. Such declarations are visible only in the file where they are declared. This is another useful way to hide the implementation details of a subsystem. [4.2](#) summarizes all the visibility modifiers.

**Table 4.2. Kotlin visibility modifiers**

Modifier	Class member	Top-level declaration
<code>public</code> (default)	Visible everywhere	Visible everywhere
<code>internal</code>	Visible in a module	Visible in a module
<code>protected</code>	Visible in subclasses	—
<code>private</code>	Visible in a class	Visible in a file

Let's look at an example. Every line in the `giveSpeech` function tries to violate the visibility rules. It compiles with an error.

**Listing 4.12. Using different visibility modifiers**

```
internal open class TalkativeButton {  
    private fun yell() = println("Hey!")  
    protected fun whisper() = println("Let's talk!")  
}  
  
fun TalkativeButton.giveSpeech() { #1  
    yell() #2  
    whisper() #3  
}
```

Kotlin forbids you to reference the less-visible type `TalkativeButton` (`internal`, in this case) from the `public` function `giveSpeech`. This is a case of a general rule that requires all types used in the list of base types and type parameters of a class, or the signature of a method, to be as visible as the class or method itself. This rule ensures that you always have access to all types you might need to invoke the function or extend a class. To solve the problem, you can either make the `giveSpeech` extension function `internal` or make the `TalkativeButton` class `public`.

Note the difference in behavior for the `protected` modifier in Java and in Kotlin. In Java, you can access a `protected` member from the same package, but Kotlin doesn't allow that.

In Kotlin, visibility rules are simple, and a `protected` member is *only* visible in the class and its subclasses. Also note that extension functions of a class don't get access to its `private` or `protected` members. This is also why you can't call the `protected` `whisper` function from the extension function `giveSpeech`.

### **Kotlin's visibility modifiers and Java**

`public`, `protected`, and `private` modifiers in Kotlin are preserved when compiling to Java bytecode. You use such Kotlin declarations from Java code as if they were declared with the same visibility in Java. The only exception is a `private` class: it's compiled to a package-private declaration under the hood (you can't make a class `private` in Java).

But, you may ask, what happens with the `internal` modifier? There's no direct analogue in Java. Package-private visibility is a totally different thing: a module usually consists of several packages, and different modules may contain declarations from the same package. Thus an `internal` modifier becomes `public` in the bytecode.

This correspondence between Kotlin declarations and their Java analogues (or their bytecode representation) explains why sometimes you can access something from Java code that you can't access from Kotlin. For instance, you can access an `internal` class or a top-level declaration from Java code in another module, or a `protected` member from Java code in the same package

(similar to how you do that in Java).

But note that the names of `internal` members of a class are mangled. Technically, `internal` members can be used from Java, but they look ugly in the Java code. That helps avoid unexpected clashes in overrides when you extend a class from another module, and it prevents you from accidentally using `internal` classes.

One more difference in visibility rules between Kotlin and Java is that an outer class doesn't see `private` members of its inner (or nested) classes in Kotlin. Let's discuss inner and nested classes in Kotlin next and look at an example.

#### 4.1.4 Inner and nested classes: nested by default

If you want to encapsulate a helper class or want to keep code close to where it is used, you can declare a class inside another class. However, unlike in Java, Kotlin nested classes don't have access to the outer class instance, unless you specifically request that. Let's look at an example showing why this is important.

Imagine you want to define a `View` element, the state of which can be serialized. It may not be easy to serialize a view, but you can copy all the necessary data to another helper class. You declare the `State` interface that implements `Serializable`. The `View` interface declares `getCurrentState` and `restoreState` methods that can be used to save the state of a view.

**Listing 4.13. Declaring a view with serializable state**

```
interface State: Serializable

interface View {
    fun getCurrentState(): State
    fun restoreState(state: State) {}
}
```

It's handy to define a class that saves a button state in the `Button` class. Let's see how it can be done in Java (the similar Kotlin code will be shown in a moment).

#### **Listing 4.14. Implementing View in Java with an inner class**

```
/* Java */
public class Button implements View {
    @Override
    public State getCurrentState() {
        return new ButtonState();
    }

    @Override
    public void restoreState(State state) { /*...*/ }

    public class ButtonState implements State { /*...*/ }
}
```

You define the `ButtonState` class that implements the `State` interface and holds specific information for `Button`. In the `getCurrentState` method, you create a new instance of this class. In a real case, you'd initialize `ButtonState` with all necessary data.

What's wrong with this code? Why do you get a `java.io.NotSerializableException: Button` exception if you try to serialize the state of the declared button? That may look strange at first: the variable you serialize is `state` of the `ButtonState` type, not the `Button` type.

Everything becomes clear when you recall that in Java, when you declare a class in another class, it becomes an inner class by default. The `ButtonState` class in the example implicitly stores a reference to its outer `Button` class. That explains why `ButtonState` can't be serialized: `Button` isn't serializable, and the reference to it breaks the serialization of `ButtonState`.

To fix this problem, you need to declare the `ButtonState` class as `static`. Declaring a nested class as `static` removes the implicit reference from that class to its enclosing class.

In Kotlin, the default behavior of inner classes is the opposite of what we've just described, as shown next.

#### **Listing 4.15. Implementing View in Kotlin with a nested class**

```
class Button : View {
```

```

override fun getCurrentState(): State = ButtonState()

override fun restoreState(state: State) { /*...*/ }

class ButtonState : State { /*...*/ } #1
}

```

A nested class in Kotlin with no explicit modifiers is the same as a static nested class in Java. To turn it into an inner class so that it contains a reference to an outer class, you use the `inner` modifier. [4.3](#) describes the differences in this behavior between Java and Kotlin; and the difference between nested and inner classes is illustrated in [4.1](#).

**Table 4.3. Correspondence between nested and inner classes in Java and Kotlin**

Class A declared within another class B	In Java	In Kotlin
Nested class (doesn't store a reference to an outer class)	<code>static class A</code>	<code>class A</code>
Inner class (stores a reference to an outer class)	<code>class A</code>	<code>inner class A</code>

**Figure 4.1. Nested classes don't reference their outer class, whereas inner classes do.**



The syntax to reference an instance of an outer class in Kotlin also differs from Java. You write `this@Outer` to access the `Outer` class from the `Inner` class:

```

class Outer {
    inner class Inner {
        fun getOuterReference(): Outer = this@Outer
    }
}

```

You've learned the difference between inner and nested classes in Java and in Kotlin. Next, let's discuss how to create a hierarchy containing a limited number of classes.

#### 4.1.5 Sealed classes: defining restricted class hierarchies

Even before this chapter, you've already made your first acquaintance with class hierarchies in Kotlin. Recall the expression hierarchy example from [2.3.6](#), which you used to encode expressions like `(1 + 2) + 4`. The interface `Expr` is implemented by `Num`, which represents a number; and `Sum`, which represents a sum of two expressions. It's convenient to handle all the possible cases in a `when` expression. But you have to provide the `else` branch to specify what should happen if none of the other branches match:

##### **Listing 4.16. Expressions as interface implementations**

```

interface Expr
class Num(val value: Int) : Expr
class Sum(val left: Expr, val right: Expr) : Expr

```

```

fun eval(e: Expr): Int =
    when (e) {
        is Num -> e.value
        is Sum -> eval(e.right) + eval(e.left)
        else -> #1
            throw IllegalArgumentException("Unknown expression")
    }

```

Because you're using the `when` construct as an expression (that is, using its return value), it needs to be exhaustive: the Kotlin compiler forces you to check for the default option. In this example, you can't return something meaningful, so you throw an exception.

Always having to add a default branch isn't convenient. This can also turn into a problem: if you (or a colleague, or the author of a dependency you use) add a new subclass, the compiler won't alert you that you're missing a case. If you forget to add a branch to handle that new subclass, it will simply choose the default branch, which can lead to subtle bugs.

Kotlin comes with a solution to this problem: `sealed` classes. You mark a superclass with the `sealed` modifier, and that restricts the possibility of creating subclasses. All *direct* subclasses of a sealed class must be known at compile time and declared in the same package as the sealed class itself, and *all* subclasses need to be located within the same module.

Instead of using an interface, you could make `Expr` a `sealed` class, and have `Num` and `Sum` subclass it:

#### **Listing 4.17. Expressions as sealed classes**

```

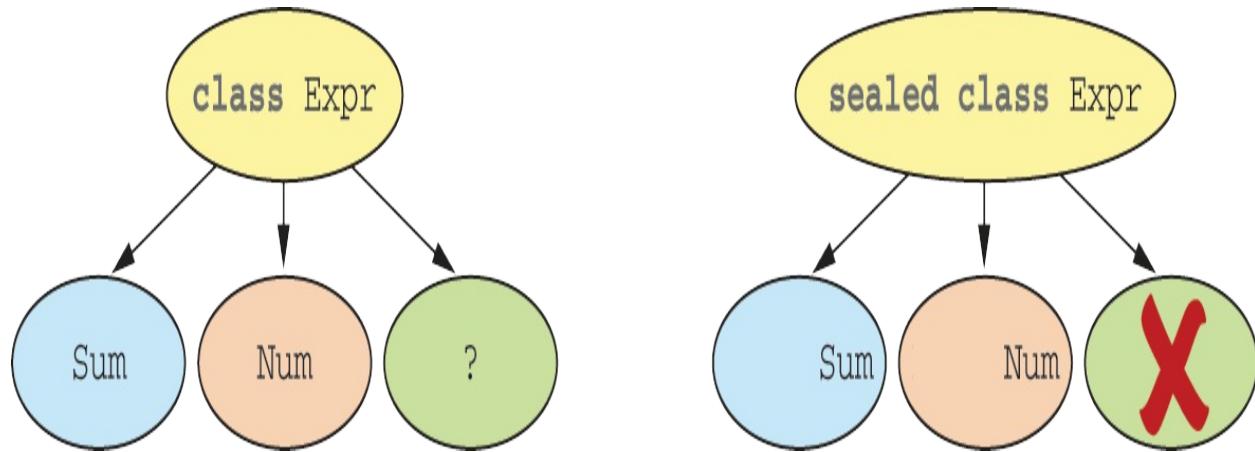
sealed class Expr #1
class Num(val value: Int) : Expr()
class Sum(val left: Expr, val right: Expr) : Expr() #2

fun eval(e: Expr): Int =
    when (e) { #3
        is Num -> e.value
        is Sum -> eval(e.right) + eval(e.left)
    }

```

If you handle all subclasses of a sealed class in a `when` expression, you don't need to provide the default branch—the compiler can ensure that you've covered all possible branches. Note that the `sealed` modifier implies that the class is abstract; you don't need an explicit `abstract` modifier, and can declare abstract members. The behavior of sealed classes is illustrated in [4.2](#).

**Figure 4.2.** All direct subclasses of a sealed class must be known at compile time.



When you use `when` with sealed classes and add a new subclass, the `when` expression returning a value fails to compile, which points you to the code that must be changed—for example, when you define a multiplication operator `Mul`, but don't handle it in the `eval` function:

**Listing 4.18.** Adding a new class to a sealed hierarchy

```

sealed class Expr
class Num(val value: Int) : Expr()
class Sum(val left: Expr, val right: Expr) : Expr()
class Mul(val left: Expr, val right: Expr): Expr()

fun eval(e: Expr): Int =
    when (e) {
        is Num -> e.value
        is Sum -> eval(e.right) + eval(e.left)
        // ERROR: 'when' expression must be exhaustive,
        // add necessary 'is Mul' branch or 'else' branch instead
    }

```

Besides classes, you can also use the `sealed` modifier to define a sealed

interface. Sealed interfaces follow the same rules: once the module that contains the sealed interface is compiled, no new implementations for it can be provided:

```
sealed interface Toggleable {  
    fun toggle() #1  
}  
  
class LightSwitch: Toggleable { #2  
    override fun toggle() = println("Lights!")  
}  
  
class Camera: Toggleable { #2  
    override fun toggle() = println("Camera!")  
}
```

Handling all implementations of a sealed interface in a when statement also means that you do not have to specify an else branch.

As you'll recall, in Kotlin, you use a colon both to extend a class and to implement an interface:

```
class Num(val value: Int) : Expr() #1  
class LightSwitch: Toggleable #2
```

What we have yet to discuss is the meaning of the parentheses after the class name in `Expr()`. We'll talk about it in the next section, which covers initializing classes in Kotlin.

[3] Technically, it does return a value: `Unit`, the Kotlin equivalent of Java's `void`. You'll take a detailed look at it in [8.1.6](#)

## 4.2 Declaring a class with nontrivial constructors or properties

In object-oriented languages, classes can typically have one or more constructors. Kotlin is the same, but makes an important, explicit distinction: it differentiates between a *primary* constructor (which is usually the main, concise way to initialize a class and is declared outside of the class body) and a *secondary* constructor (which is declared in the class body). It also allows

you to put additional initialization logic in *initializer blocks*. First we'll look at the syntax of declaring the primary constructor and initializer blocks, and then we'll explain how to declare several constructors. After that, we'll talk more about properties.

### 4.2.1 Initializing classes: primary constructor and initializer blocks

In [2.2](#), you saw how to declare a simple class:

```
class User(val nickname: String)
```

Typically, all declarations in a class go inside curly braces, so you may wonder why this class is different. It has no curly braces, and instead has only a declaration in parentheses. This block of code surrounded by parentheses is called a *primary constructor*. It serves two purposes: specifying constructor parameters and defining properties that are initialized by those parameters. Let's unpack what happens here and look at the most explicit code you can write that does the same thing:

```
class User constructor(_nickname: String) { #1
    val nickname: String

    init { #2
        nickname = _nickname
    }
}
```

In this example, you see two new Kotlin keywords: `constructor` and `init`. The `constructor` keyword begins the declaration of a primary or secondary constructor. The `init` keyword introduces an *initializer block*. Such blocks contain initialization code that's executed when the class is created, and are intended to be used together with primary constructors. Because the primary constructor has a constrained syntax, it can't contain the initialization code; that's why you have initializer blocks. If you want to, you can declare several initializer blocks in one class.

The underscore in the constructor parameter `_nickname` serves to distinguish the name of the property from the name of the constructor parameter. An

alternative possibility is to use the same name and write `this` to remove the ambiguity: `this.nickname = nickname`.

In this example, you don't need to place the initialization code in the initializer block, because it can be combined with the declaration of the `nickname` property. You can also omit the `constructor` keyword if there are no annotations or visibility modifiers on the primary constructor. If you apply those changes, you get the following:

```
class User(_nickname: String) { #1
    val nickname = _nickname #2
}
```

This is another way to declare the same class. Note how you can refer to primary constructor parameters in property initializers and in initializer blocks.

The two previous examples declared the property by using the `val` keyword in the body of the class. If the property is initialized with the corresponding constructor parameter, the code can be simplified by adding the `val` keyword before the parameter. This replaces the property definition in the class body:

```
class User(val nickname: String) #1
```

All these declarations of the `User` class achieve the same thing, but the last one uses the most concise syntax.

You can declare default values for constructor parameters just as you can for function parameters:

```
class User(
    val nickname: String,
    val isSubscribed: Boolean = true #1
)
```

To create an instance of a class, you call the constructor directly, without any extra keywords like `new`. Let's demonstrate how our potential users Alice got subscribed to the mailing list by default, Bob and Carol read the terms and conditions carefully and deselected the default option, and Dave is explicitly interested in what our marketing department has to share:

```
fun main() {
    val alice = User("Alice") #1
    println(alice.isSubscribed)
    // true
    val bob = User("Bob", false) #2
    println(bob.isSubscribed)
    // false
    val carol = User("Carol", isSubscribed = false) #3
    println(carol.isSubscribed)
    // false
    val dave = User(nickname = "Dave", isSubscribed = true) #4
    println(dave.isSubscribed)
    // true
}
```



### Note

If all the constructor parameters have default values, the compiler generates an additional constructor without parameters that uses all the default values. That makes it easier to use Kotlin with libraries that instantiate classes via parameterless constructors. If you have Java code that needs to call a Kotlin constructor with some default parameters, you can specify the constructor as `@JvmOverloads` `constructor`. This instructs the compiler to generate the appropriate overloads for use from Java, as discussed in [3.2.2](#).

If the constructor of a superclass takes arguments, then the primary constructor of your class also needs to initialize them. You can do so by providing the superclass constructor parameters after the superclass reference in the base class list:

```
open class User(val nickname: String) { /* ... */ }

class SocialUser(nickname: String) : User(nickname) { /* ... */ }
```

If you don't declare any constructors for a class, a default constructor without parameters that does nothing will be generated for you:

```
open class Button #1
```

If you inherit from the `Button` class and don't provide any constructors, you have to explicitly invoke the constructor of the superclass even if it doesn't

have any parameters. That's why you need empty parentheses after the name of the superclass:

```
class RadioButton: Button()
```

Note the difference with interfaces: interfaces don't have constructors, so if you implement an interface, you never put parentheses after its name in the supertype list.

If you want to ensure that your class can't be instantiated by code outside the class itself, you have to make the constructor `private`. Here's how you make the primary constructor `private`:

```
class Secret private constructor(private val agentName: String) {
```

Because the `Secret` class has only a `private` constructor, code outside the class can't instantiate it. Later, in [4.4.2](#), we'll talk about companion objects, which may be a good place to call such constructors.

#### Alternatives to private constructors

In Java, you can use a `private` constructor that prohibits class instantiation to express a more general idea: that the class is a container of static utility members or is a singleton. Kotlin has built-in language features for these purposes. You use top-level functions (which you saw in [3.2.3](#)) as static utilities. To express singletons, you use object declarations, as you'll see in [4.4.1](#).

In most real use cases, the constructor of a class is straightforward: it contains no parameters or assigns the parameters to the corresponding properties. That's why Kotlin has concise syntax for primary constructors: it works great for the majority of cases. But life isn't always that easy, so Kotlin allows you to define as many constructors as your class needs. Let's see how this works.

## 4.2.2 Secondary constructors: initializing the superclass in different ways

Generally speaking, classes with multiple constructors are much less

common in Kotlin code than in Java. The majority of situations where you'd need overloaded constructors are covered by Kotlin's support for default parameter values and named argument syntax.



**Tip**

Don't declare multiple secondary constructors to overload and provide default values for arguments. Instead, specify default values directly.

But there are still situations when multiple constructors are required. The most common one comes up when you need to extend a framework class that provides multiple constructors that initialize the class in different ways. Take for example a `Downloader` class that's declared in Java and that has two constructors—one taking a `String`, the other one a `URI` as its parameter:

```
import java.net.URI;

public class Downloader {
    public Downloader(String url) {
        // some code
    }

    public Downloader(URI uri) {
        // some code
    }
}
```

In Kotlin, the same declaration would look as follows:

```
open class Downloader {
    constructor(url: String?) { #1
        // some code
    }

    constructor(uri: URI?) {
        // some code
    }
}
```

This class doesn't declare a primary constructor (as you can tell because there are no parentheses after the class name in the class header), but it declares

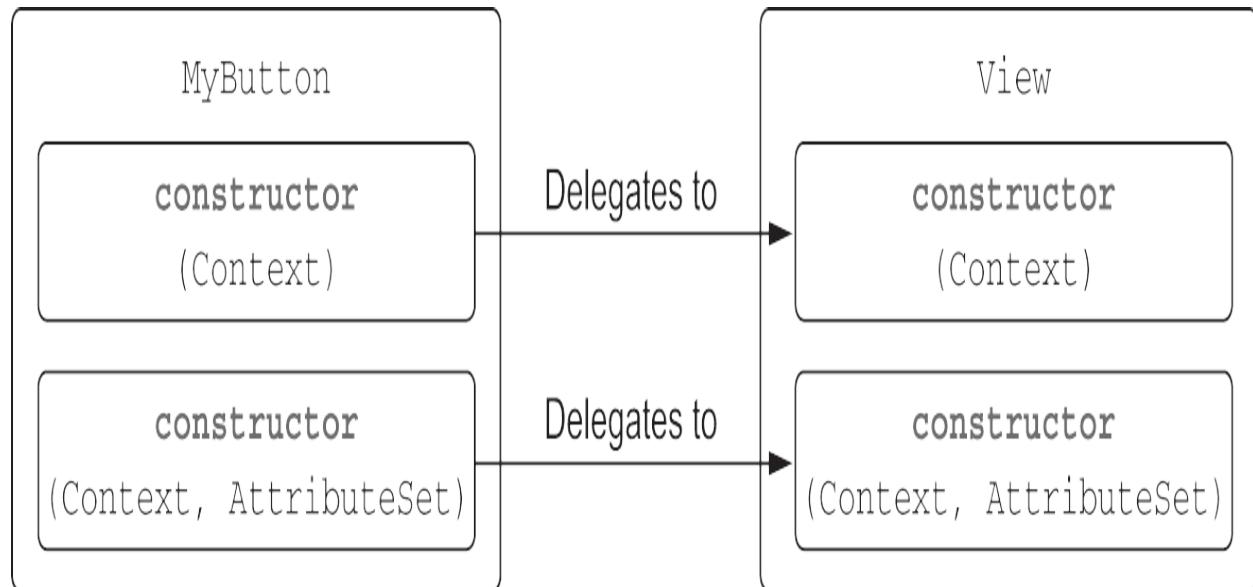
two secondary constructors. A secondary constructor is introduced using the `constructor` keyword. You can declare as many secondary constructors as you need.

If you want to extend this class, you can declare the same constructors:

```
class MyDownloader : Downloader {  
    constructor(url: String?) : super(url) {  
        // ... #1  
    }  
    constructor(uri: URI?) : super(uri) { #1  
        // ...  
    }  
}
```

Here you define two constructors, each of which calls the corresponding constructor of the superclass using the `super()` keyword. This is illustrated in [4.3](#); an arrow shows which constructor is delegated to.

**Figure 4.3. Using different superclass constructors**



Just as in Java, you also have an option to call another constructor of your own class from a constructor, using the `this()` keyword. Here's how this works:

```
class MyDownloader : Downloader {
```

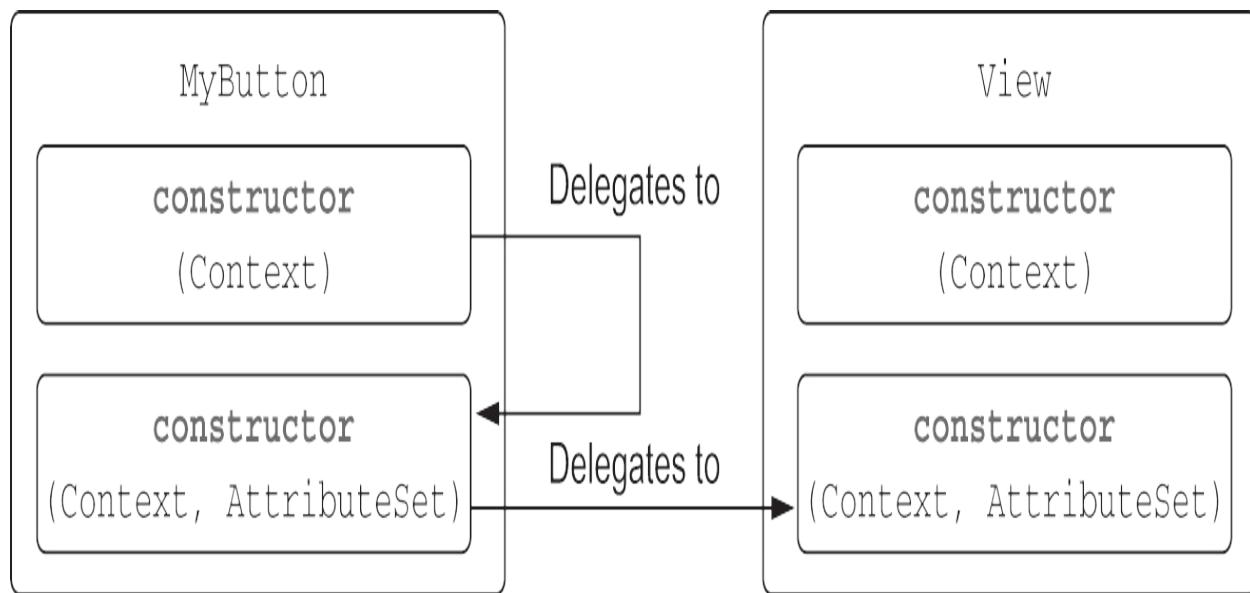
```

    constructor(url: String?) : this(URI(url)) #1
    constructor(uri: URI?) : super(uri)
}

```

You change the `MyDownloader` class so that one of the constructors delegates to the other constructor of the same class (using `this`), creating a `URI` object from the `url` string, as shown in [4.4](#). The second constructor continues to call `super()`.

**Figure 4.4. Delegating to a constructor of the same class.**



If the class has no primary constructor, then each secondary constructor has to initialize the base class or delegate to another constructor that does so. Thinking in terms of the previous figures, each secondary constructor must have an outgoing arrow starting a path that ends at any constructor of the base class.

The main use case for when you need to use secondary constructors is Java interoperability. But there's another possible case: when you have multiple ways to create instances of your class, with different parameter lists. We'll discuss an example in [4.4.2](#).

We've discussed how to define nontrivial constructors. Now let's turn our attention to nontrivial properties.

### 4.2.3 Implementing properties declared in interfaces

In Kotlin, an interface can contain abstract property declarations. Here's an example of an interface definition with such a declaration:

```
interface User {  
    val nickname: String  
}
```

This means classes implementing the `User` interface need to provide a way to obtain the value of `nickname`. The interface doesn't specify whether the value should be stored in a backing field or obtained through a getter. Therefore, the interface itself doesn't contain any state. Classes implementing the interface may store the value if they need to, or simply compute it when accessed, as you have seen in [2.2.2](#).

Let's look at a few possible implementations for the interface: `PrivateUser`, who fills in only their nickname; `SubscribingUser`, who apparently was forced to provide an email to register; and `SocialUser`, who rashly shared their account ID from a social network. All of these classes implement the abstract property in the interface in different ways.

**Listing 4.19. Implementing an interface property**

```
class PrivateUser(override val nickname: String) : User #1  
  
class SubscribingUser(val email: String) : User {  
    override val nickname: String  
        get() = email.substringBefore('@') #2  
}  
  
class SocialUser(val accountId: Int) : User {  
    override val nickname = getNameFromSocialNetwork(accountId) #  
}  
  
fun main() {  
    println(PrivateUser("test@kotlinlang.org").nickname)  
    // test@kotlinlang.org  
    println(SubscribingUser("test@kotlinlang.org").nickname)  
    // test  
}
```

For `PrivateUser`, you use the concise syntax to declare a property directly in the primary constructor. This property implements the abstract property from `User`, so you mark it as `override`.

For `SubscribingUser`, the `nickname` property is implemented through a custom getter. This property doesn't have a backing field to store its value; it has only a getter that calculates a nickname from the email on every invocation.

For `SocialUser`, you assign the value to the `nickname` property in its initializer. You use a `getNameFromSocialNetwork` function that's supposed to return the name of a Social user given their account ID. (Assume that it's defined somewhere else.) This function is costly: it needs to establish a connection with Social to get the desired data. That's why you decide to invoke it once during the initialization phase.

Pay attention to the different implementations of `nickname` in `SubscribingUser` and `SocialUser`. Although they look similar, the property of `SubscribingUser` has a custom getter that calculates `substringBefore` on every access, whereas the property in `SocialUser` has a backing field that stores the data computed during the class initialization.

In addition to abstract property declarations, an interface can contain properties with getters and setters, as long as they don't reference a backing field. (A backing field would require storing state in an interface, which isn't allowed.) Let's look at an example:

```
interface EmailUser {  
    val email: String  
    val nickname: String  
        get() = email.substringBefore('@') #1  
}
```

This interface contains the abstract property `email`, as well as the `nickname` property with a custom getter. The first property must be overridden in subclasses, whereas the second one can be inherited.

## When to prefer properties to functions

In 2.2.2, we already briefly touched upon when to declare a function without parameters, or a read-only property with a custom getter. We established that *characteristics* of a class should generally be declared as a property, and *behavior* should be declared as methods. There are a few additional stylistic conventions in Kotlin for using read-only properties instead of functions. If the code in question:

- doesn't throw exceptions,
- is cheap to calculate (or cached on the first run), and
- returns the same result across multiple invocations if the object state hasn't changed,

prefer a property over a function. Otherwise, consider using a function instead.

Unlike properties implemented in interfaces, properties implemented in classes have full access to backing fields. Let's see how you can refer to them from accessors.

#### 4.2.4 Accessing a backing field from a getter or setter

You've seen a few examples of two kinds of properties: properties that store values and properties with custom accessors that calculate values on every access. Now let's see how you can combine the two and implement a property that stores a value and provides additional logic that's executed when the value is accessed or modified. To support that, you need to be able to access the property's backing field from its accessors.

Let's say you're building a contact management system that keeps track of User objects, their name, as well as their address. Let's say you want to log any change of data stored in a property like address. To do so, you declare a mutable property, and execute the additional code on each setter access:

**Listing 4.20. Accessing the backing field in a setter**

```
class User(val name: String) {  
    var address: String = "unspecified"  
        set(value: String) {
```

```

        println(
        """
            Address was changed for $name:
            "$field" -> "$value". #1
            """.trimIndent()
        )
        field = value #2
    }
}

fun main() {
    val user = User("Alice")
    user.address = "Christoph-Rapparini-Bogen 23"
    // Address was changed for Alice:
    // "unspecified" -> "Christoph-Rapparini-Bogen 23".
}

```

You change a property value as usual by saying `user.address = "new value"`, which invokes its setter under the hood. In this example, the setter is redefined, so the additional logging code is executed (for simplicity, in this case you print it out).

In the body of the setter, you use the special identifier `field` to access the value of the backing field. In a getter, you can only read the value; and in a setter, you can both read and modify it.

Note that if your getter or setter for a mutable property is trivial, you can choose to define only the accessor where you need custom behavior without redefining the other one. For example, the getter in [4.20](#) is trivial and only returns the field value, so you don't need to redefine it.

You may wonder what the difference is between making a property that has a backing field and one that doesn't. The way you access the property doesn't depend on whether it has a backing field. The compiler will generate the backing field for the property if you either reference it explicitly or use the default accessor implementation. If you provide custom accessor implementations that don't use `field` (for the getter if the property is a `val` and for both accessors if it's a mutable property), the compiler understands that the property doesn't need to store any information itself, so no backing field will be generated. [4.21](#) illustrates this with a property `ageIn2050` that is purely defined in terms of the `birthYear` of a `Person`:

**Listing 4.21. A property that doesn't store information itself does not need and has no backing field.**

```
class Person(var birthYear: Int) {  
    var ageIn2050  
        get() = 2050 - birthYear #1  
        set(value) {  
            birthYear = 2050 - value #2  
        }  
}
```

Sometimes you don't need to change the default implementation of an accessor, but you still need to change its visibility. Let's see how you can do this.

## 4.2.5 Changing accessor visibility

The accessor's visibility by default is the same as the property's. But you can change this if you need to, by putting a visibility modifier before the get or set keyword. To see how you can use it, let's look at an example—a small class called `LengthCounter` that keeps track of the total length of words that are added to it:

**Listing 4.22. Declaring a property with a private setter**

```
class LengthCounter {  
    var counter: Int = 0  
        private set #1  
  
    fun addWord(word: String) {  
        counter += word.length  
    }  
}
```

The `counter` property which holds the total length is `public`, because it's part of the API the class provides to its clients. But you need to make sure it's only modified in the class, because otherwise external code could change it and store an incorrect value. Therefore, you let the compiler generate a getter with the default visibility, and you change the visibility of the setter to `private`.

Here's how you can use this class:

```
fun main() {
    val lengthCounter = LengthCounter()
    lengthCounter.addWord("Hi!")
    println(lengthCounter.counter)
    // 3
}
```

You create an instance of `LengthCounter`, and then you add a word "Hi!" of length 3. Now the `counter` property stores 3. As anticipated, attempting to write to the property from outside the class results in a compile-time error:

```
fun main() {
    lengthCounter.counter = 0
    // Error: Cannot assign to 'counter': the setter is private i
}
```

### More about properties later

Later in the book, we'll continue our discussion of properties. Here are some references:

- The `lateinit` modifier on a non-null property specifies that this property is initialized later, after the constructor is called, which is a common case for some frameworks. This feature will be covered in [7.1.8](#).
- Lazy initialized properties, as part of the more general *delegated properties* feature, will be covered in [9.5](#).
- For compatibility with Java frameworks, you can use annotations that emulate Java features in Kotlin. For instance, the `@JvmField` annotation on a property exposes a public field without accessors. You'll learn more about annotations in [Chapter 12](#).
- The `const` modifier makes working with annotations more convenient and lets you use a property of a primitive type or `String` as an annotation argument. [Chapter 12](#) provides details.

That concludes our discussion of writing nontrivial constructors and properties in Kotlin. Next, you'll see how to make value-object classes even friendlier, using the concept of data classes.

## 4.3 Compiler-generated methods: data classes and class delegation

The Java platform defines a number of methods that need to be present in many classes and are usually implemented in a mechanical way, such as `equals` (indicating whether two objects are equal to each other), `hashCode` (providing a hash code for the object, as required by data structures like hash maps), and `toString` (returning a textual representation of the object).

Fortunately, IDEs can automate the generation of these methods, so you usually don't need to write them by hand. But in this case, your codebase still contains boilerplate code that you have to maintain. The Kotlin compiler takes things a step further: it can perform the mechanical code generation behind the scenes, without cluttering your source code files with the results.

You already saw how this works for trivial class constructor and property accessors. Let's look at more examples of cases where the Kotlin compiler generates typical methods that are useful for simple data classes and greatly simplifies the class-delegation pattern.

### 4.3.1 Universal object methods

All Kotlin classes have several methods you may want to override, just like in Java: `toString`, `equals`, and `hashCode`. Let's look at what these methods are and how Kotlin can help you generate their implementations automatically. As a starting point, you'll use a simple `Customer` class that stores a customer's name and postal code.

**Listing 4.23. Initial declaration of the `Customer` class**

```
class Customer(val name: String, val postalCode: Int)
```

Let's see how class instances are represented as strings.

#### String representation: `toString()`

All classes in Kotlin, just as in Java, provide a way to get a string representation of the class's objects. This is primarily used for debugging and logging, although you can use this functionality in other contexts as well. By default, the string representation of an object looks like `Customer@5e9f23b4` (the class name and the memory address at which the object is stored), which in practice isn't very useful. To change this, you need to override the `toString` method.

**Listing 4.24. Implementing `toString()` for `Customer`**

```
class Customer(val name: String, val postalCode: Int) {  
    override fun toString() = "Customer(name=$name, postalCode=$p  
}
```

Now the representation of a customer looks like this:

```
fun main() {  
    val customer1 = Customer("Alice", 342562)  
    println(customer1)  
    // Customer(name=Alice, postalCode=342562)  
}
```

Much more informative, isn't it?

## Object equality: `equals()`

All computations with the `Customer` class take place outside of it. This class just stores the data; it's meant to be plain and transparent. Nevertheless, you may have some requirements for the behavior of such a class. For example, suppose you want the objects to be considered equal if they contain the same data:

```
fun main() {  
    val customer1 = Customer("Alice", 342562)  
    val customer2 = Customer("Alice", 342562)  
    println(customer1 == customer2) #1  
    // false  
}
```

You see that the objects aren't equal. To address this, means you must override `equals` for the `Customer` class.

## **== for equality**

In Java, you can use the `==` operator to compare primitive and reference types. If applied to primitive types, Java's `==` compares values, whereas `==` on reference types compares references. Thus, in Java, there's the well-known practice of always calling `equals`, and there's the well-known problem of forgetting to do so.

In Kotlin, the `==` operator is the default way to compare two objects: it compares their values by calling `equals` under the hood. Thus, if `equals` is overridden in your class, you can safely compare its instances using `==`. For reference comparison, you can use the `===` operator, which works exactly the same as `==` in Java by comparing the object references. In other words, `==` checks whether two references point to the same object in memory.

Let's look at the changed `Customer` class. In Kotlin, the `equals` function takes the nullable parameter `other` of type `Any`—the superclass of all classes in Kotlin, which you'll get to know more intimately in [8.1.5](#):

**Listing 4.25. Implementing `equals()` for `Customer`**

```
class Customer(val name: String, val postalCode: Int) {  
    override fun equals(other: Any?): Boolean { #1  
        if (other == null || other !is Customer) #2  
            return false  
        return name == other.name && #3  
                postalCode == other.postalCode  
    }  
    override fun toString() = "Customer(name=$name, postalCode=$p  
}
```

Just to remind you, the `is` check checks whether a value has the specified type, and smart casts `other` to the `Customer` type (it is the analogue of `instanceof` in Java). Like the `!in` operator, which is a negation for the `in` check (we discussed both in [2.4.4](#)), the `!is` operator denotes the negation of the `is` check. Such operators make your code easier to read. In [7](#), we'll discuss nullable types in detail and why the condition `other == null || other !is Customer` can be simplified to `other !is Customer`.

Because in Kotlin the `override` modifier is mandatory, you're protected from

accidentally writing `fun equals(other: Customer)`, which would add a new method instead of overriding `equals`. After you override `equals`, you may expect that customers with the same property values are equal. Indeed, the equality check `customer1 == customer2` in the previous example returns `true` now. But if you want to do more complicated things with customers, it doesn't work. The usual interview question is, "What's broken, and what's the problem?" You may say that the problem is that `hashCode` is missing. That is in fact the case, and we'll now discuss why this is important.

## Hash containers: `hashCode()`

The `hashCode` method should be always overridden together with `equals`. This section explains why.

Let's create a set with one element: a customer named Alice. Then you create a new `Customer` instance containing the same data and check whether it's contained in the set. You'd expect the check to return `true`, because the two instances are equal, but in fact it returns `false`:

```
fun main() {
    val processed = hashSetOf(Customer("Alice", 342562))
    println(processed.contains(Customer("Alice", 342562)))
    // false
}
```

The reason is that the `Customer` class is missing the `hashCode` method. Therefore, it violates the general `hashCode` contract: if two objects are equal, they must have the same hash code. The `processed` set is a `HashSet`. Values in a `HashSet` are compared in an optimized way: at first their hash codes are compared, and then, only if their hash codes are equal, the actual values are compared. The hash codes are different for two different instances of the `Customer` class in the previous example, so the set decides that it doesn't contain the second object, even though `equals` would return `true`. Therefore, if the rule isn't followed, data structures like `HashSet` can't work correctly with such objects.

To fix that, you have to provide an appropriate implementation of `hashCode` to the class.

**Listing 4.26. Implementing hashCode() for Customer**

```
class Customer(val name: String, val postalCode: Int) {  
    /* ... */  
    override fun hashCode(): Int = name.hashCode() * 31 + postalC  
}
```

Now you have a class that works as expected in all scenarios—but notice how much code you’ve had to write. Fortunately, the Kotlin compiler can help you by generating all of those methods automatically. Let’s see how you can ask it to do that.

### 4.3.2 Data classes: autogenerated implementations of universal methods

If you want your class to be a convenient holder for your data, you need to override these methods: `toString`, `equals`, and `hashCode`. Usually, the implementations of those methods are straightforward, and IDEs like IntelliJ IDEA can help you generate them automatically and verify that they’re implemented correctly and consistently.

The good news is, you don’t have to generate all of these methods in Kotlin. If you add the modifier `data` to your class, the necessary methods are automatically generated for you.

**Listing 4.27. Customer as a data class**

```
data class Customer(val name: String, val postalCode: Int)
```

Easy, right? Now you have a class that overrides all the standard methods you usually need:

- `equals` for comparing instances
- `hashCode` for using them as keys in hash-based containers such as `HashMap`
- `toString` for generating string representations showing all the fields in declaration order

The `equals` and `hashCode` methods take into account all the properties

declared in the primary constructor. The generated `equals` method checks that the values of all the properties are equal. The `hashCode` method returns a value that depends on the hash codes of all the properties. Note that properties that aren't declared in the primary constructor don't take part in the equality checks and hash code calculation.

[4.28](#) shows an example to help you believe this magic.

**Listing 4.28. The `Customer` data class has implementations for all standard methods out of the box.**

```
fun main() {
    val c1 = Customer("Sam", 11521)
    val c2 = Customer("Mart", 15500)
    val c3 = Customer("Sam", 11521)
    println(c1)
    // Customer(name=Sam, postalCode=11521)
    println(c1 == c2)
    // false
    println(c1 == c3)
    // true
    println(c1.hashCode())
    // 2580770
}
```

This isn't a complete list of useful methods generated for data classes. The next section reveals one more, and [9.4](#) fills in the rest.

## Data classes and immutability: the `copy()` method

Note that even though the properties of a data class aren't required to be `val` —you can use `var` as well—it's strongly recommended that you use only read-only properties, making the instances of the data class *immutable*. This is required if you want to use such instances as keys in a `HashMap` or a similar container, because otherwise the container could get into an invalid state if the object used as a key was modified after it was added to the container. Immutable objects are also much easier to reason about, especially in multithreaded code: once an object has been created, it remains in its original state, and you don't need to worry about other threads modifying the object while your code is working with it.

To make it even easier to use data classes as immutable objects, the Kotlin compiler generates one more method for them: a method that allows you to *copy* the instances of your classes, changing the values of some properties. Creating a copy is usually a good alternative to modifying the instance in place: the copy has a separate lifecycle and can't affect the places in the code that refer to the original instance. Here's what the copy method would look like if you implemented it manually:

```
class Customer(val name: String, val postalCode: Int) {  
    /* ... */  
    fun copy(name: String = this.name,  
            postalCode: Int = this.postalCode) =  
        Customer(name, postalCode)  
}
```

And here's how the copy method can be used:

```
fun main() {  
    val bob = Customer("Bob", 973293)  
    println(bob.copy(postalCode = 382555))  
    // Customer(name=Bob, postalCode=382555)  
}
```

### Kotlin data classes and Java records

In Java 14, *records* were first introduced. They are conceptually very similar to Kotlin's data classes, in that they hold immutable groups of values. Records also auto-generate some methods based on its values, such as `toString`, `hashCode` and `equals`. Other convenience functions, like the `copy` method, are absent in records.

Compared to Kotlin data classes, Java records also impose more structural restrictions:

- All properties are required to be `private` and `final`.
- A record can't extend a superclass.
- You can't specify additional properties in the class body.

For interoperability purposes, you can declare record classes in Kotlin by annotating a `data` class with the `@JvmRecord` annotation. In this case, the

data class needs to adhere to the same structural restrictions that apply to records.

You've seen how the `data` modifier makes value-object classes more convenient to use. Now let's talk about the other Kotlin feature that lets you avoid IDE-generated boilerplate code: class delegation.

### 4.3.3 Class delegation: using the "by" keyword

A common problem in the design of large object-oriented systems is fragility caused by implementation inheritance. When you extend a class and override some of its methods, your code becomes dependent on the implementation details of the class you're extending. When the system evolves and the implementation of the base class changes or new methods are added to it, the assumptions about its behavior that you've made in your class can become invalid, so your code may end up not behaving correctly.

The design of Kotlin recognizes this problem and treats classes as `final` by default, as you have seen in [4.1.2](#). This ensures that only those classes that are designed for extensibility can be inherited from. When working on such a class, you see that it's open, and you can keep in mind that modifications need to be compatible with derived classes.

But often you need to add behavior to another class, even if it wasn't designed to be extended. A commonly used way to implement this is known as the *Decorator* design pattern. The essence of the pattern is that a new class is created, implementing the same interface as the original class and storing the instance of the original class as a field. Methods in which the behavior of the original class doesn't need to be modified are forwarded to the original class instance.

One downside of this approach is that it requires a fairly large amount of boilerplate code (so much that IDEs like IntelliJ IDEA have dedicated features to generate that code for you). For example, this is how much code you need for a decorator that implements an interface as simple as `Collection`, even when you don't modify any behavior:

```
class DelegatingCollection<T> : Collection<T> {
```

```

private val innerList = arrayListOf<T>()

override val size: Int get() = innerList.size
override fun isEmpty(): Boolean = innerList.isEmpty()
override fun contains(element: T): Boolean = innerList.contains(element)
override fun iterator(): Iterator<T> = innerList.iterator()
override fun containsAll(elements: Collection<T>): Boolean =
    innerList.containsAll(elements)
}

```

The good news is that Kotlin includes first-class support for delegation as a language feature. Whenever you’re implementing an interface, you can say that you’re *delegating* the implementation of the interface to another object, using the `by` keyword. Here’s how you can use this approach to rewrite the previous example:

```

class DelegatingCollection<T>(
    innerList: Collection<T> = ArrayList<T>()
) : Collection<T> by innerList

```

All the method implementations in the class are gone. The compiler will generate them, and the implementation is similar to that in the `DelegatingCollection` example. Because there’s little interesting content in the code, there’s no point in writing it manually when the compiler can do the same job for you automatically.

Now, when you need to change the behavior of some methods, you can override them, and your code will be called instead of the generated methods. You can leave out methods for which you’re satisfied with the default implementation of delegating to the underlying instance.

Let’s see how you can use this technique to implement a collection that counts the number of attempts to add an element to it. For example, if you’re performing some kind of deduplication, you can use such a collection to measure how efficient the process is, by comparing the number of attempts to add an element with the resulting size of the collection.

#### **Listing 4.29. Using class delegation**

```

class CountingSet<T>(
    private val innerSet: MutableCollection<T> = HashSet<T>()
)

```

```

) : MutableCollection<T> by innerSet { #1

    var objectsAdded = 0

    override fun add(element: T): Boolean { #2
        objectsAdded++
        return innerSet.add(element)
    }

    override fun addAll(elements: Collection<T>): Boolean { #2
        objectsAdded += elements.size
        return innerSet.addAll(elements)
    }
}

fun main() {
    val cset = CountingSet<Int>()
    cset.addAll(listOf(1, 1, 2))
    println("Added ${cset.objectsAdded} objects, ${cset.size} uni
        // Added 3 objects, 2 uniques.
}

```

As you see, you override the `add` and `addAll` methods to increment the count, and you delegate the rest of the implementation of the `MutableCollection` interface to the container you're wrapping.

The important part is that you aren't introducing any dependency on how the underlying collection is implemented. For example, you don't care whether that collection implements `addAll` by calling `add` in a loop, or if it uses a different implementation optimized for a particular case. You have full control over what happens when the client code calls your class, and you rely only on the documented API of the underlying collection to implement your operations, so you can rely on it continuing to work.

You've now seen how the Kotlin compiler can generate useful methods for classes. Let's proceed to the final big part of Kotlin's class story: the `object` keyword and the different situations in which it comes into play.

## 4.4 The "object" keyword: declaring a class and creating an instance, combined

The `object` keyword comes up in Kotlin in a number of cases, but they all share the same core idea: the keyword defines a class and creates an instance (in other words, an object) of that class at the same time. Let's look at the different situations when it's used:

- *Object declaration* is a way to define a singleton.
- *Companion objects* can contain factory methods and other methods that are related to this class but don't require a class instance to be called. Their members can be accessed via class name.
- *Object expressions* are used instead of Java's anonymous inner class.

It's time to discuss these Kotlin features in detail.

#### 4.4.1 Object declarations: singletons made easy

A fairly common occurrence in the design of object-oriented systems is a class for which you need only one instance. This is usually implemented using the *Singleton* pattern in languages like Java: you define a class with a private constructor and a static field holding the only existing instance of the class.

Kotlin provides first-class language support for this using the *object declaration* feature. The object declaration combines a *class declaration* and a declaration of a *single instance* of that class.

For example, you can use an object declaration to represent the payroll of an organization. You probably don't have multiple payrolls, so—depending on the overall complexity of your application—using an object for could be reasonable:

```
object Payroll {  
    val allEmployees = mutableListOf<Person>()  
  
    fun calculateSalary() {  
        for (person in allEmployees) {  
            /* ... */  
        }  
    }  
}
```

Object declarations are introduced with the `object` keyword. An object declaration effectively defines a class and a variable of that class with the same name in a single statement.

Just like a class, an object declaration can contain declarations of properties, methods, initializer blocks, and so on. The only things that aren't allowed are constructors (either primary or secondary). Unlike instances of regular classes, object declarations are created immediately at the point of definition, not through constructor calls from other places in the code. Therefore, defining a constructor for an object declaration doesn't make sense. Likewise, it also means that any initial state you want to give to your object declaration needs to be provided as a part of that object's body.

And just like a variable, an object declaration lets you call methods and access properties by using the object name to the left of the `.` character:

```
Payroll.allEmployees.add(Person(* ... *))
```

```
Payroll.calculateSalary()
```

Object declarations can also inherit from classes and interfaces. This is often useful when the framework you're using requires you to implement an interface, but your implementation doesn't contain any state. For example, let's take the `Comparator` interface. A `Comparator` implementation receives two objects and returns an integer indicating which of the objects is greater. Comparators almost never store any data, so you usually need just a single `Comparator` instance for a particular way of comparing objects. That's a perfect use case for an object declaration.

As a specific example, let's implement a comparator that compares file paths case-insensitively.

#### **Listing 4.30. Implementing Comparator with an object**

```
object CaseInsensitiveFileComparator : Comparator<File> {
    override fun compare(file1: File, file2: File): Int {
        return file1.path.compareTo(file2.path,
            ignoreCase = true)
    }
}
```

```
fun main() {
    println(
        CaseInsensitiveFileComparator.compare(
            File("/User"), File("/user")
        )
    )
    // 0
}
```

You use singleton objects in any context where an ordinary object (an instance of a class) can be used. For example, you can pass this object as an argument to a function that takes a `Comparator`:

```
fun main() {
    val files = listOf(File("/Z"), File("/a"))
    println(files.sortedWith(CaseInsensitiveFileComparator))
    // [/a, /Z]
}
```

Here you're using the `sortedWith` function, which returns a list sorted according to the specified `Comparator`.

### **Singletons and dependency injection**

Just like the Singleton pattern, object declarations aren't always ideal for use in large software systems. They're great for small pieces of code that have few or no dependencies, but not for large components that interact with many other parts of the system. The main reason is that you don't have any control over the instantiation of objects, and you can't specify parameters for the constructors.

This means you can't replace the implementations of the object itself, or other classes the object depends on, in unit tests or in different configurations of the software system. If you need that ability, you should use regular Kotlin classes and dependency injection.

You can also declare objects in a class. Such objects also have just a single instance; they don't have a separate instance per instance of the containing class. For example, it's logical to place a comparator comparing objects of a particular class inside that class.

#### **Listing 4.31. Implementing comparator with a nested object**

```
data class Person(val name: String) {
    object NameComparator : Comparator<Person> {
        override fun compare(p1: Person, p2: Person): Int =
            p1.name.compareTo(p2.name)
    }
}

fun main() {
    val persons = listOf(Person("Bob"), Person("Alice"))
    println(persons.sortedWith(Person.NameComparator))
    // [Person(name=Alice), Person(name=Bob)]
}
```

#### **Using Kotlin objects from Java**

An object declaration in Kotlin is compiled as a class with a static field holding its single instance, which is always named `INSTANCE`. If you implemented the Singleton pattern in Java, you'd probably do the same thing by hand. Thus, to use a Kotlin object from the Java code, you access the static `INSTANCE` field:

```
/* Java */
CaseInsensitiveFileComparator.INSTANCE.compare(file1, file2);
Person.NameComparator.INSTANCE.compare(person1, person2)
```

In this example, the `INSTANCE` fields have type `CaseInsensitiveFileComparator` and `NameComparator`, respectively.

Now let's look at a special case of objects nested inside a class: *companion objects*.

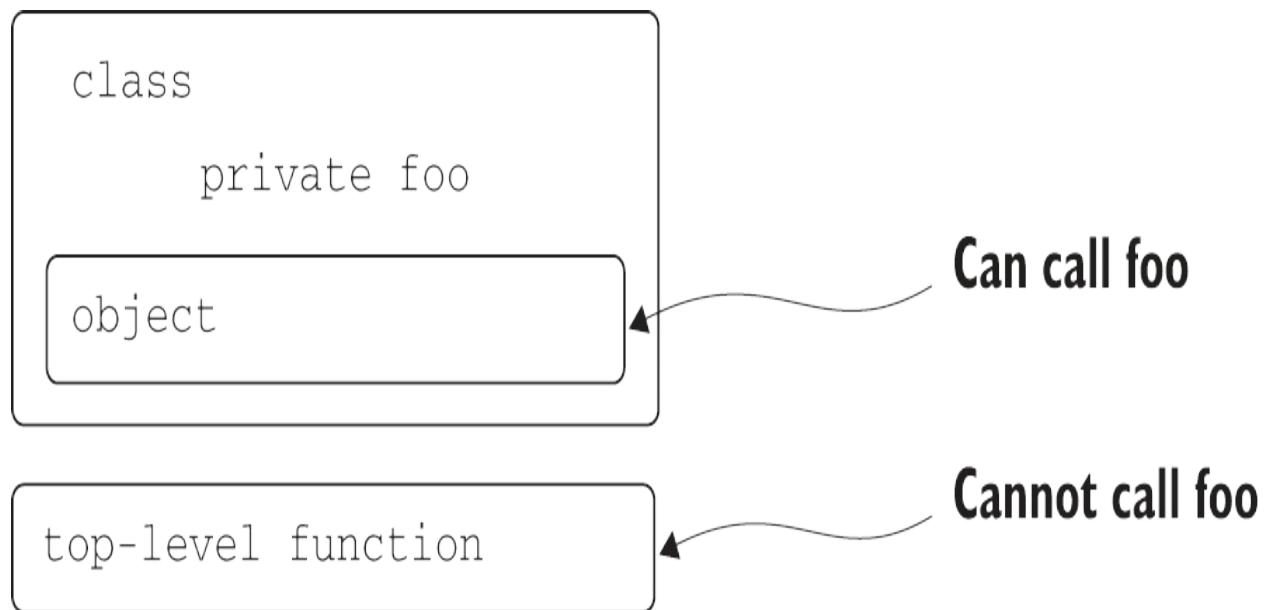
### **4.4.2 Companion objects: a place for factory methods and static members**

Classes in Kotlin can't have static members. In fact, Kotlin doesn't have a `static` keyword like Java does. Instead, Kotlin relies on package-level functions (which can replace static methods in many situations) and object declarations (which serve to replace static methods as well as static fields in other cases). In most cases, it's recommended that you use top-level

functions. But top-level functions can't access private members of a class, as illustrated by [4.5](#). An example of such a function that needs access to private members would be a *factory method*. Factory methods are responsible for the creation of an object, and as such often need access to its private members.

To write a function that can be called without having a class instance but has access to the internals of a class, you can write it as a member of an object declaration inside that class.

**Figure 4.5. Private members can't be used in top-level functions outside of the class.**



Exactly one of the object declarations defined in a class can be marked with a special keyword: `companion`. If you do that, you gain the ability to access the methods and properties of that object directly through the name of the containing class, without specifying the name of the object explicitly. The resulting syntax looks exactly like static method invocation in Java. Here's a basic example showing the syntax:

```
class MyClass {\n    companion object {\n        fun callMe() {\n            println("Companion object called")\n        }\n    }\n}
```

```
}

fun main() {
    MyClass.callMe()
    // Companion object called
}
```

It's important to keep in mind that a companion object belongs to its respective class. You can't access the companion object's members on an instance of the class.

This also sets them apart from static members in Java.

```
fun main() {
    val myObject = MyClass()
    myObject.bar()
    // Error: Unresolved reference: bar
}
```

Remember when we promised you a good place to call a private constructor in [4.2.1](#)? That's the companion object. The companion object has access to all private members of the class, including the private constructor. That makes it an ideal candidate to implement the Factory pattern.

Let's look at an example of declaring two constructors and then change it to use factory methods declared in the companion object. We'll build on [4.19](#), with SocialUser and SubscribingUser. Previously, these entities were different classes implementing the common interface User. Now you decide to manage with only one class, but to provide different means of creating it.

**Listing 4.32. Defining a class with multiple secondary constructors**

```
class User {
    val nickname: String

    constructor(email: String) { #1
        nickname = email.substringBefore('@')
    }

    constructor(socialAccountId: Int) { #1
        nickname = getSocialNetworkName(socialAccountId)
    }
}
```

An alternative approach to express the same logic, which may be beneficial for many reasons, is to use factory methods to create instances of the class. The `User` instance is created through factory methods, not via multiple constructors.

**Listing 4.33. Replacing secondary constructors with factory methods**

```
class User private constructor(val nickname: String) { #1
    companion object { #2
        fun newSubscribingUser(email: String) = #3
            User(email.substringBefore('@'))

        fun newSocialUser(accountId: Int) = #3
            User(getNameFromSocialNetwork(accountId))
    }
}
```

You can invoke the methods of `companion object` via the class name:

```
fun main() {
    val subscribingUser = User.newSubscribingUser("bob@gmail.com")
    val socialUser = User.newSocialUser(4)
    println(subscribingUser.nickname)
    // bob
}
```

Factory methods are very useful. They can be named according to their purpose, as shown in the example. In addition, a factory method can return subclasses of the class where the method is declared, as in the example when `SubscribingUser` and `SocialUser` are classes. You can also avoid creating new objects when it's not necessary. For example, you can ensure that every email corresponds to a unique `User` instance and return an existing instance instead of a new one when the factory method is called with an email that's already in the cache. But if you need to extend such classes, using several constructors may be a better solution, because companion object members can't be overridden in subclasses.

### 4.4.3 Companion objects as regular objects

A companion object is a regular object that is declared in a class. Just like other object declarations, it can be named, implement an interface, or have

extension functions or properties. In this section, we'll look at an example.

Suppose you're working on a web service for a company's payroll, and you need to serialize and deserialize objects as JSON. You can place the serialization logic in a companion object.

**Listing 4.34. Declaring a named companion object**

```
class Person(val name: String) {
    companion object Loader { #1
        fun fromJSON(jsonText: String): Person = /* ... */
    }
}

fun main() {
    val person = Person.Loader.fromJSON("""{"name": "Dmitry"}""")
    println(person.name)
    // Dmitry
    val person2 = Person.fromJSON("""{"name": "Brent"}""") #2
    println(person2.name)
    // Brent
}
```

In most cases, you refer to the companion object through the name of its containing class, so you don't need to worry about its name. But you can specify a name for it if needed, as in [4.34](#): `companion object Loader`.

Regardless of whether you give it a name or not, your class can only have one companion object, and you'll be able to access its members via the class name. If you omit the name of the companion object, the default name assigned to it is `Companion`. You'll see some examples using this name later in [4.3.2](#), when we talk about companion-object extensions.

You can find a number of such singleton companion objects in the Kotlin standard library as well. For example, the `companion object Default` for Kotlin's `Random` class provides access to the default random number generator:

```
val chance = Random.nextInt(from = 0, until = 100) #1
val coin = Random.Default.nextBoolean() #1
```

## Implementing interfaces in companion objects

Just like any other object declaration, a companion object can implement interfaces. As you'll see in a moment, you can use the name of the containing class directly as an instance of an object implementing the interface.

Suppose you have many kinds of objects in your system, including `Person`. You want to provide a common way to create objects of all types. Let's say you have an interface `JSONFactory` for objects that can be deserialized from JSON, and all objects in your system should be created through this factory. You can provide an implementation of that interface for your `Person` class via the companion object. (This example uses generics that you'll get to know in [Chapter 11](#), but is hopefully still clear):

**Listing 4.35. Implementing an interface in a companion object**

```
interface JSONFactory<T> {
    fun fromJSON(jsonText: String): T
}

class Person(val name: String) {
    companion object : JSONFactory<Person> {
        override fun fromJSON(jsonText: String): Person = /* ... */
    }
}
```

Then, if you have a function that uses an abstract factory to load entities, you can pass the `Person` object to it.

```
fun <T> loadFromJSON(factory: JSONFactory<T>): T {
    /* ... */
}

loadFromJSON(Person) #1
```

Note that the name of the `Person` class is used as an instance of `JSONFactory`.

### Kotlin companion objects and static members

The companion object for a class is compiled similarly to a regular object: a static field in a class refers to its instance. If the companion object isn't

named, it can be accessed through the `Companion` reference from the Java code:

```
/* Java */
Person.Companion.fromJSON("...");
```

If a companion object has a name, you use this name instead of `Companion`.

But you may need to work with Java code that requires a member of your class to be static. You can achieve this with the `@JvmStatic` annotation on the corresponding member. If you want to declare a `static` field, use the `@JvmField` annotation on a top-level property or a property declared in an object. These features exist specifically for interoperability purposes and are not, strictly speaking, part of the core language. We'll cover annotations in detail in [Chapter 12](#).

Note that Kotlin can access static methods and fields declared in Java classes, using the same syntax as Java.

## Companion-object extensions

As you saw in [3.3](#), extension functions allow you to define methods that can be called on *instances of a class* defined elsewhere in the codebase. But what if you need to define functions that can be called on *the class itself*, using the same syntax as companion-object methods? If the class has a companion object, you can do so by defining extension functions on it. More specifically, if class `C` has a companion object, and you define an extension function `func` on `C.Companion`, you can call it as `C.func()`.

For example, imagine that you want to have a cleaner separation of concerns for your `Person` class. The class itself will be part of the core business-logic module, but you don't want to couple that module to any specific data format. Because of that, the deserialization function needs to be defined in the module responsible for client/server communication. To keep the same nice syntax for invoking the deserialization function, you can use an extension function on the companion object. Note how you use the default name (`Companion`) to refer to the companion object that was declared without an explicit name:

#### **Listing 4.36. Defining an extension function for a companion object**

```
// business logic module
class Person(val firstName: String, val lastName: String) {
    companion object { #1
    }
}

// client/server communication module
fun Person.Companion.fromJSON(json: String): Person { #2
    /* ... */
}

val p = Person.fromJSON(json)
```

You call `fromJSON` as if it was defined as a method of the companion object, but it's actually defined outside of it as an extension function. As always with extension functions, it looks like a member, but it's not (and just like other extension functions, extending a companion object does not give you access to its private members). But note that you have to declare a companion object in your class, even an empty one, in order to be able to define extensions to it.

You've seen how useful companion objects can be. Now let's move to the next feature in Kotlin that's expressed with the same `object` keyword: object expressions.

#### **4.4.4 Object expressions: anonymous inner classes rephrased**

The `object` keyword can be used not only for declaring named singleton-like objects, but also for declaring *anonymous objects*. Anonymous objects replace Java's use of anonymous inner classes.

For example, let's see how you could provide a typical *event listener* in Kotlin. Let's say you are working with a `Button` class that takes an instance of the `MouseListener` interface to specify the behavior when the user interacts with the button:

```
interface MouseListener {
    fun onEnter()
    fun onClick()
}
```

```
class Button(private val listener: MouseListener) { /* ... */ }
```

You can use an object expression to create an ad-hoc implementation of the `MouseListener` interface, and pass it to the `Button` constructor:

**Listing 4.37. Implementing an event listener with an anonymous object**

```
fun main() {
    Button(object : MouseListener { #1
        override fun onEnter() { /* ... */ } #2
        override fun onClick() { /* ... */ } #2
    })
}
```

The syntax is the same as with object declarations, except that you omit the name of the object. However, unlike object declarations, anonymous objects aren't singletons. Every time an object expression is executed, a new instance of the object is created.

The object expression declares a class and creates an instance of that class, but it doesn't assign a name to the class or the instance. Typically, neither is necessary, because you'll use the object as a parameter in a function call. If you do need to assign a name to the object, you can store it in a variable:

```
val listener = object : MouseListener {
    override fun onEnter() { /* ... */ }
    override fun onClick() { /* ... */ }
}
```

Anonymous objects in Kotlin are quite flexible: they can implement one interface, multiple interfaces, or no interfaces at all.

Code in an object expression can access the variables in the function where it was created—just like in Java's anonymous classes. But unlike in Java, this isn't restricted to `final` variables. In Kotlin, you can also modify the values of variables from within an object expression. For example, let's see how you can use the `listener` to count the number of clicks in a window.

**Listing 4.38. Accessing local variables from an anonymous object**

```
fun main() {
    var clickCount = 0 #1
    Button(object : MouseListener {
        override fun onEnter() { /* ... */ }
        override fun onClick() {
            clickCount++ #2
        }
    })
}
```



#### Note

Object expressions are mostly useful when you need to override multiple methods in your anonymous object. If you only need to implement a single-method interface (such as `Runnable`), you can rely on Kotlin's support for SAM conversion (converting a function literal to an implementation of an interface with a single abstract method) and write your implementation as a function literal (lambda). We'll discuss lambdas and SAM conversion in much more detail in [single\\_abstract\\_method5.2](#).

## 4.5 Extra type safety without overhead: Inline classes

With data classes, you've already seen how code generated by the compiler can help you keep your code readable and clutter-free. Let's look at another example that shows off the power of the Kotlin compiler: inline classes.

Assume for a moment you're building a simple system to keep track of your expenses:

```
fun addExpense(expense: Int) {
    // save the expense as USD cent
}
```

During your next trip to Japan, you want to add the invoice for a tasty Nikuman (a steamed bun) which costs you 200¥:

```
addExpense(200) // Japanese Yen
```

The problem becomes apparent quite quickly: Because the signature of your function accepts a plain `Int`, there is nothing preventing callers of the function from passing values that actually have different semantics. In this specific case, there is nothing preventing the caller from interpreting the parameter as "yen", even though the actual implementation requires the value to be passed to mean "USD cent".

The classical approach to preventing this is to use a class instead of a plain `Int`:

```
class UsdCent(val amount: Int)

fun addExpense(expense: UsdCent) {
    // save the expense as USD cent
}

fun main() {
    addExpense(UsdCent(147))
}
```

While this approach makes it far less likely to accidentally pass a value with the wrong semantics to the function, it comes with performance considerations: a new `UsdCent` object needs to be created for each `addExpense` function call, which is then unwrapped inside the function body and discarded. If this function is called a lot, it means a large number of short-lived objects need to be allocated and subsequently garbage-collected.

This is where *inline classes* come into play. They allow you to introduce a layer of type-safety without compromising performance.

To turn the `UsdCent` class into an inline class, mark it with the `value` keyword, and annotate it with `@JvmInline`:

```
@JvmInline
value class UsdCent(val amount: Int)
```

This small change avoids the needless instantiation of objects without giving up on the type-safety provided by your `UsdCent` wrapper type. At runtime, instances of `UsdCent` will be represented as the wrapped property. This is also where inline classes get their name from: the data of the class is *inlined* at the

usage sites.



### Note

To be entirely accurate, the Kotlin compiler represents the inline class as its underlying type *wherever possible*. There are cases in which it is necessary to keep the wrapper type—most notably, when the inline class is used as a type parameter. You can find a discussion of these special cases in the Kotlin documentation<sup>[4]</sup>. We'll take a closer look at the idea of wrapping and unwrapping for types in [7](#).

To qualify as "inline", your class must have exactly one property, which needs to be initialized in the primary constructor. Inline classes also don't participate in class hierarchies: They don't extend other classes, and can't be extended themselves.

However, they can still implement interfaces, define methods, or provide computed properties (which you learned about in [2.2.2](#)):

```
interface PrettyPrintable {
    fun prettyPrint()
}

@JvmInline
value class UsdCent(val amount: Int): PrettyPrintable {
    val salesTax get() = amount * 0.06
    override fun prettyPrint() = println("${amount}¢")
}

fun main() {
    val expense = UsdCent(1_99)
    println(expense.salesTax)
    // 11.94
    expense.prettyPrint()
    // 199¢
}
```

You'll mostly find yourself reaching for inline classes to make the semantics of basic values explicit, such as indicating units of measurement used for plain number types or differentiating the meaning of different strings. They

prevent function callers from accidentally passing compatible values with differing semantics. We'll see another prime example of inline classes in [8.1.2](#).

### Inline classes and Project Valhalla

Currently, inline classes are a feature of the Kotlin compiler. It knows how to emit code that doesn't incur the performance penalty of allocating objects in most cases.

But that won't have to be the case forever: Project Valhalla<sup>[5]</sup> is a set of JDK enhancement proposals (JEPs) that aim to bring inline classes (now referred to as "primitive classes", previously also "value classes") support into the JVM itself. This means the runtime environment would natively understand the concept of inline classes.

Thus, Valhalla is also the reason why inline classes in Kotlin are currently annotated with `@JvmInline`: It makes it explicit that inline classes currently get special treatment from the Kotlin compiler. When a Valhalla-based implementation is available in the future, you'll be able to declare your inline classes in Kotlin without the annotation, and make use of the builtin JVM support by default.

We've finished our discussion of classes, interfaces, and objects. In the next chapter, we'll move on to one of the most interesting areas of Kotlin: lambdas and functional programming.

[4] <https://kotlinlang.org/docs/inline-classes.html>

[5] <https://openjdk.org/projects/valhalla/>

## 4.6 Summary

- Interfaces in Kotlin are similar to Java's but can contain default implementations and properties.
- All declarations are `final` and `public` by default.
- To make a declaration non-`final`, mark it as open.

- internal declarations are visible in the same module.
- Nested classes aren't inner by default. Use the keyword `inner` to store a reference to the outer class.
- All direct subclasses of sealed classes and all implementations of sealed interfaces need to known at compile time.
- Initializer blocks and secondary constructors provide flexibility for initializing class instances.
- You use the `field` identifier to reference a property backing field from the accessor body.
- Data classes provide compiler-generated `equals`, `hashCode`, `toString`, `copy`, and other methods.
- Class delegation helps to avoid many similar delegating methods in your code.
- Object declaration is Kotlin's way to define a singleton class.
- Companion objects (along with package-level functions and properties) replace Java's static method and field definitions.
- Companion objects, like other objects, can implement interfaces, as well as have extension functions and properties.
- Object expressions are Kotlin's replacement for Java's anonymous inner classes, with added power such as the ability to implement multiple interfaces and to modify the variables defined in the scope where the object is created.
- Inline classes allow you to introduce a layer of type-safety to your program while avoiding potential performance hits caused by allocating many short-lived objects.

# 5 Programming with lambdas

## This chapter covers

- Using lambda expressions and member references to pass snippets of code and behavior to functions
- Defining functional interfaces in Kotlin and using Java functional interfaces
- Using lambdas with receivers

*Lambda expressions*, or simply *lambdas*, are essentially small chunks of code that can be passed to other functions. With lambdas, you can easily extract common code structures into library functions, allowing you to reuse more code and be more expressive while you’re at it. The Kotlin standard library makes heavy use of them. In this chapter, you’ll learn what a lambda is, see examples of some typical use cases for lambda functions, what they look like in Kotlin, and their relationship to *member references*.

You’ll also see how lambdas are fully interoperable with Java APIs and libraries—even those that weren’t originally designed with lambdas in mind—and how you can use functional interfaces in Kotlin to make code dealing with function types even more expressive. Finally, we’ll look at *lambdas with receivers*—a special kind of lambdas where the body is executed in a different context than the surrounding code.

## 5.1 Lambda expressions and member references

The introduction of lambdas to Java 8 was one of the longest-awaited changes in the evolution of the language. Why was it such a big deal? In this section, you’ll find out why lambdas are so useful and what the syntax of lambda expressions in Kotlin looks like.

### 5.1.1 Introduction to lambdas: blocks of code as values

Passing and storing pieces of behavior in your code is a frequent task. For example, you often need to express ideas like "When an event happens, run this handler" or "Apply this operation to all elements in a data structure." In older versions of Java, you could accomplish this through anonymous inner classes. While anonymous inner classes get the job done, they require verbose syntax.

There is another approach to solve this problem: the ability to *treat functions as values*. Instead of declaring a class and passing an instance of that class to a function, you can pass a function directly. With lambda expressions, the code is even more concise. You don't need to declare a function: instead, you can, effectively, pass a block of code directly as a function parameter. This approach of treating functions as values and combining functions to express behavior is also one of the main pillars of *functional programming*.

### A quick refresher on functional programming

In [1.2.3](#), we already briefly talked about Kotlin's nature as a multi-paradigm language, and the benefits that functional programming can bring to your projects: succinctness, a focus on immutability, and an even stronger power of abstraction. To refresh your memory, here are some of the hallmarks of functional programming again:

- *First-class functions*— Functions (pieces of behavior) are treated as values. You can store them in variables, pass them as parameters, or return them from other functions. Lambdas, as you will explore them in this chapter, is one of Kotlin's language features that enables comfortably treating functions as first-class.
- *Immutability*— You design your objects in a way that guarantees their internal state can't change after their creation: They cannot mutate.
- *No side effects*— You structure your functions to return the same result given the same inputs without modifying the state of other objects or the outside world. Such functions are called *pure*.

Let's look at an example to illustrate where the approach of using lambda expressions really shines. Imagine that you need to define the behavior for clicking a button. To do so, a `Button` object may require you to pass an instance of the corresponding `OnClickListener` interface that is responsible

for handling the click. This interface specifies one method, `onClick`. In Kotlin, you could implement it by using an object declaration, as introduced in [4.4.1](#):

**Listing 5.1. Implementing a listener with an object declaration.**

```
button.setOnClickListener(object: OnClickListener {  
    override fun onClick(v: View) {  
        println("I was clicked!")  
    }  
})
```

The verbosity required to declare an object like this becomes irritating when repeated many times. A notation to express just the *behavior*—what should be done on clicking—helps you eliminate redundant code: You can rewrite the snippet above using a lambda.

**Listing 5.2. Implementing a listener with a lambda.**

```
button.setOnClickListener {  
    println("I was clicked!")  
}
```

This Kotlin code does the same thing as using an anonymous object, but is more concise and readable. We'll discuss the details of this example (and why it can be used to implement an interface like `OnClickListener`) later in this section.

You saw how a lambda can be used as an alternative to an anonymous object with only one method. Let's now continue and briefly explore another classical use of lambda expressions: working with collections.

## 5.1.2 Lambdas and collections

One of the main tenets of good programming style is to avoid duplication in your code. Most of the tasks we perform with collections follow a few common patterns. Lambdas enable Kotlin to provide a good, convenient standard library that provides powerful functionality to work with collections.

Let's look at an example. You'll use the `Person` class that contains information about a person's name and age.

```
data class Person(val name: String, val age: Int)
```

Suppose you have a list of people, and you need to find the oldest of them. If you had no experience with lambdas, you might rush to implement the search manually. You'd introduce two intermediate variables—one to hold the maximum age and another to store the first found person of this age—and then iterate over the list, updating these variables.

**Listing 5.3. Searching through a collection manually via a `for`-loop**

```
fun findTheOldest(people: List<Person>) {
    var maxAge = 0 #1
    var theOldest: Person? = null #2
    for (person in people) {
        if (person.age > maxAge) { #3
            maxAge = person.age
            theOldest = person
        }
    }
    println(theOldest)
}

fun main() {
    val people = listOf(Person("Alice", 29), Person("Bob", 31))
    findTheOldest(people)
    // Person(name=Bob, age=31)
}
```

With enough experience, you can bang out such loops pretty quickly. But there's quite a lot of code here, and it's easy to make mistakes. For example, you might get the comparison wrong and find the minimum element instead of the maximum.

In Kotlin, there's a better way. You can use a function from the standard library, as shown next.

**Listing 5.4. Searching through a collection by using the `maxByOrNull` function from the standard library with a lambda**

```
fun main() {  
    val people = listOf(Person("Alice", 29), Person("Bob", 31))  
    println(people.maxByOrNull { it.age }) #1  
    // Person(name=Bob, age=31)  
}
```

The `maxByOrNull` function can be called on any collection and takes one argument: the function that specifies what values should be compared to find the maximum element. The code in curly braces `{ it.age }` is a lambda implementing this "selector logic": It receives a collection element as an argument and returns a value to compare. Because the lambda only takes one argument (the collection item) and we don't specify an explicit name for it, we refer to it using the implicit name `it`. In this example, the collection element is a `Person` object, and the value to compare is its age, stored in the `age` property.

If a lambda just delegates to a function or property, it can be replaced by a member reference.

#### **Listing 5.5. Searching using a member reference**

```
people.maxByOrNull(Person::age)
```

This code means the same thing as [5.4](#). [5.1.5](#) will cover the details.

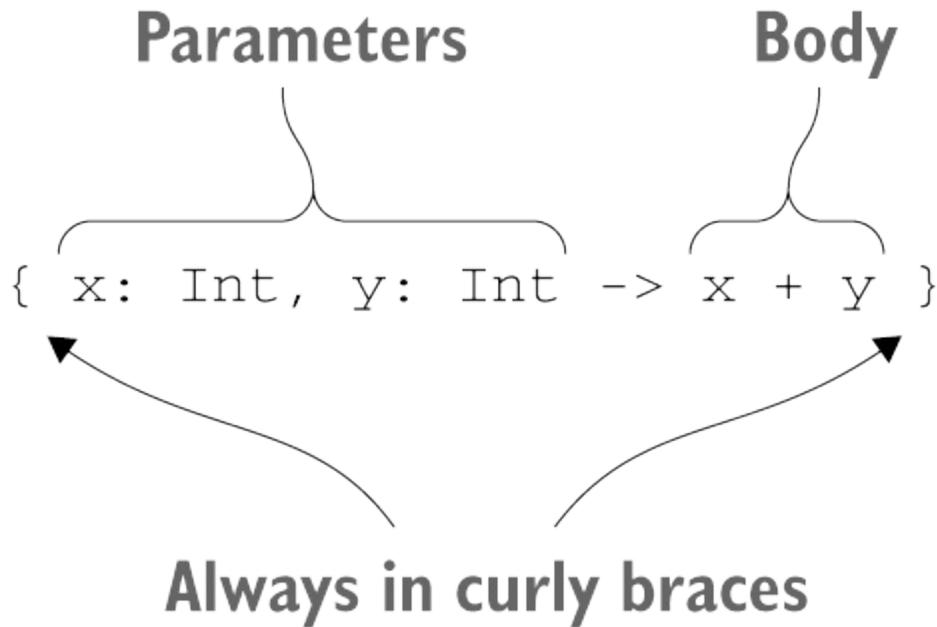
Most of the things we typically do with collections can be concisely expressed with library functions taking lambdas or member references. The resulting code is much shorter and easier to understand, and often communicates your intent (that is, what your code is trying to achieve) more clearly than its loop-based counterpart. To help you start getting used to it, let's look at the syntax for lambda expressions. Later, in [6](#), we will take a more detailed look at what functionality is available to you out for manipulating collections using lambdas.

### **5.1.3 Syntax for lambda expressions**

As we've mentioned, a lambda encodes a small piece of behavior that you can pass around as a value. It can be declared independently and stored in a variable. But more frequently, it's declared directly when passed to a

function. [5.1](#) shows the syntax for declaring lambda expressions.

**Figure 5.1. Lambda expression syntax:** A lambda is always surrounded by curly braces, specifies a number of parameters, and provides a body of the lambda which contains the actual logic.



A lambda expression in Kotlin is always surrounded by curly braces. Note that there are no parentheses around the arguments. The arrow separates the argument list from the body of the lambda.

You can store a lambda expression in a variable and then treat this variable like a normal function (call it with the corresponding arguments):

```
fun main() {  
    val sum = { x: Int, y: Int -> x + y }  
    println(sum(1, 2)) #1  
    // 3  
}
```

If you want to, you can call the lambda expression directly:

```
fun main() {  
    { println(42) }()  
    // 42  
}
```

But such syntax isn't readable and doesn't make much sense (it's equivalent to executing the lambda body directly). If you need to enclose a piece of code in a block, you can use the library function `run` that executes the lambda passed to it:

```
fun main() {
    run { println(42) } #1
    // 42
}
```

The `run` function becomes especially useful when you need to execute a block of several statements in a place where an expression is expected. Consider a declaration of a top-level variable that performs some setup or does some additional work:

```
val myFavoriteNumber = run {
    println("I'm thinking!")
    println("I'm doing some more work...")
    42
}
```

In [Chapter 10](#), you'll learn why such invocations—unlike the creation of a lambda expression and calling it directly via `{...}()`—have no runtime overhead and are as efficient as built-in language constructs. Let's return to [5.4](#), which finds the oldest person in a list:

```
fun main() {
    val people = listOf(Person("Alice", 29), Person("Bob", 31))
    println(people.maxByOrNull { it.age })
    // Person(name=Bob, age=31)
}
```

If you rewrite this example without using any syntax shortcuts, you get the following:

```
people.maxByOrNull({ p: Person -> p.age })
```

It should be clear what happens here: the piece of code in curly braces is a lambda expression, and you pass it as an argument to the function. The lambda expression takes one argument of type `Person` and returns its age.

But this code is verbose. First, there's too much punctuation, which hurts

readability. Second, the type can be inferred from the context and therefore omitted. Last, you don't need to assign a name to the lambda argument in this case.

Let's make these improvements, starting with the braces. In Kotlin, a syntactic convention lets you move a lambda expression out of parentheses if it's the last argument in a function call. In this example, the lambda is the only argument, so it can be placed after the parentheses:

```
people.maxByOrNull() { p: Person -> p.age }
```

When the lambda is the only argument to a function, you can also remove the empty parentheses from the call:

```
people.maxByOrNull { p: Person -> p.age }
```

All three syntactic forms mean the same thing, but the last one is the easiest to read. If a lambda is the only argument, you'll definitely want to write it without the parentheses. If a function takes several arguments, and only the last argument is a lambda, it's also considered good style in Kotlin to keep the lambda outside the parentheses. If you want to pass two or more lambdas, you can't move more than one out, so it's usually better to keep all of them inside the parentheses.

**Figure 5.2.** You simplified your `maxByOrNull` call to get the oldest person from the `people` collection in six steps. You first moved the lambda out of the parentheses (1), removed the now empty pair of parentheses (2), used the Kotlin compiler's type inference instead of specifying the parameter type for `p` explicitly (3), and used the implicit name for the only lambda parameter `it` (5). You also learned an additional shorthand in the form of member references (6).

(1) people.maxByOrNull(( { p: Person -> p.age } ))

(2) people.maxByOrNull(() { p: Person -> p.age })

(3) people.maxByOrNull { p: Person -> p.age }

(4) people.maxByOrNull { p -> p.age }

(5) people.maxByOrNull { it.age }

(6) people.maxByOrNull(Person::age)

To see what these options look like with a more complex call, let's go back to the `joinToString` function that you used extensively in [3](#). It's also defined in the Kotlin standard library, with the difference that the standard library version takes a function as an additional parameter. This function can be used to convert an element to a string differently than its `toString` function. Here's how you can use it to print names only.

**Listing 5.6. Passing a lambda as a named argument**

```
fun main() {  
    val people = listOf(Person("Alice", 29), Person("Bob", 31))  
    val names = people.joinToString(  
        separator = " ",  
        transform = { p: Person -> p.name }  
    )  
    println(names)  
    // Alice Bob  
}
```

And here's how you can rewrite that call with the lambda outside the parentheses.

**Listing 5.7. Passing a lambda outside of parentheses**

```
people.joinToString(" ") { p: Person -> p.name }
```

[5.6](#) uses a named argument to pass the lambda, making it clear what the lambda is used for. [5.7](#) is more concise, but it doesn't express explicitly what the lambda is used for, so it may be harder to understand for people not familiar with the function being called.

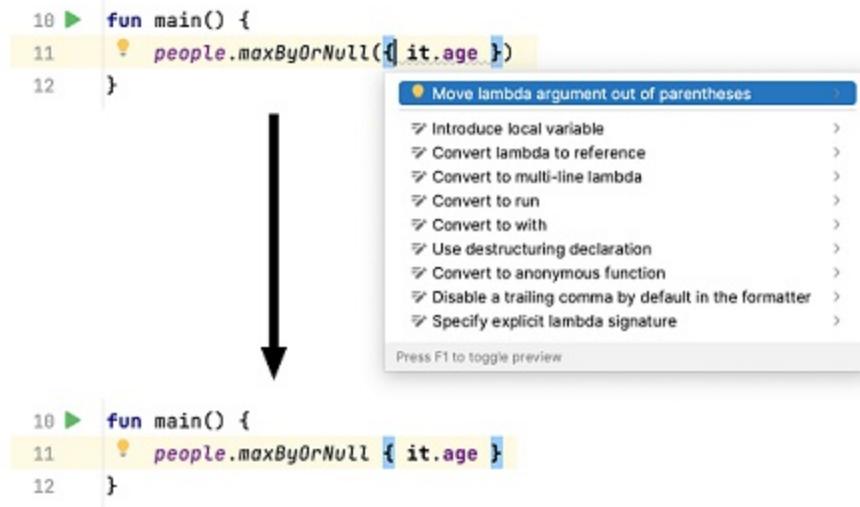


**Moving lambda arguments in IntelliJ IDEA and Android Studio**

IntelliJ IDEA and Android Studio allow you to automatically convert between the two syntactic forms of where to place a last-argument lambda. To convert one form to the other, place your cursor on the lambda. Press Alt + Enter (or Option + Return on macOS) or click the floating yellow lightbulb icon, and select "Move lambda argument out of parentheses" and "Move

lambda argument into parentheses."

Figure 5.3. The "move lambda" intention actions allow you to



Let's move on with simplifying the syntax and get rid of the parameter type.

Listing 5.8. Omitting lambda parameter type: Often, the Kotlin compiler can infer the type of the variables accepted by the lambda.

```
people.maxByOrNull { p: Person -> p.age } #1  
people.maxByOrNull { p -> p.age } #2
```

As with local variables, if the type of a lambda parameter can be inferred, you don't need to specify it explicitly. With the `maxByOrNull` function, the parameter type is always the same as the collection element type. The compiler knows you're calling `maxByOrNull` on a collection of `Person` objects, so it can understand that the lambda parameter will also be of type `Person`.

There are cases when the compiler can't infer the lambda parameter type, but we won't discuss them here. The simple rule you can follow is to always start without the types; if the compiler complains, specify them.

You can specify only some of the argument types while leaving others with just names. Doing so may be convenient if the compiler can't infer one of the

types or if an explicit type improves readability.

The last simplification you can make in this example is to replace a parameter with the default parameter name: `it`. This name is generated if the context expects a lambda with only one argument, and its type can be inferred.

**Listing 5.9. Using the default parameter name `it`**

```
people.maxByOrNull { it.age } #1
```

This default name is generated only if you don't specify the argument name explicitly.



**Note**

The `it` convention is great for shortening your code, but you shouldn't abuse it. In particular, in the case of nested lambdas, it's better to declare the parameter of each lambda explicitly; otherwise it's difficult to understand which value the `it` refers to (and you will get a warning along the lines of "Implicit parameter `it` of enclosing lambda is shadowed"). It's useful also to declare parameters explicitly if the meaning or the type of the parameter isn't clear from the context.

If you store a lambda in a variable, there's no context from which to infer the parameter types, so you have to specify them explicitly:

```
val getAge = { p: Person -> p.age }
people.maxByOrNull(getAge)
```

So far, you've only seen examples with lambdas that consist of one expression or statement. But lambdas aren't constrained to such a small size and can contain multiple statements. In this case, the last expression is the result—no explicit `return` statement needed:

```
fun main() {
    val sum = { x: Int, y: Int ->
        println("Computing the sum of $x and $y...")
        x + y
    }
}
```

```
    println(sum(1, 2))
    // Computing the sum of 1 and 2...
    // 3
}
```

Next, let's talk about a concept that often goes side-by-side with lambda expressions: capturing variables from the context.

### 5.1.4 Accessing variables in scope

You know that when you declare an anonymous inner class in a function, you can refer to parameters and local variables of that function from inside the class. With lambdas, you can do exactly the same thing. If you use a lambda in a function, you can access the parameters of that function as well as the local variables declared before the lambda.

To demonstrate this, let's use the `forEach` standard library function. It's one of the most basic collection-manipulation functions; all it does is call the given lambda on every element in the collection. The `forEach` function is somewhat more concise than a regular `for` loop, but it doesn't have many other advantages, so you don't need to rush to convert all your loops to lambdas.

The following listing takes a list of messages and prints each message with the same prefix.

**Listing 5.10. Using function parameters in a lambda:** A lambda defined inside a function can access the parameters of that function, and all local variables that were declared before that lambda.

```
fun printMessagesWithPrefix(messages: Collection<String>, prefix:
    messages.forEach { #1
        println("$prefix $it") #2
    }
}

fun main() {
    val errors = listOf("403 Forbidden", "404 Not Found")
    printMessagesWithPrefix(errors, "Error:")
    // Error: 403 Forbidden
    // Error: 404 Not Found
```

}

**Figure 5.4.** The `foreach` lambda can access the `prefix` variable defined in the surrounding scope, and any other variables defined in surrounding scopes—all the way up to the surrounding class and file scopes.

Class / file scope

```
fun printMessagesWithPrefix(messages: ..., prefix: ...) {  
    messages.forEach {  
        println("$prefix $it")  
    }  
}
```

The diagram illustrates the scope of variables in a piece of pseudocode. The code consists of a function definition with parameters and a body containing a `forEach` loop and a `println` statement.

- Class / file scope:** The entire code block is considered to have this scope.
- Lambda scope:** The variables `messages` and `prefix` are considered to have this scope, as they are parameters to the function.
- Function scope:** The variable `it` is considered to have this scope, as it is a parameter to the `forEach` block.

A vertical arrow points from the `messages` parameter in the parameter list up to the `messages` variable in the `forEach` loop, indicating they are the same variable. Another vertical arrow points from the `it` variable in the `forEach` loop down to the `it` variable in the `println` statement, indicating they are the same variable.

One important difference between Kotlin and Java is that in Kotlin, you aren't restricted to accessing final variables: You can also modify variables from within a lambda. In the next listing, you're counting the number of client and server errors in a given collection of response status codes. You do so by incrementing the `clientErrors` and `serverErrors` variables defined in the `printProblemCounts` functions from within the `forEach` lambda:

**Listing 5.11. Changing local variables from a lambda**

```
fun printProblemCounts(responses: Collection<String>) {  
    var clientErrors = 0 #1  
    var serverErrors = 0 #1  
    responses.forEach {  
        if (it.startsWith("4")) {  
            clientErrors++ #2  
        } else if (it.startsWith("5")) {  
            serverErrors++ #2  
        }  
    }  
    println("$clientErrors client errors, $serverErrors server errors")  
}  
  
fun main() {  
    val responses = listOf("200 OK", "418 I'm a teapot",  
                          "500 Internal Server Error")  
    printProblemCounts(responses)  
    // 1 client errors, 1 server errors  
}
```

Kotlin, unlike Java, allows you to access non-final variables and even modify them in a lambda. External variables accessed from a lambda, such as `prefix`, `clientErrors`, and `serverErrors` in these examples, are said to be *captured* by the lambda.

Note that, by default, the lifetime of a local variable is constrained by the function in which the variable is declared. But if it's captured by the lambda, the code that uses this variable can be stored and executed later. You may ask how this works. When you capture a final variable, its value is stored together with the lambda code that uses it. For non-final variables, the value is enclosed in a special wrapper that lets you change it, and the reference to the wrapper is stored together with the lambda.

## Capturing a mutable variable: implementation details

Java allows you to capture only final variables. When you want to capture a mutable variable, you can use one of the following tricks: either declare an array of one element in which to store the mutable value, or create an instance of a wrapper class that stores the reference that can be changed. If you used this technique explicitly in Kotlin, the code would be as follows:

```
class Ref<T>(var value: T) #1

fun main() {
    val counter = Ref(0)
    val inc = { counter.value++ } #2
}
```

In real code, you don't need to create such wrappers. Instead, you can mutate the variable directly:

```
fun main() {
    var counter = 0
    val inc = { counter++ }
}
```

How does it work? The first example shows how the second example works under the hood. Any time you capture a final variable (val), its value is copied, as in Java. When you capture a mutable variable (var), its value is stored as an instance of a `Ref` class. The `Ref` variable is final and can be easily captured, whereas the actual value is stored in a field and can be changed from the lambda.

An important caveat is that, if a lambda is used as an event handler or is otherwise executed asynchronously, the modifications to local variables will occur only when the lambda is executed. For example, the following code isn't a correct way to count button clicks:

```
fun tryToCountButtonClicks(button: Button): Int {
    var clicks = 0
    button.onClick { clicks++ }
    return clicks
}
```

This function will always return 0. Even though the `onclick` handler will modify the value of `clicks`, you won't be able to observe the modification, because the `onclick` handler will be called after the function returns. A correct implementation of the function would need to store the click count not in a local variable, but in a location that remains accessible outside the function—for example, in a property of a class.

We've discussed the syntax for declaring lambdas and how variables are captured in lambdas. Now let's talk about member references, a feature that lets you easily pass references to existing functions.

### 5.1.5 Member references

You've seen how lambdas allow you to pass a block of code as a parameter to a function. But what if the code that you need to pass as a parameter is already defined as a function? Of course, you can pass a lambda that calls that function, but doing so is somewhat redundant. Can you pass the function directly?

In Kotlin, just like in Java 8, you can do so if you convert the function to a value. You use the `::` operator for that:

```
val getAge = Person::age
```

This expression is called *member reference*, and it provides a short syntax for creating a function value that calls exactly one method or accesses a property. A double colon separates the name of a class from the name of the member you need to reference (a method or property), as shown in [5.5](#).

**Figure 5.5. Member reference syntax**

## Class Member



## Separated by double colon

This is a more concise expression of a lambda that does the same thing:

```
val getAge = { person: Person -> person.age }
```

Note that, regardless of whether you're referencing a function or a property, you shouldn't put parentheses after its name in a member reference. After all, you're not invoking it, but working with a reference to it.

A member reference has the same type as a lambda that calls that function, so you can use the two interchangeably:

```
people.maxByOrNull(Person::age) #1  
people.maxByOrNull { person: Person -> person.age } #1
```

You can have a reference to a function that's declared at the top level (and isn't a member of a class), as well:

```
fun salute() = println("Salute!")  
  
fun main() {  
    run(::salute) #1  
    // Salute!  
}
```

In this case, you omit the class name and start with ::. The member reference ::salute is passed as an argument to the library function run, which calls the corresponding function.

When a lambda delegates to a function which takes several parameters, it's especially convenient to provide a member reference—it lets you avoid repeating the parameter names and their types:

```
val action = { person: Person, message: String -> #1
    sendEmail(person, message)
}
val nextAction = ::sendEmail #2
```

You can store or postpone the action of creating an instance of a class using a *constructor reference*. The constructor reference is formed by specifying the class name after the double colons:

```
data class Person(val name: String, val age: Int)

fun main() {
    val createPerson = ::Person #1
    val p = createPerson("Alice", 29)
    println(p)
    // Person(name=Alice, age=29)
}
```

Note that you can also reference extension functions in the same way:

```
fun Person.isAdult() = age >= 21
val predicate = Person::isAdult
```

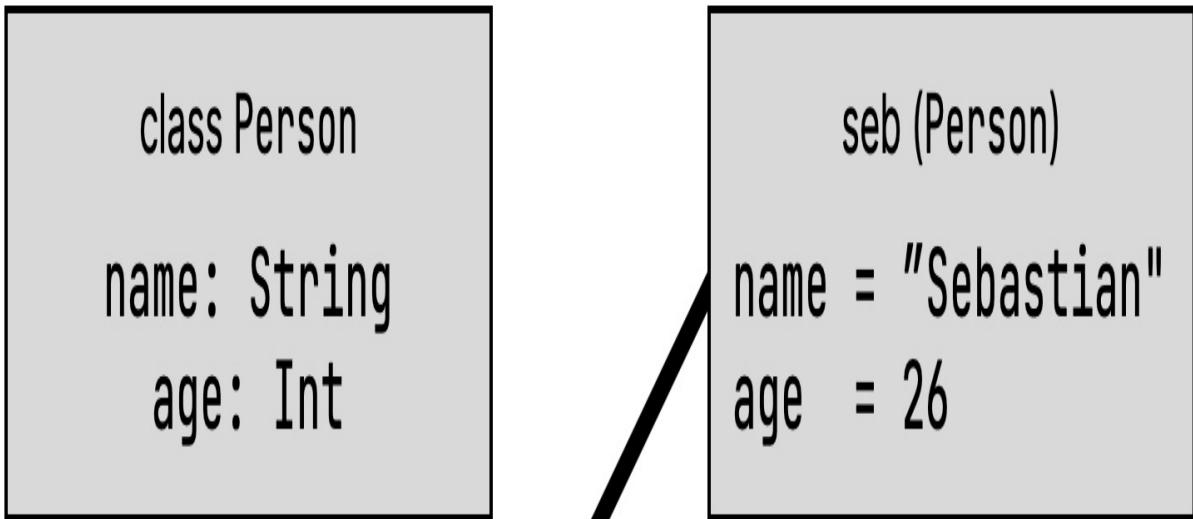
Although `isAdult` isn't a member of the `Person` class, you can access it via reference, just as you can access it as a member on an instance:  
`person.isAdult()`.

## 5.1.6 Bound callable references

```
fun main() {
    val seb = Person("Sebastian", 26)
    val personsAgeFunction = Person::age #1
    println(personsAgeFunction(seb)) #2
    // 26
    val sebsAgeFunction = seb::age #3
    println(sebsAgeFunction()) #4
    // 26
}
```

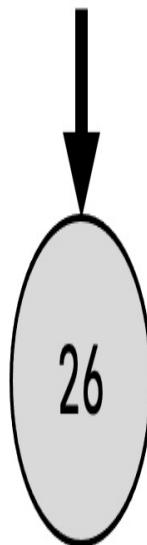
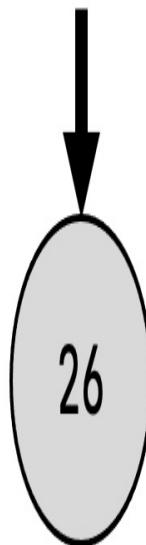
As you can see, the way we defined `sebsAgeFunction` in this example—`seb::age`—is equivalent to writing the lambda `{ seb.age }` explicitly, but more concise.

**Figure 5.6. A regular member reference, like `Person::age`, takes an instance of an object as a parameter and returns the value of the member. A bound member reference like `seb::age` takes no arguments, and returns the value of the member belonging to the object it is bound to.**



Person::age(seb)

seb::age()



In the following section, we'll look at many library functions that work great with lambda expressions, as well as member references.

We've thoroughly discussed a frequently used application of lambda expressions: using them to simplify manipulating collections. Now let's continue with another important topic: using lambdas with an existing Java API.

## 5.2 Using Java functional interfaces: Single Abstract Methods (SAM)

There are already a lot of libraries in the JVM ecosystem written in Kotlin, and those libraries can directly make use of Kotlin's lambdas. However, there is a good chance you may want to use a library written in Java in your Kotlin project. The good news is that Kotlin lambdas are fully interoperable with Java APIs. In this section, you'll see exactly how this works.

At the beginning of the chapter, you saw an example of passing a lambda to a Java method:

```
button.setOnClickListener { #1
    println("I was clicked!")
}
```

The `Button` class sets a new listener to a button via an `setOnClickListener` method that takes an argument of type `OnClickListener`:

```
/* Java */
public class Button {
    public void setOnClickListener(OnClickListener l) { ... }
}
```

The `OnClickListener` interface declares one method, `onClick`:

```
/* Java */
public interface OnClickListener {
    void onClick(View v);
}
```

Depending on the Java version, implementing the `OnClickListener` interface can be quite involved: Prior to Java 8, you had to create a new instance of an anonymous class to pass it as an argument to the `setOnItemClickListener` method:

```
/* Before Java 8 */
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        ...
    }
}

/* Only since Java 8 */
button.setOnClickListener(view -> { ... });
```

In Kotlin, you simply pass a lambda:

```
button.setOnClickListener { view -> ... }
```

The lambda used to implement `OnClickListener` has one parameter of type `View`, as in the `onClick` method. The mapping is illustrated in [5.7](#).

**Figure 5.7. Parameters of the lambda correspond to method parameters.**

```
public interface OnClickListener {
    void onClick(View v);           → { view -> ... }
}
```

This works because the `OnClickListener` interface has only one abstract method. Such interfaces are called *functional interfaces*, or *SAM interfaces*, where SAM stands for *single abstract method*. The Java API is full of functional interfaces like `Runnable` and `Callable`, as well as methods working with them. Kotlin allows you to use lambdas when calling Java methods that take functional interfaces as parameters, ensuring that your Kotlin code remains clean and idiomatic.

Let's look in detail at what happens when you pass a lambda to a method that expects an argument of a functional interface type.

### 5.2.1 Passing a lambda as a parameter to a Java method

You can pass a lambda to any Java method that expects a functional interface. For example, consider this method, which has a parameter of type `Runnable`:

```
/* Java */
void postponeComputation(int delay, Runnable computation);
```

In Kotlin, you can invoke it and pass a lambda as an argument. The compiler will automatically convert it into an instance of `Runnable`:

```
postponeComputation(1000) { println(42) }
```

Note that when we say "an instance of `Runnable`," what we mean is "an instance of an anonymous class implementing `Runnable`." The compiler will create that for you and will use the lambda as the body of the single abstract method—the `run` method, in this case.

You can achieve the same effect by creating an anonymous object that implements `Runnable` explicitly:

```
postponeComputation(1000, object : Runnable { #1
    override fun run() {
        println(42)
    }
})
```

But there's a difference. When you explicitly declare an object, a new instance is created on each invocation. With a lambda, the situation is different: if the lambda doesn't access any variables from the function where it's defined, the corresponding anonymous class instance is reused between calls:

```
postponeComputation(1000) { println(42) } #1
```

If the lambda captures variables from the surrounding scope, it's no longer

possible to reuse the same instance for every invocation. In that case, the compiler creates a new object for every call and stores the values of the captured variables in that object. For example, in the following function, every invocation uses a new `Runnable` instance, storing the `id` value as a field:

```
fun handleComputation(id: String) {
    postponeComputation(1000) { #1
        println(id) #2
    }
}
```

Note that the discussion of creating an anonymous class and an instance of this class for a lambda is valid for Java methods expecting functional interfaces, but does not apply to working with collections using Kotlin extension methods. If you pass a lambda to a Kotlin function that's marked `inline`, no anonymous classes are created. And most of the library functions are marked `inline`. Details of how this works are discussed in [Chapter 10](#).

As you've seen, in most cases the conversion of a lambda to an instance of a functional interface happens automatically, without any effort on your part. But there are cases when you need to perform the conversion explicitly. Let's see how to do that.

### 5.2.2 SAM constructors: explicit conversion of lambdas to functional interfaces

A *SAM constructor* is a compiler-generated function that lets you perform an explicit conversion of a lambda into an instance of an interface with a single abstract method. You can use it in contexts when the compiler doesn't apply the conversion automatically. For instance, if you have a method that returns an instance of a functional interface, you can't return a lambda directly; you need to wrap it into a SAM constructor. Here's a simple example.

**Listing 5.12. Using a SAM constructor to return a value**

```
fun createAllDoneRunnable(): Runnable {
    return Runnable { println("All done!") }
}
```

```
fun main() {
    createAllDoneRunnable().run()
    // All done!
}
```

The name of the SAM constructor is the same as the name of the underlying functional interface. The SAM constructor takes a single argument—a lambda that will be used as the body of the single abstract method in the functional interface—and returns an instance of the class implementing the interface.

In addition to returning values, SAM constructors are used when you need to store a functional interface instance generated from a lambda in a variable. Suppose you want to reuse one listener for several buttons, as in the following listing (in an Android application, this code can be a part of the `Activity.onCreate` method).

**Listing 5.13. Using a SAM constructor to reuse a listener instance**

```
val listener = OnClickListener { view -> #1
    val text = when (view.id) { #2
        R.id.button1 -> "First button"
        R.id.button2 -> "Second button"
        else -> "Unknown button"
    }
    toast(text) #3
}
button1.setOnClickListener(listener)
button2.setOnClickListener(listener)
```

listener checks which button was the source of the click and behaves accordingly. You could define a listener by using an object declaration that implements `onClickListener`, but SAM constructors give you a more concise option.

**Lambdas and adding/removing listeners**

Note that there's no `this` in a lambda as there is in an anonymous object: there's no way to refer to the anonymous class instance into which the lambda is converted. From the compiler's point of view, the lambda is a

block of code, not an object, and you can't refer to it as an object. The `this` reference in a lambda refers to a surrounding class.

If your event listener needs to unsubscribe itself while handling an event, you can't use a lambda for that. Use an anonymous object to implement a listener, instead. In an anonymous object, the `this` keyword refers to the instance of that object, and you can pass it to the API that removes the listener.

Also, even though SAM conversion in method calls typically happens automatically, there are cases when the compiler can't choose the right overload when you pass a lambda as an argument to an overloaded method. In those cases, applying an explicit SAM constructor is a good way to resolve the compilation error.

## 5.3 Defining SAM interfaces in Kotlin: "fun" interfaces

In Kotlin, you can often use function types to express behavior where you would otherwise have to use a functional interface. In [Chapter 11](#), we'll see a way to give more expressive names to function types in Kotlin through type aliases.

However, there may be a few cases where you want to be more explicit in your code. By declaring a `fun` `interface` in Kotlin, you can define your own functional interfaces.

Functional interfaces in Kotlin contain exactly one abstract method, but can also contain several additional non-abstract methods. This can help you express more complex constructs which you couldn't fit into a function type's signature.

In this example, you define a functional interface called `IntCondition` with an abstract method `check`. You define an additional, non-abstract method called `checkString`, which invokes `check` after converting its parameter to an integer. Like with Java SAMs, you use the SAM constructor to instantiate the interface with a lambda that specifies the implementation of `check`:

**Listing 5.14. A functional interface in Kotlin contains exactly one abstract method, but may contain additional non-abstract methods.**

```
fun interface IntCondition {  
    fun check(i: Int): Boolean #1  
    fun checkString(s: String) = check(s.toInt()) #2  
    fun checkChar(c: Char) = check(c.digitToInt()) #2  
}  
  
fun main() {  
    val isOdd = IntCondition { it % 2 != 0 }  
    println(isOdd.check(1))  
    // true  
    println(isOdd.checkString("2"))  
    // false  
    println(isOdd.checkChar('3'))  
    // true  
}
```

When a function accepts a parameter whose type is defined as a `fun interface`, you may once again also just provide a lambda implementation directly, or pass a reference to a lambda, both of which dynamically instantiate the interface implementation.

In the following example, you’re defining a `checkCondition` function that takes an `IntCondition` as we previously defined it. You then have multiple options of calling that function, for example by passing a lambda directly, or by passing a reference to a function that has the correct type `((Int) → Boolean)`:

**Listing 5.15. Whether you pass a lambda directly or pass a reference to a function with the correct signature, the functional interface is dynamically instantiated.**

```
fun checkCondition(i: Int, condition: IntCondition): Boolean {  
    return condition.check(i)  
}  
  
fun main() {  
    checkCondition(1) { it % 2 != 0 } #1  
    // true  
    val isOdd: (Int) -> Boolean = { it % 2 != 0 }  
    checkCondition(1, isOdd) #2  
    // true  
}
```

## Cleaner Java call sites with functional interfaces

If you're writing code that you expect to be used from both Java and Kotlin code, using a `fun interface` can also improve the cleanliness of the Java call sites. Kotlin function types are translated as objects whose generic types are the parameters and return types. For functions that don't return anything, Kotlin uses `Unit` as its analog to Java's `void` (we will talk about necessity and usefulness of the `Unit` type in more depth in [8.1.6](#)).

That also means that when such a Kotlin function type is invoked from Java, the caller needs to explicitly return `Unit.INSTANCE`. Using a `fun interface` takes that requirement away, making the call site more concise. In this example, the functions `consumeHello` and `consumeHelloFunctional` do the same thing, but are defined once using a functional interface, and once using Kotlin functional types:

```
fun interface StringConsumer {
    fun consume(s: String)
}

fun consumeHello(t: StringConsumer) {
    t.consume("Hello")
}

fun consumeHelloFunctional(t: (String) -> Unit) {
    t("Hello")
}
```

When used from Java, the variant of the function using a `fun interface` can be called with a simple lambda, whereas the variant using Kotlin functional types requires the lambda to explicitly return Kotlin's `Unit.INSTANCE`:

```
import kotlin.Unit;

public class MyApp {
    public static void main(String[] args) {
        /* Java */
        MainKt.consumeHello(s -> System.out.println(s.toUpperCase()));
        MainKt.consumeHelloFunctional(s -> {
            System.out.println(s.toUpperCase());
            return Unit.INSTANCE; #1
        });
    }
}
```

```
}
```

As a general rule of thumb, simple functional types work well if your API can accept any function that takes a specific set of parameters and returns a specific type. When you need to express more complex contracts, or operations that you can't express in a functional type signature, a functional interface is a good choice.

We'll further discuss the use of function types in function declarations in **Chapter 10**, and take a closer look at typealiases in Kotlin in **Chapter 11**.

To finish our discussion of lambda syntax and usage, let's look at lambdas with receivers and how they're used to define convenient library functions that look like built-in constructs.

## 5.4 Lambdas with receivers: "with", "apply", and "also"

This section demonstrates the `with` and `apply` functions from the Kotlin standard library. These functions are convenient, and you'll find many uses for them even without understanding how they're declared. Later, in **Chapter 13**, you'll see how you can declare similar functions for your own needs. The explanations in this section, however, help you become familiar with a unique feature of Kotlin's lambdas that isn't available with Java: the ability to call methods of a different object in the body of a lambda without any additional qualifiers. Such lambdas are called lambdas with receivers. Let's begin by looking at the `with` function, which uses a lambda with a receiver.

### 5.4.1 Performing multiple operations on the same object: "with"

Many languages have special statements you can use to perform multiple operations on the same object without repeating its name. Kotlin also has this facility, but it's provided as a library function called `with`, not as a special language construct.

To see how it can be useful, consider the following example, which you'll

then refactor using `with`.

#### **Listing 5.16. Building the alphabet**

```
fun alphabet(): String {
    val result = StringBuilder()
    for (letter in 'A'...'Z') {
        result.append(letter)
    }
    result.append("\nNow I know the alphabet!")
    return result.toString()
}

fun main() {
    println(alphabet())
    // ABCDEFGHIJKLMNOPQRSTUVWXYZ
    // Now I know the alphabet!
}
```

In this example, you call several different methods on the `result` instance and repeating the `result` name in each call. This isn't too bad, but what if the expression you were using was longer or repeated more often?

Here's how you can rewrite the code using `with`.

#### **Listing 5.17. Using `with` to build the alphabet**

```
fun alphabet(): String {
    val stringBuilder = StringBuilder()
    return with(stringBuilder) { #1
        for (letter in 'A'...'Z') {
            this.append(letter) #2
        }
        this.append("\nNow I know the alphabet!") #3
        this.toString() #4
    }
}
```

The `with` structure looks like a special construct, but it's a function that takes two arguments: `stringBuilder`, in this case, and a lambda. The convention of putting the lambda outside of the parentheses works here, and the entire invocation looks like a built-in feature of the language. Alternatively, you could write this as `with(stringBuilder, { ... })`, but it's less readable.

The `with` function converts its first argument into a *receiver* of the lambda that's passed as a second argument. You can access this receiver via an explicit `this` reference. Alternatively, as usual for a `this` reference, you can omit it and access methods or properties of this value without any additional qualifiers.

**Figure 5.8.** Inside the lambda of the `with` function, the first argument is available as the receiver type `this`. IDEs like IntelliJ IDEA and Android Studio have the option to visualize this receiver type via an inlay hint after the opening parenthesis.

```
val stringBuilder = StringBuilder()  
with(stringBuilder) { this: StringBuilder  
    // ...  
}
```

In [5.17](#), `this` refers to `stringBuilder`, which is passed to `with` as the first argument. You can access methods on `stringBuilder` via explicit `this` references, as in `this.append(letter)`; or directly, making your code even more concise:

**Listing 5.18.** You don't need to specify `this` explicitly inside the `with` lambda.

```
fun alphabet(): String {  
    val stringBuilder = StringBuilder()  
    return with(stringBuilder) { #1  
        for (letter in 'A'..'Z') {  
            append(letter) #1  
        }  
        append("\nNow I know the alphabet!") #1
```

```
        toString() #1
    }
}
```

### Lambdas with receiver and extension functions

You may recall that you saw a similar concept with this referring to the function receiver. In the body of an extension function, this refers to the instance of the type the function is extending, and it can be omitted to give you direct access to the receiver's members.

Note that an extension function is, in a sense, a function with a receiver. The following analogy can be applied:

Regular function	Regular lambda
Extension function	Lambda with a receiver

A lambda is a way to define behavior similar to a regular function. A lambda with a receiver is a way to define behavior similar to an extension function.

Let's refactor the initial alphabet function even further and get rid of the extra `StringBuilder` variable.

#### **Listing 5.19. Using with and an expression body to build the alphabet**

```
fun alphabet() = with(StringBuilder()) {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
    toString()
}
```

This function now only returns an expression, so it's rewritten using the expression-body syntax. You create a new instance of `StringBuilder` and pass it directly as an argument, and then you reference it without the explicit

this in the lambda.

#### Method-name conflicts

What happens if the object you pass as a parameter to with has a method with the same name as the class in which you're using with? In this case, you can add an explicit label to the this reference to specify which method you need to call.

Imagine that the alphabet function is a method of the class OuterClass. If you need to refer to the `toString` method defined in the outer class instead of the one in `StringBuilder`, you can do so using the following syntax:

```
this@OuterClass.toString()
```

The value that with returns is the result of executing the lambda code. The result is the last expression in the lambda. But sometimes you want the call to return the receiver object, not the result of executing the lambda. That's where the apply library function can be of use.

### 5.4.2 Initializing and configuring objects: "apply"

The apply function works almost exactly the same as with ; the only difference is that apply always returns the object passed to it as an argument (in other words, the receiver object). Let's refactor the alphabet function again, this time using apply.

#### Listing 5.20. Using apply to build the alphabet

```
fun alphabet() = StringBuilder().apply {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
}.toString()
```

You can call the apply function as an extension function on any type – in this case, you're calling it on your newly created `StringBuilder` instance. Its receiver becomes the receiver of the lambda passed as an argument. The

result of executing `apply` is `StringBuilder`, so you call `toString` to convert it to `String` afterward.

**Figure 5.9.** Like the `with` function, `apply` makes the object it was called on the receiver type inside the lambda. `apply` also returns the object it was called on. Inlay hints in IntelliJ IDEA and Android Studio help visualize this.

```
val result: StringBuilder = StringBuilder().apply { this: StringBuilder  
    // ...  
}
```

One of many cases where this is useful is when you’re creating an instance of an object and need to initialize some properties right away. In Java, this is usually accomplished through a separate `Builder` object; and in Kotlin, you can use `apply` on any object without any special support from the library where the object is defined.

To see how `apply` is used for such cases, let’s look at an example that creates an Android `TextView` component with some custom attributes.

**Listing 5.21. Using `apply` to initialize a `TextView`**

```
fun createViewWithCustomAttributes(context: Context) =  
    TextView(context).apply {  
        text = "Sample Text"  
        textSize = 20.0  
        setPadding(10, 0, 0, 0)  
    }
```

The `apply` function allows you to use the compact expression body style for the function. You create a new `TextView` instance and immediately pass it to

`apply`. In the lambda passed to `apply`, the `TextView` instance becomes the receiver, so you can call methods and set properties on it. After the lambda is executed, `apply` returns that instance, which is already initialized; it becomes the result of the `createViewWithCustomAttributes` function.

The `with` and `apply` functions are basic generic examples of using lambdas with receivers. More specific functions can also use the same pattern. For example, you can simplify the `alphabet` function even further by using the `buildString` standard library function, which will take care of creating a `StringBuilder` and calling `toString`. The argument of `buildString` is a lambda with a receiver, and the receiver is always a `StringBuilder`.

**Listing 5.22. Using `buildString` to build the alphabet**

```
fun alphabet() = buildString {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
}
```

The `buildString` function is an elegant solution for the task of creating a `String` with the help of `StringBuilder`. The Kotlin standard library also comes with collection builder functions, which help you create a read-only, `List`, `Set` or `Map` while allowing you to treat the collection as mutable during the construction phase:

**Listing 5.23. Using `buildList` and `buildMap` to create collections**

```
val fibonacci = buildList {
    addAll(listOf(1, 1, 2))
    add(3)
    add(index = 0, element = 3)
}

val shouldAdd = true

val fruits = buildSet {
    add("Apple")
    if(shouldAdd) {
        addAll(listOf("Apple", "Banana", "Cherry"))
    }
}
```

```
}

val medals = buildMap<String, Int> {
    put("Gold", 1)
    putAll(listOf("Silver" to 2, "Bronze" to 3))
}
```

### 5.4.3 Performing additional actions with an object: "also"

Just like apply, you can use the also function to take a receiver object, perform an action on it, and then return the receiver object. The main difference is that within the lambda of also, you access the receiver object as an argument – either by giving it a name, or by using the default name `it`. This makes also a good fit for running actions that take the original receiver object as an argument (as opposed to operations that work on the object's properties and functions). When you see also in the code, you can interpret it as executing additional effects: "...and also do the following with the object."

**Figure 5.10.** When using also, the object doesn't become the receiver type inside the lambda, but makes the object available as an argument, by default named `it`. The also function returns the object it was called on, as you can see in these inlay hints.

```
val x: List<Int> = listOf(1, 2, 3).also { it: List<Int>
    // ...
}
```

In the following example, you're mapping a collection of fruits to their uppercased names, and *also* add the result of that mapping to an additional collection. You then filter for those fruits from the collection whose name is longer than five characters, and *also* print that result, before finally reversing

the list:

**Listing 5.24. Using `also` to perform additional effects**

```
fun main() {
    val fruits = listOf("Apple", "Banana", "Cherry")
    val uppercaseFruits = mutableListOf<String>()
    val reversedLongFruits = fruits
        .map { it.uppercase() }
        .also { uppercaseFruits.addAll(it) }
        .filter { it.length > 5 }
        .also { println(it) }
        .reversed()
    // [BANANA, CHERRY]
    println(uppercaseFruits)
    // [APPLE, BANANA, CHERRY]
    println(reversedLongFruits)
    // [CHERRY, BANANA]
}
```

You'll see more interesting examples for lambdas with receivers in [Chapter 13](#), when we begin discussing domain-specific languages. Lambdas with receivers are great tools for building DSLs; we'll show you how to use them for that purpose and how to define your own functions that call lambdas with receivers.

## 5.5 Summary

- Lambdas allow you to pass chunks of code as arguments to functions, so you can easily extract common code structures.
- Kotlin lets you pass lambdas to functions outside of parentheses to make your code clean and concise.
- If a lambda only takes a single parameter, you can refer to it with its implicit name `it`. This saves you the effort of explicitly naming the only lambda parameter in short and simple lambdas.
- Lambdas can *capture* external variables. That means you can, for example, use your lambdas to access and modify variables in the function containing the call to the lambda.
- You can create references to methods, constructors, and properties by prefixing the name of the function with `::`. You can pass such references

- to functions instead of lambdas as a shorthand.
- To implement interfaces with a single abstract method (also known as SAM interfaces), you can simply pass lambdas instead of having to create an object implementing the interface explicitly.
- *Lambdas with receivers* are lambdas that allow you to directly call methods on a special receiver object. Because the body of these lambdas is executed in a different context than the surrounding code, they can help with structuring your code.
- The `with` standard library function allows you to call multiple methods on the same object without repeating the reference to the object. `apply` lets you construct and initialize any object using a builder-style API. `also` lets you perform additional actions with an object.

# 6 Working with collections and sequences

## This chapter covers

- Working with collections in a functional style
- Sequences: performing collection operations lazily

In 5, you learned about lambdas as a way of passing small blocks of code to other functions. One of the most common uses for lambdas is *working with collections*. In this chapter, you will see how replacing common collection access patterns with a combination of standard library functions and your own lambdas can make your code more expressive, elegant, and concise—whether you’re filtering your data based on predicates, need to group data, or transform collection items from one form to another.

You will also explore *Sequences* as an alternative way to apply multiple collection operations efficiently and without creating much overhead. You will learn the difference between *eager* and *lazy* execution of collection operations in Kotlin, and how to use either one in your programs.

## 6.1 Functional APIs for collections

A functional programming style provides many benefits when it comes to manipulating collections. For the majority of tasks, you can use functions provided by the standard library, and customize their behaviour by passing lambdas as arguments to these functions. Compared to navigating through collections and aggregating data manually, this allows you to express common operations consistently using a vocabulary of functions that you share with other Kotlin developers.

In this section, we’ll explore some of the functions in the Kotlin standard library that you might find yourself reaching for when working with

collections. We'll start with staple functions like `filter` and `map` that help you transform and the concepts behind them. We'll also cover other useful functions and give you tips about how not to overuse them and how to write clear and comprehensible code.

Note that these functions weren't invented by the designers of Kotlin. These or similar functions are available for all languages that support lambdas, including C#, Groovy, and Scala. If you're already familiar with these concepts, you can quickly look through the following examples and skip the explanations.

### 6.1.1 Removing and transforming elements: `filter` and `map`

The `filter` and `map` functions form the basis for manipulating collections. Many collection operations can be expressed with their help. Whenever you're faced with a task of filtering a collection based on a specific predicate, or you need to transform each element of a collection into a different form, these functions should come to mind.

For each function, we'll provide one example with numbers and one using the familiar `Person` class:

```
data class Person(val name: String, val age: Int)
```

The `filter` function goes through a collection and selects the elements for which the given lambda returns `true`. For example, given a list of some numbers, `filter` can help you extract only the even numbers (where the remainder of a division by 2 is zero):

```
fun main() {
    val list = listOf(1, 2, 3, 4)
    println(list.filter { it % 2 == 0 }) #1
    // [2, 4]
}
```

The result is a new collection that contains only the elements from the input collection that satisfy the predicate, as illustrated in [6.1](#).

**Figure 6.1. The `filter` function selects elements matching given predicate**



If you want to keep only people older than 30, you can use `filter`:

```
fun main() {
    val people = listOf(Person("Alice", 29), Person("Bob", 31))
    println(people.filter { it.age > 30 })
    // [Person(name=Bob, age=31)]
}
```

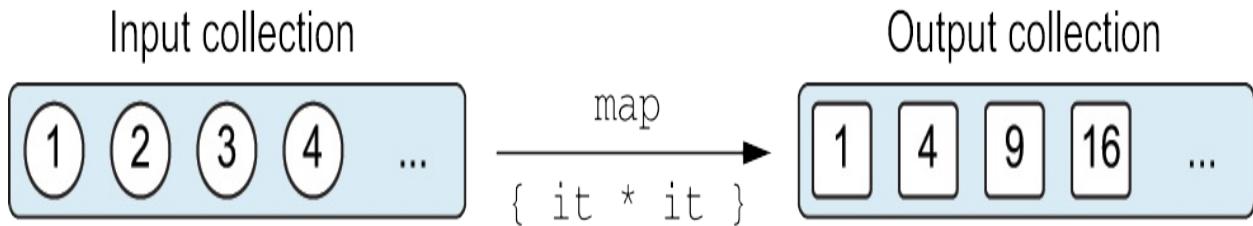
The `filter` function can create a new collection of elements that match a given predicate, but does not *transform* the elements in the process—your output is still a collection of `Person` objects. You can think of it as "extracting" entries from your collection, without changing their type.

Compare this to the `map` function, which allows you to transform the elements of your input collection. It applies the given function to each element in the collection and collects the return values into a new collection. You could use it to transform a list of numbers into a list of their squares, for example:

```
fun main() {
    val list = listOf(1, 2, 3, 4)
    println(list.map { it * it })
    // [1, 4, 9, 16]
}
```

The result is a new collection that contains the same number of elements, but each element is has been transformed according to the given predicate function (see [6.2](#)).

**Figure 6.2.** The `map` function applies a lambda to all elements in a collection.



If you want to print just a list of names, not a list of people, you can transform the list using `map`:

```
fun main() {
    val people = listOf(Person("Alice", 29), Person("Bob", 31))
    println(people.map { it.name })
    // [Alice, Bob]
}
```

Note that this example can be nicely rewritten using member references:

```
people.map(Person::name)
```

You can easily chain several calls like that. For example, let's find the names of people older than 30:

```
println(people.filter { it.age > 30 }.map(Person::name))
// [Bob]
```

Now, let's say you need to find the oldest people in the group. You can find the maximum age of the people in the group and return everyone who is that age. It's easy to write such code using lambdas:

```
people.filter {
    val oldestPerson = people.maxByOrNull(Person::age)
    it.age == oldestPerson?.age
}
```

But note that this code repeats the process of finding the maximum age for every person, so if there are 100 people in the collection, the search for the maximum age will be performed 100 times!

The following solution improves on that and calculates the maximum age just once:

```
val maxAge = people.maxByOrNull(Person::age)?.age
people.filter { it.age == maxAge }
```

Don't repeat a calculation if you don't need to! Simple-looking code using lambda expressions can sometimes obscure the complexity of the underlying operations. Always keep in mind what is happening in the code you write.

If your filtering and transformation operations depend on the index of the elements in addition to their actual values, you can use the sibling functions `filterIndexed` and `mapIndexed`, which provide your lambda with both the index of an element, as well as the element itself. In this example, you're filtering a list of numbers to only contain those values at an even index and greater than 3. You are also mapping a second list to sum the index number and the numerical value of each element:

```
fun main() {
    val numbers = listOf(1, 2, 3, 4, 5, 6, 7)
    val filtered = numbers.filterIndexed { index, element ->
        index % 2 == 0 && element > 3
    }
    println(filtered)
    // [5, 7]

    val mapped = numbers.mapIndexed { index, element ->
        index + element
    }
    println(mapped)
}
// [5, 7]
// [1, 3, 5, 7, 9, 11, 13]
```

Filter and transformation functions can also be applied to maps:

```
fun main() {
    val numbers = mapOf(0 to "zero", 1 to "one")
    println(numbers.mapValues { it.value.uppercase() })
    // {0=ZERO, 1=ONE}
}
```

There are separate functions to handle keys and values. `filterKeys` and `mapKeys` filter and transform the keys of a map, respectively, whereas `filterValues` and `mapValues` filter and transform the corresponding values.

## 6.1.2 Accumulating values for collections: reduce and fold

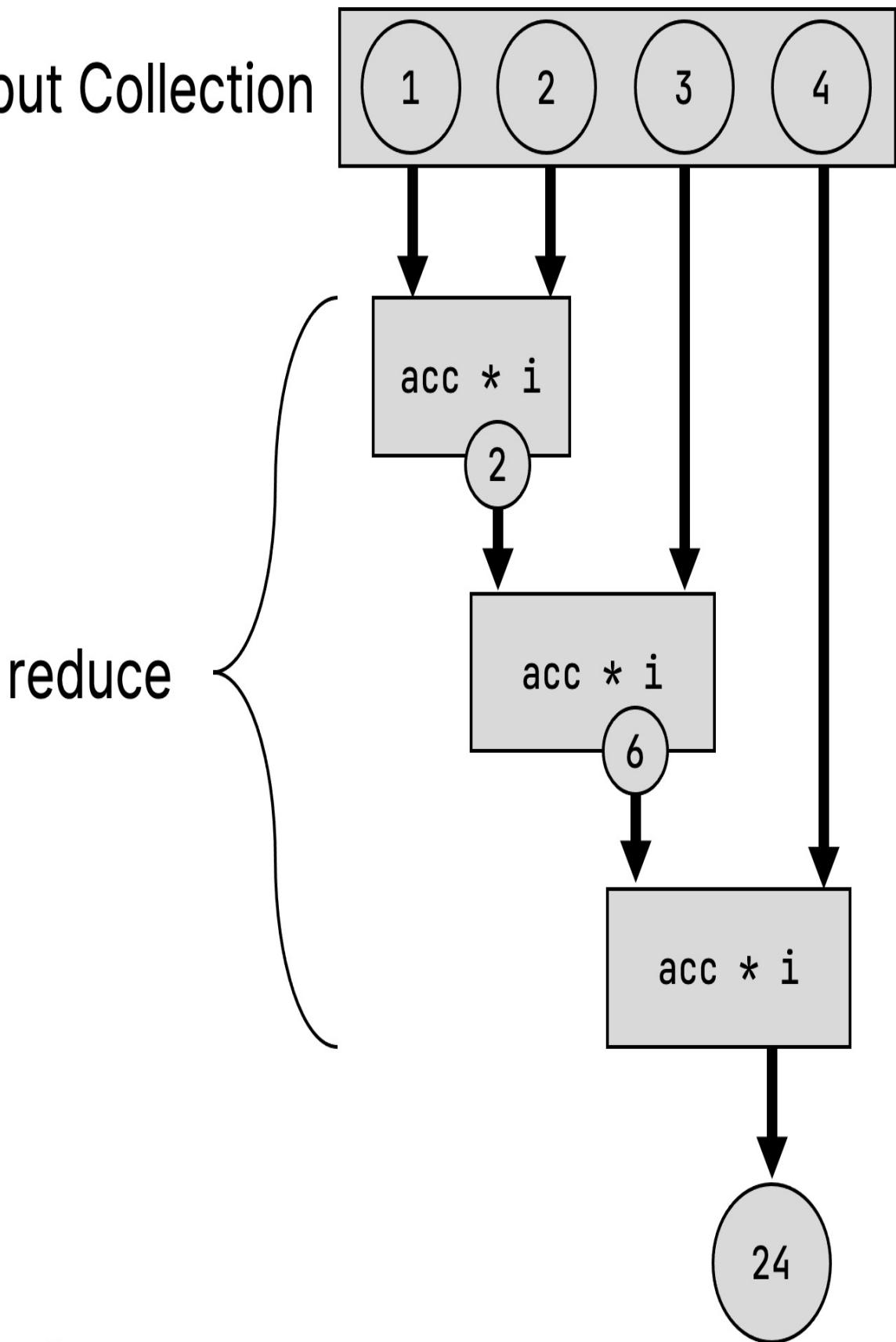
Next to the `filter` and `map` functions, `reduce` and `fold` are two more essential building blocks when working with collections in a functional style. These functions are used to *aggregate* information from a collection: given a collection of items, they return a single value. This value is gradually built up in a so-called "accumulator". Your lambda is invoked for each element, needs to return a new accumulator value.

When using `reduce`, you start with the first element of your collection in the accumulator (so don't call it on an empty collection!) Your lambda is then called with the accumulator and the second element. In this example, you use `reduce` to sum and multiply the values of the input collection, respectively:

```
fun main() {
    val list = listOf(1, 2, 3, 4)
    val summed = list.reduce { acc, element ->
        acc + element
    }
    println(summed)
    // 10
    val multiplied = list.reduce { acc, element ->
        acc * element
    }
    println(multiplied)
    // 24
}
```

Figure 6.3. The `reduce` function gradually builds up a result in its accumulator, invoking your lambda repeatedly with each element and the previous accumulator value.

# Input Collection



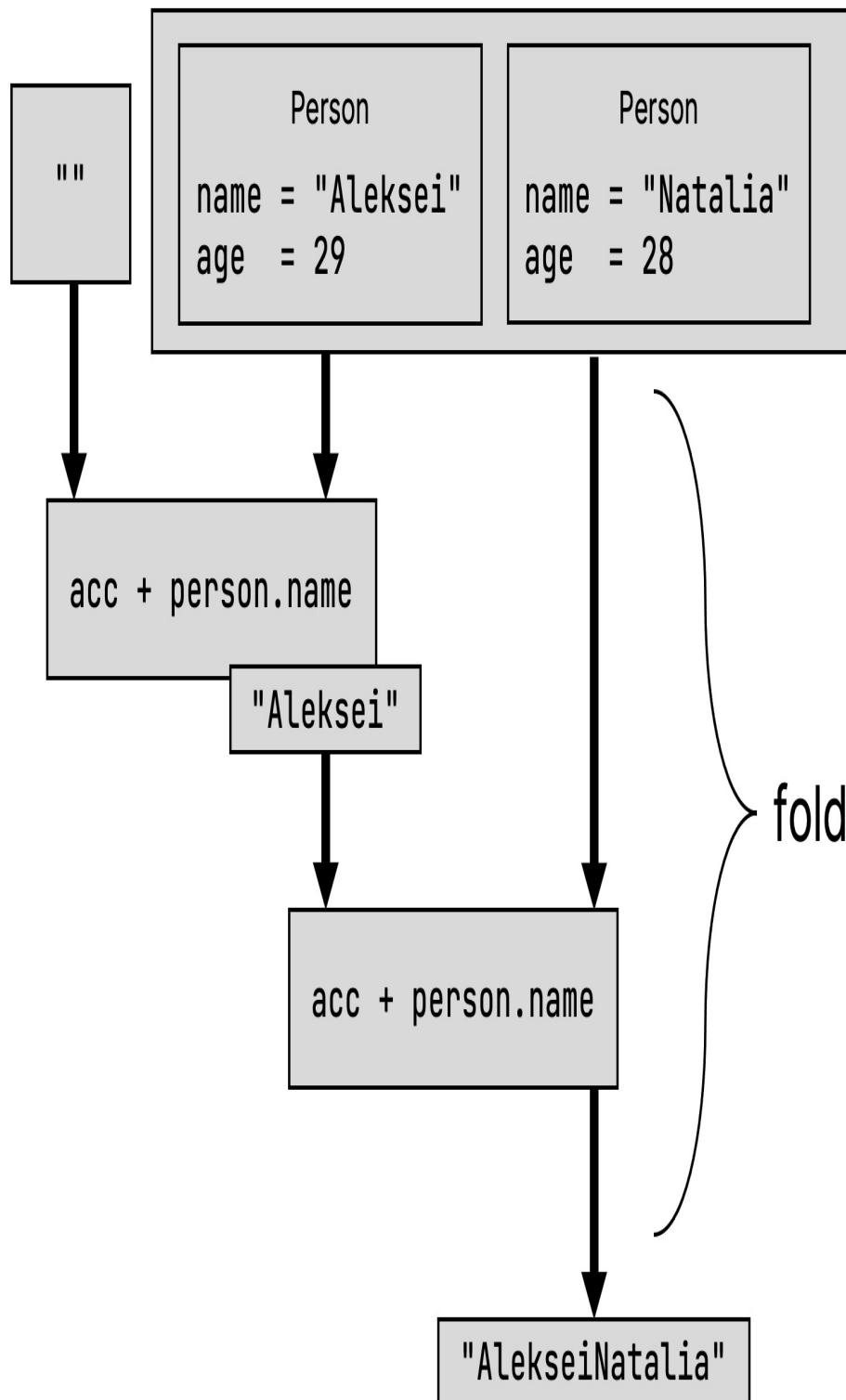
The `fold` function is conceptually very similar to `reduce`, but instead of putting the first element of your collection into the accumulator at the beginning, you can choose an arbitrary start value. In this example, you’re concatenating the `name` property of two `Person` objects using `fold` (a job usually tailored for the `joinToString` function we explored earlier, but an illustrative example nonetheless). You initialize the accumulator with an empty string, and then gradually build up the final text in your lambda:

```
fun main() {
    val people = listOf(
        Person("Aleksei", 29),
        Person("Natalia", 28)
    )
    val folded = people.fold("") { acc, person ->
        acc + person.name
    }
    println(folded)
}
// AlekseiNatalia
```

**Figure 6.4.** The `fold` function allows you to specify the initial value and type of the accumulator. The result is gradually built up in the accumulator, applying your lambda repeatedly on each accumulator-element pair.

## Input Collection

Accumulator  
start value



There are many algorithms you can express concisely using a `fold` or `reduce` operation. For cases where you want to retrieve all intermittent values the `reduce` or `fold` operations, the `runningReduce` and `runningFold` functions come to the rescue. Their only difference to the `reduce` and `fold` functions as we just discussed them is that these functions return a *list*. It contains all intermittent accumulator values alongside the final result. In this example, you're using the running counterpart of the snippets discussed in the previous paragraphs:

```
fun main() {
    val list = listOf(1, 2, 3, 4)
    val summed = list.runningReduce { acc, element ->
        acc + element
    }
    println(summed)
    // [1, 3, 6, 10] #1
    val multiplied = list.runningReduce { acc, element ->
        acc * element
    }
    println(multiplied)
    // [1, 2, 6, 24] #1
    val people = listOf(
        Person("Aleksei", 29),
        Person("Natalia", 28)
    )
    println(people.runningFold("") { acc, person ->
        acc + person.name
    })
    // [, Aleksei, AlekseiNatalia] #1
}
```

### 6.1.3 Applying a predicate to a collection: `all`, `any`, `none`, `count`, and `find`

Another common task is checking whether all, some, or no elements in a collection match a certain condition. In Kotlin, this is expressed through the `all`, `any`, and `none` functions. The `count` function checks how many elements satisfy the predicate, and the `find` function returns the first matching element.

To demonstrate those functions, let's define the predicate `canBeInClub27` to check whether a person is 27 or younger:

```
val canBeInClub27 = { p: Person -> p.age <= 27 }
```

If you're interested in whether all the elements satisfy this predicate, you use the `all` function:

```
fun main() {
    val people = listOf(Person("Alice", 27), Person("Bob", 31))
    println(people.all(canBeInClub27))
}
false
```

If you need to check whether there's at least one matching element, use `any`:

```
fun main() {
    println(people.any(canBeInClub27))
    // true
}
```

Note that `!all` ("not all") with a condition can be replaced with `any` with a negation of that condition, and vice versa. To make your code easier to understand, you should choose a function that doesn't require you to put a negation sign before it:

```
fun main() {
    val list = listOf(1, 2, 3)
    println(!list.all { it == 3 }) #1
    // true
    println(list.any { it != 3 }) #2
    // true
}
```

The first check ensures that not all elements are equal to 3. That's the same as having at least one non-3, which is what you check using `any` on the second line.

Likewise, you can replace `!any` with `none`:

```
fun main() {
    val list = listOf(1, 2, 3)
    println(!list.any { it == 4 }) #1
    // true
    println(list.none { it == 4 }) #2
    // true
}
```

```
}
```

The first invocation checks whether there are any elements in the collection are equal to 4, and negates that result. A more natural way of expressing this, both in code and words, is to check whether none of the elements are equal to 4.

### Predicates and empty collections

While reading the description of the different types of predicates—any, none, and all—you may have started to wonder what these functions return when called on empty collections. Let's address this mystery.

In the case of any, the collection contains no elements that can satisfy the provided predicate. That means it returns `false`:

```
fun main() {
    println(listOf<Int>().any { it > 42 })
    // false
}
```

As you've seen, the `none` function is the inverse of the `any` function. That's also reflected in the case of an empty collection: There is no element that can violate the predicate, so the function returns `true`:

```
fun main() {
    println(listOf<Int>().none { it > 42 })
    // true
}
```

The function with the perhaps largest mind-bending potential is `all`. Regardless of the predicate, it returns `true` when called on an empty collection:

```
fun main() {
    println(listOf<Int>().all { it > 42 })
    // true
}
```

This might surprise you at first, but upon further investigation, you'll find that this is a very reasonable return value. You can't name an element that

violates the predicate, so the predicate clearly has to be true for *all* elements in the collection—even if there are none! This concept is known as the *vacuous truth*, and actually ends up a usually good fit for conditionals that should also work with empty collections.

If you want to know how many elements satisfy a predicate, use `count`:

```
fun main() {
    val people = listOf(Person("Alice", 27), Person("Bob", 31))
    println(people.count(canBeInClub27))
    // 1
}
```

**Using the right function for the job: "count" vs. "size"**

It's easy to forget about `count` and implement it by filtering the collection and getting its size:

```
println(people.filter(canBeInClub27).size)
// 1
```

But in this case, an intermediate collection is created to store all the elements that satisfy the predicate. On the other hand, the `count` method tracks only the number of matching elements, not the elements themselves, and is therefore more efficient.

As a general rule, try to find the most appropriate operation that suits your needs.

To find an element that satisfies the predicate, use the `find` function:

```
fun main() {
    val people = listOf(Person("Alice", 27), Person("Bob", 31))
    println(people.find(canBeInClub27))
    // Person(name=Alice, age=27)
}
```

This returns the first matching element if there are many or `null` if nothing satisfies the predicate. A synonym of `find` is `firstOrNull`, which you can use if it expresses the idea more clearly for you.

## 6.1.4 Splitting a list to a pair of lists: partition

In some situations, you need to divide a collection into two groups based on a given predicate: Elements that fulfill a Boolean predicate, and those that don't. If you need both groups, you could use the functions `filter` and its sibling `filterNot` (which inverts the predicate) to create these two lists. In this example, you're finding out who is allowed in the club—and who isn't:

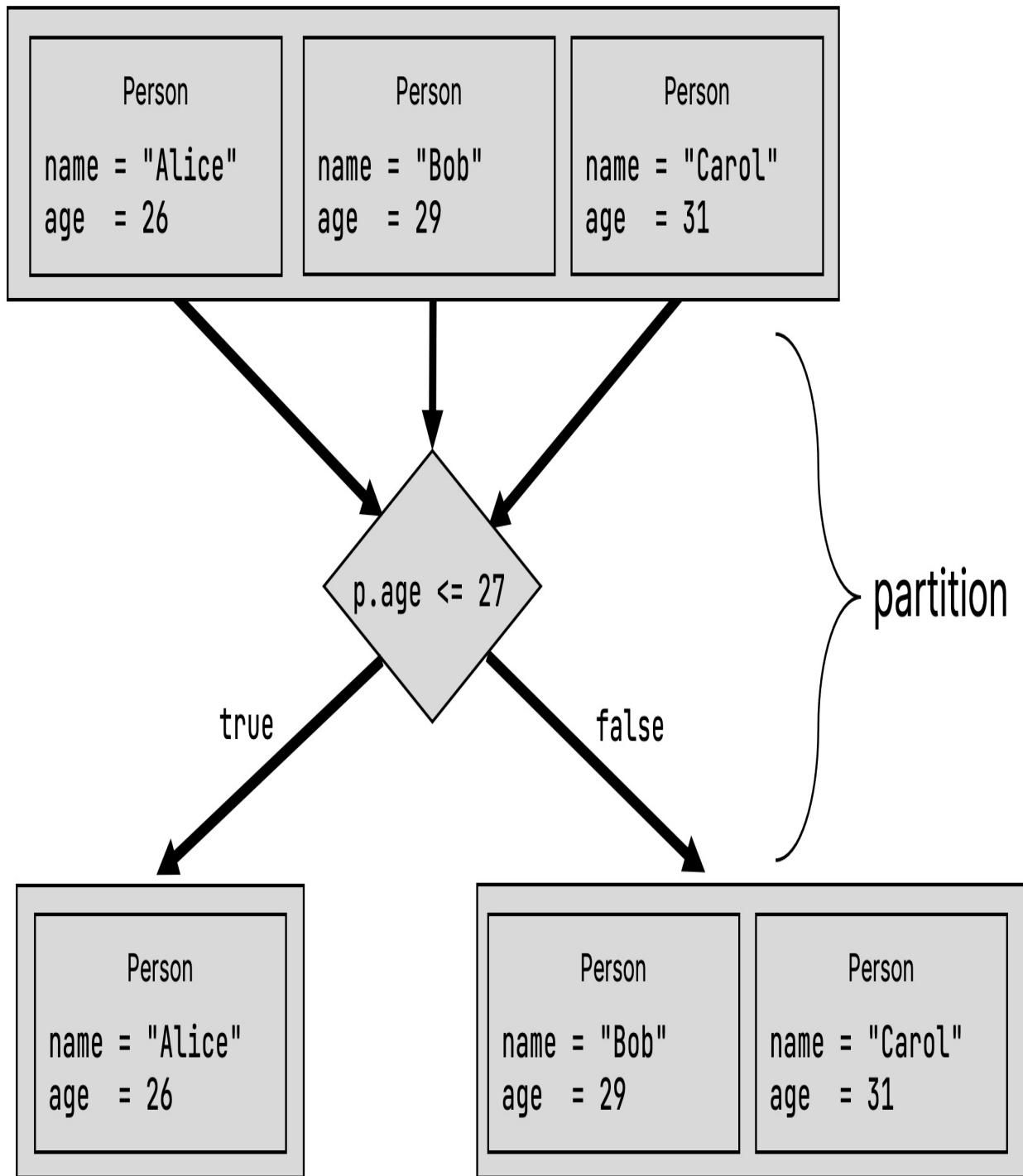
```
fun main() {
    val people = listOf(
        Person("Alice", 26),
        Person("Bob", 29),
        Person("Carol", 31)
    )
    val comeIn = people.filter(canBeInClub27)
    val stayOut = people.filterNot(canBeInClub27)
    println(comeIn)
    // [Person(name=Alice, age=26)]
    println(stayOut)
    // [Person(name=Bob, age=29), Person(name=Carol, age=31)]
}
```

But there is also a more concise way to do this: using the `partition` function. It returns this pair of lists, without having to repeat the predicate, and without having to iterate the input collection twice. This means you can express the same logic from the previous code snippet as follows:

```
val (comeIn, stayOut) = people.partition(canBeInClub27) #1
println(comeIn)
// [Person(name=Alice, age=26)]
println(stayOut)
// [Person(name=Bob, age=29), Person(name=Carol, age=31)]
```

**Figure 6.5.** The `partition` function returns a pair consisting of two lists: Those that satisfy the given Boolean predicate, and those that don't.

## Input Collection



## Output Collections

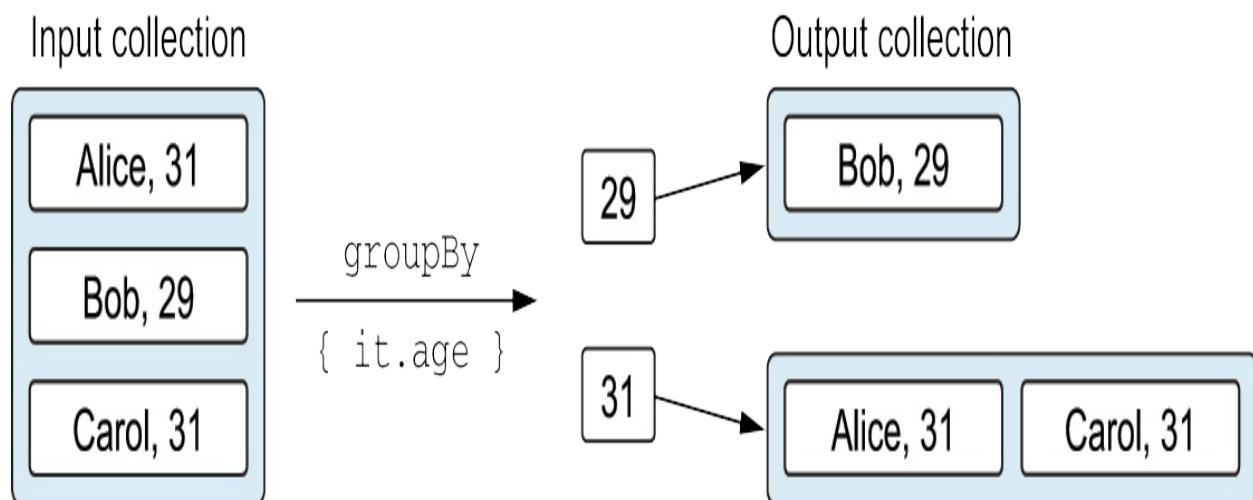
## 6.1.5 Converting a list to a map of groups: `groupBy`

Often enough, the elements in a collection can't be clustered into just the "true" and "false" groups that `partition` returns. Instead, you may want to divide all elements into different groups according to some *quality*. For example, you want to group people of the same age. It's convenient to pass this quality directly as a parameter. The `groupBy` function can do this for you:

```
fun main() {
    val people = listOf(
        Person("Alice", 31),
        Person("Bob", 29),
        Person("Carol", 31)
    )
    println(people.groupBy { it.age })
}
```

The result of this operation is a map from the key by which the elements are grouped (age, in this case) to the groups of elements (persons); see [6.6](#).

**Figure 6.6. The result of applying the `groupBy` function**



For this example, the output is as follows:

```
{29=[Person(name=Bob, age=29)], 31=[Person(name=Alice, age=31), Person(name=Carol, age=31)]}
```

Each group is stored in a list, so the result type is `Map<Int, List<Person>>`. You can do further modifications with this map, using functions such as `mapKeys` and `mapValues`.

As another example, let's see how to group strings by their first character using member references:

```
fun main() {
    val list = listOf("apple", "apricot", "banana", "cantaloupe")
    println(list.groupBy(String::first))
    // {a=[apple, apricot], b=[banana], c=[cantaloupe]}
}
```

Note that `first` here isn't a member of the `String` class, it's an extension. Nevertheless, you can access it as a member reference.

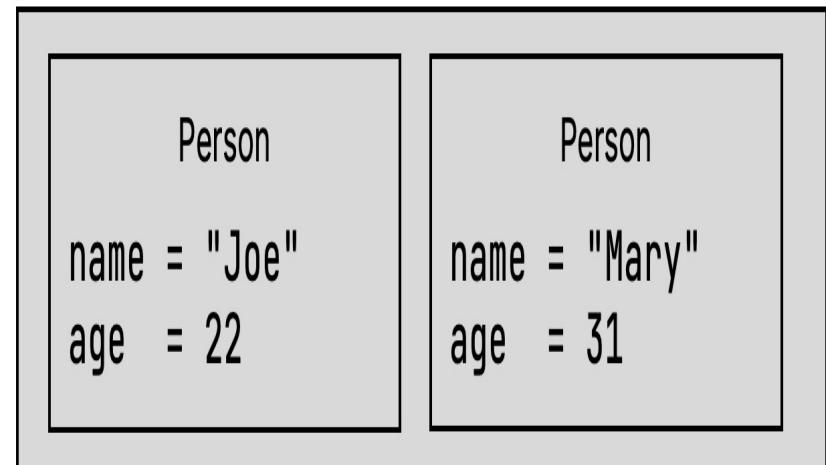
### 6.1.6 Transforming collections into maps: `associate`, `associateWith` and `associateBy`:

With `groupBy`, you now already know a way of creating an associative data structure from a list—by grouping elements based on a common property. If you want to create a map from the elements of a collection *without* grouping elements, the `associate` function comes into play. You provide it with a lambda that expresses a transformation from an item in your input collection to a key-value pair that will be put into a map. In this example, you use the `associate` function to turn a list of `Person` objects into a map of names to ages, and query an example value, like you would with any other `Map<String, Int>`. You use the infix function to which we introduced in [3.4.3](#) to specify the individual key-value pairs:

```
fun main() {
    val people = listOf(Person("Joe", 22), Person("Mary", 31))
    val nameToAge = people.associate { it.name to it.age } #1
    println(nameToAge)
    // {Joe=22, Mary=31}
    println(nameToAge["Joe"])
    // 22
}
```

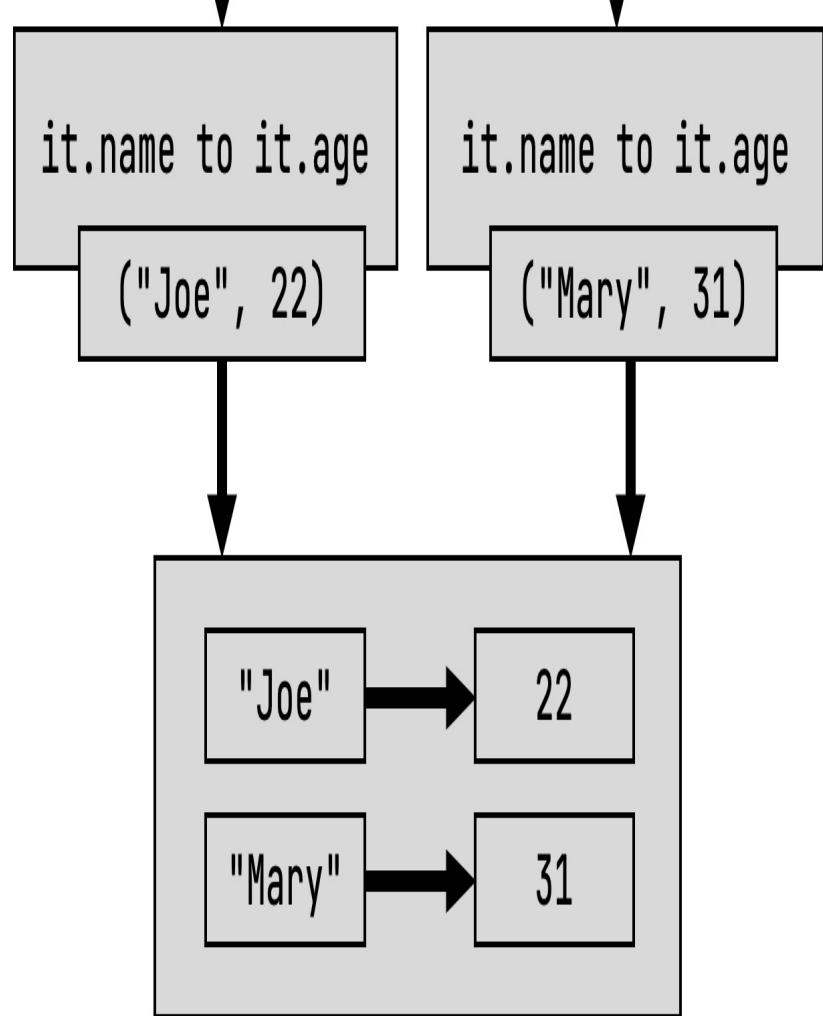
**Figure 6.7. The `associate` function turns a list into a map based on the pairs of keys and values returned by your lambda.**

Input Collection



associate

Output map



Instead of creating pairs of custom keys *and* custom values, you may want to create an association between the elements of your collection with another certain value. You can do this with the `associateWith` and `associateBy` functions.

`associateWith` uses the original elements of your collection as keys. The lambda that you provide generates the corresponding value for each key. On the other hand, `associateBy` uses the original elements of your collection as values, and uses your lambda to generate the keys of the map.

In this example, you're first creating a map of people to their ages by using the `associateWith` function. You create the inverse map, from ages to people, using the `associateBy` function:

```
fun main() {
    val people = listOf(
        Person("Joe", 22),
        Person("Mary", 31),
        Person("Jamie", 22)
    )
    val personToAge = people.associateWith { it.age }
    println(personToAge)
    // {Person(name=Joe, age=22)=22, Person(name=Mary, age=31)=31
    //  Person(name=Jamie, age=22)=22} #1
    val ageToPerson = people.associateBy { it.age } #2
    println(ageToPerson)
    // {22=Person(name=Jamie, age=22), 31=Person(name=Mary, age=3
}
```

Keep in mind that keys for maps have to be unique, and the ones generated by `associate`, `associateBy` and `associateWith` are no exception. If your transformation function would result in the same key being added multiple times, the last result overwrites any previous assignments.

### 6.1.7 Replacing elements in mutable collections: `replaceAll` and `fill`

Generally, a functional programming style encourages you to work with immutable collections, but there may still be some situations where it is more

ergonomic to work with mutable collections. For those situations, the Kotlin standard library comes with a few convenience functions to help you change the contents of your collections.

When applied to a `MutableList`, the `replaceAll` function replaces each element in the list with a result from the lambda you specify. For the special case of replacing all elements in the mutable list with the same value, you can use the `fill` function as a shorthand. In this example, you first replace all elements in your input collection with their uppercase variant, and replace all names with a placeholder text afterwards:

```
fun main() {
    val names = mutableListOf("Martin", "Samuel")
    println(names)
    // [Martin, Samuel]
    names.replaceAll { it.uppercase() }
    println(names)
    // [MARTIN, SAMUEL]
    names.fill("(redacted)")
    println(names)
    // [(redacted), (redacted)]
}
```

## 6.1.8 Handling special cases for collections: `ifEmpty`

Often, it only makes sense for a program to proceed if an input collection is not empty—so that there are some actual elements to process. With the `ifEmpty` function, you can provide a lambda that generates a default value in case your collection does not contain any elements:

```
fun main() {
    val empty = emptyList<String>()
    val full = listOf("apple", "orange", "banana")
    println(empty.ifEmpty { listOf("no", "values", "here") })
    // [no, values, here]
    println(full.ifEmpty { listOf("no", "values", "here") })
    // [apple, orange, banana]
}
```

### `ifBlank`: `ifEmpty`'s sibling function for strings

When working with text (which we often also simply treat as a collection of

characters), we sometimes relax the requirement of "empty" to "blank": strings containing only whitespace characters are seldom more expressive than pure empty strings. To generate a default value for them, we can use the `ifBlank` function.

```
fun main() {
    val blankName = " "
    val name = "J. Doe"
    println(blankName.isEmpty { "(unnamed)" })
    //
    println(blankName.isBlank { "(unnamed)" })
    // (unnamed)
    println(name.isBlank { "(unnamed)" })
    // J. Doe
}
```

### 6.1.9 Splitting collections: "chunked" and "windowed"

When the data in your collection represents a series of information, you may want to work with multiple consecutive values at a time. Consider, for example, a list of daily measurements of a temperature sensor:

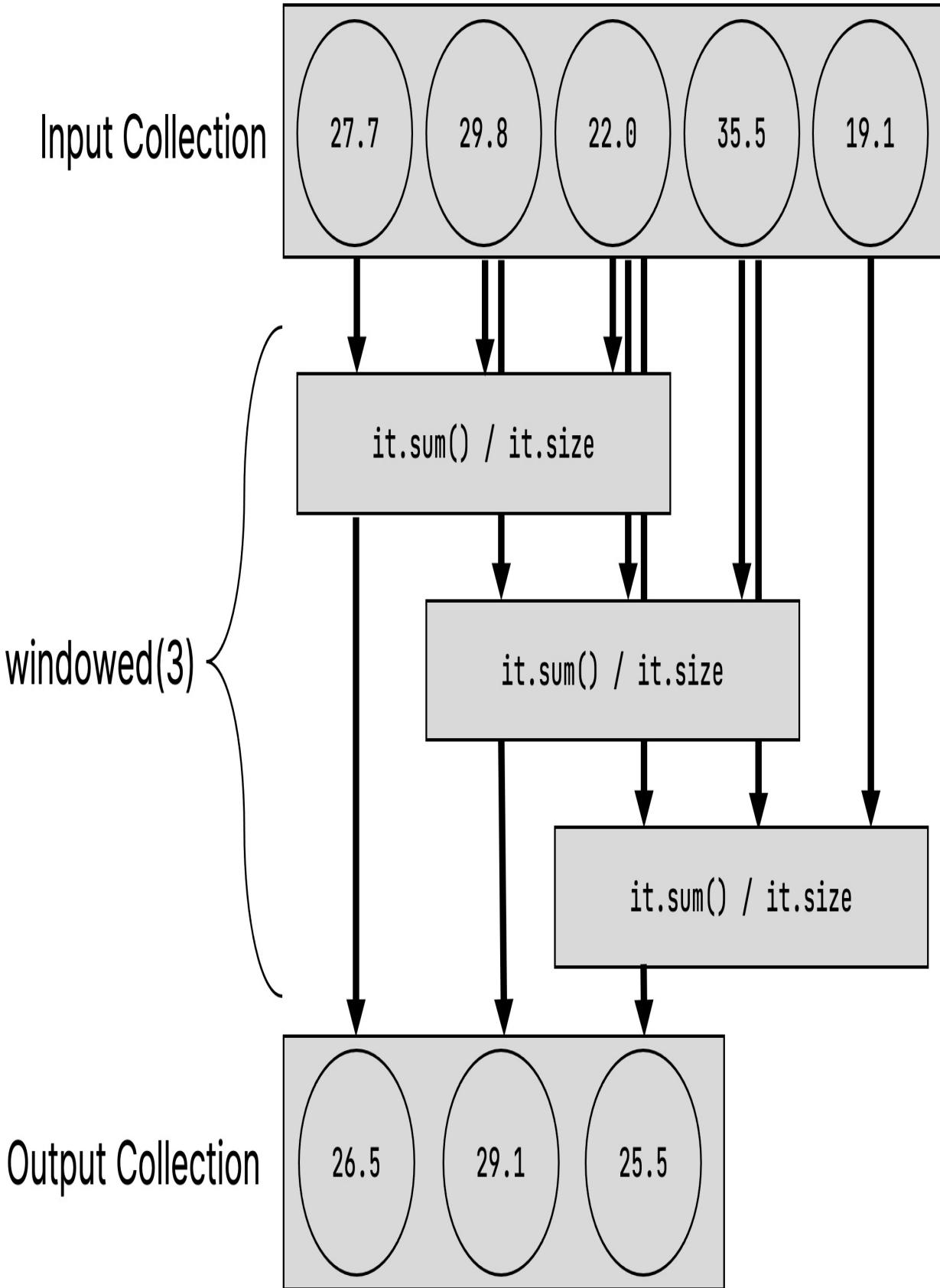
```
val temperatures = listOf(27.7, 29.8, 22.0, 35.5, 19.1)
```

To get a three-day average for each set of days in this list of values, you would use a *sliding window* of size 3: you would first average the first three values: 27.7, 29.8, and 22.0. Then, you would "slide" the window over by one index, averaging 29.8, 22.0, and 35.5. You would keep sliding until you reach the final three values—22.0, 35.5, and 19.1.

To generate these kinds of sliding windows, you can use the `windowed` function. `windowed` optionally lets you pass a lambda that transforms the output. In the case of temperature measurements, that could be calculating the average of each window:

```
fun main() {
    println(temperatures.windowed(3))
    // [[27.7, 29.8, 22.0], [29.8, 22.0, 35.5], [22.0, 35.5, 19.1]
    println(temperatures.windowed(3) { it.sum() / it.size })
    // [26.5, 29.09999999999998, 25.53333333333333]
}
```

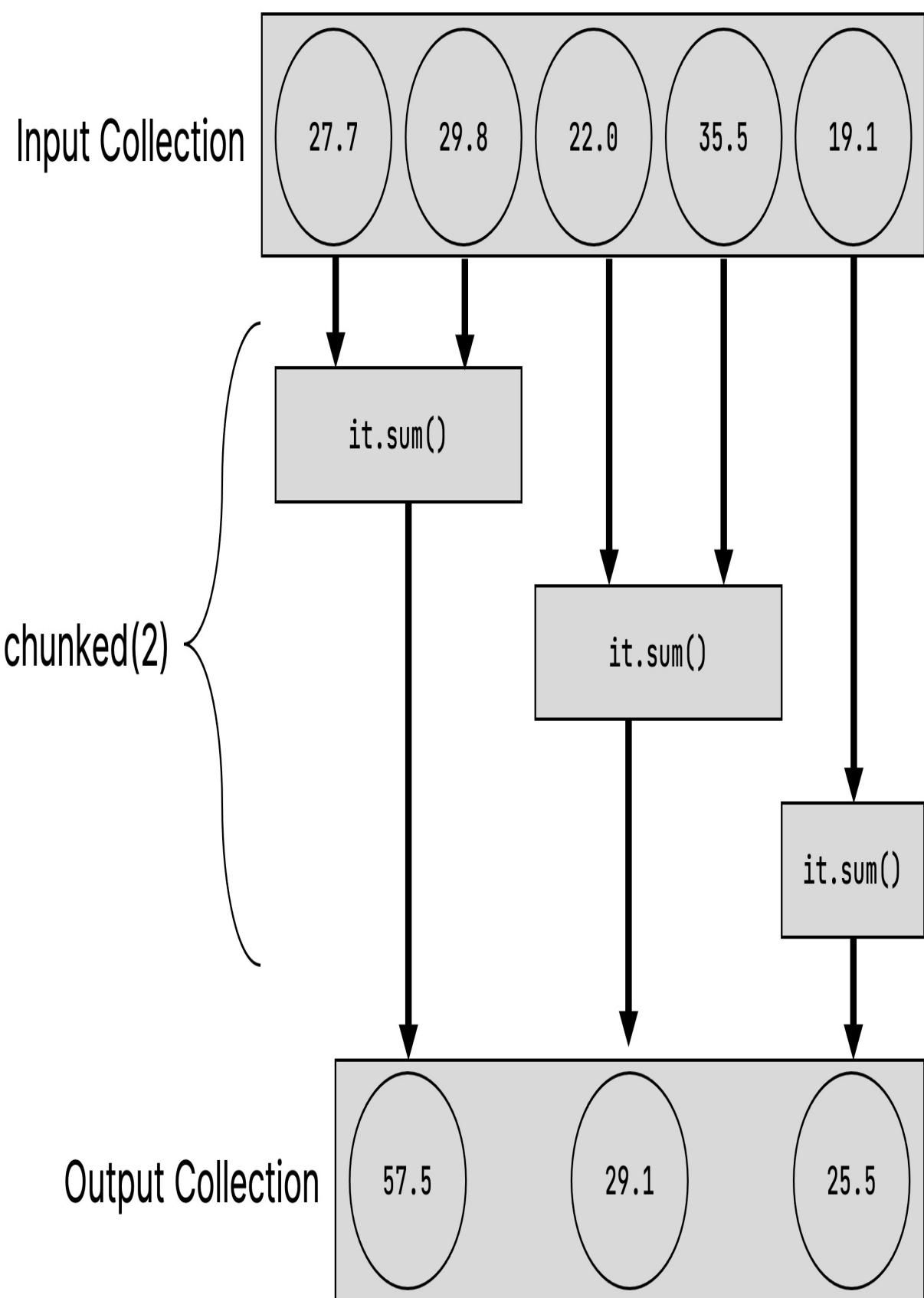
**Figure 6.8.** The `windowed` function processes your input collection using a sliding window.



Instead of running a sliding window over your input collection, you may want to break the collection into distinct parts of a given size. The `chunked` function helps you achieve this. Once again, you can also pass a lambda which transforms the output:

```
fun main() {  
    println(temperatures.chunked(2))  
    // [[27.7, 29.8], [22.0, 35.5], [19.1]]  
    println(temperatures.chunked(2) { it.sum() })  
    // [57.5, 57.5, 19.1]  
}
```

**Figure 6.9.** The `chunked` function processes your input collection in non-overlapping segments of the specified size.



Note that in the example above, even though you specify a chunk size of 2, the last generated chunk may have a smaller size: since the input collection has an odd number of items, the chunked function creates two chunks of size 2, and puts the remaining item in a third chunk.

### 6.1.10 Merging collections: zip

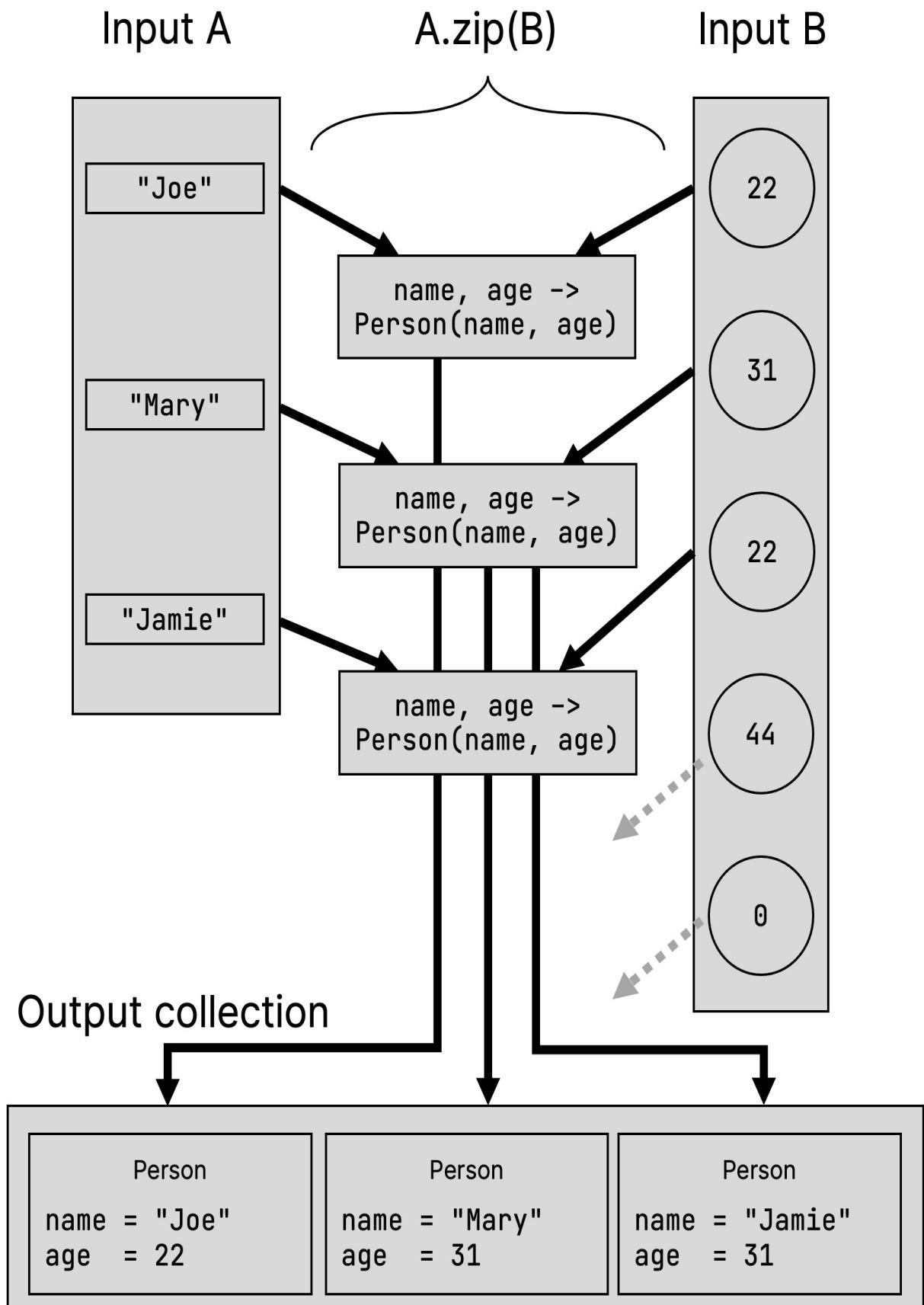
You may encounter situations where your data isn't yet formatted nicely in proper lists of Kotlin objects. Instead, you may have to work with separate lists that contain related information. For example, instead of a list of Person objects, you have two lists for people's names and ages respectively:

```
val names = listOf("Joe", "Mary", "Jamie")
val ages = listOf(22, 31, 22, 44, 0)
```

If you know that the values in each list correspond by their index (that is, the name at index 0, "Joe", corresponds to the age at index 0, 22) you can use the zip function to create a list of pairs from values at same index from two collections. Passing a lambda to the function also allows you to specify how the output should be transformed. In this example, you create a list of Person objects from each pair of name and age:

```
fun main() {
    val names = listOf("Joe", "Mary", "Jamie")
    val ages = listOf(22, 31, 22, 44, 0)
    println(names.zip(ages))
    // [(Joe, 22), (Mary, 31), (Jamie, 22)] #1
    println(names.zip(ages) { name, age -> Person(name, age) })
    // [Person(name=Joe, age=22), Person(name=Mary, age=31),
    // Person(name=Jamie, age=22)]
}
```

**Figure 6.10.** The `zip` function correlates each element of its two inputs at the same index using the lambda you pass to it, or creates pairs of the elements otherwise. Elements that don't have a counterpart in the other collection are ignored.



Note that the size of the resulting collection is the same as the shorter of the two lists: `zip` only processes items at those indexes that exist in both input collections.

Like the `to` function to create `Pair` objects, the `zip` function can also be called as an *infix* function (as introduced in [3.4.3](#))—though you won’t be able to pass a transforming lambda in this case:

```
println(names zip ages)
// [(Joe, 22), (Mary, 31), (Jamie, 22)]
```

Like any other function, you can chain multiple `zip` calls to combine more than two lists. Since `zip` always operates on two lists, though, you’ll need to be aware that your resulting structure will consist of nested pairs, rather than simply a list of lists:

```
val countries = listOf("DE", "NL", "US")
println(names zip ages zip countries)
// [((Joe, 22), DE), ((Mary, 31), NL), ((Jamie, 22), US)] #1
```

### 6.1.11 Processing elements in nested collections: `flatMap` and `flatten`

Now let’s put aside our discussion of people and switch to books. Suppose you have a storage of books, represented by the class `Book`:

```
class Book(val title: String, val authors: List<String>)
```

Each book was written by one or more authors, and you have a number of books in your library:

```
val library = listOf(
    Book("Kotlin in Action", listOf("Isakova", "Elizarov", "Aigner"),
        Book("Atomic Kotlin", listOf("Eckel", "Isakova")),
        Book("The Three-Body Problem", listOf("Liu")))
)
```

If you want to compute all authors in your library, you might start by using the `map` function you’ve gotten to know in [6.1.1](#):

```
fun main() {
    val authors = library.map { it.authors }
    println(authors)
    // [[Isakova, Elizarov, Aigner, Jemerov], [Eckel, Isakova], [
}
```

Likely, this isn't the result you had in mind though: Because `authors` is a `List<String>` in itself, your resulting collection is a `List<List<String>>`—a nested collection.

With the `flatMap` function, you can compute the set of all the authors in your library, without any extra nesting. It does two things: At first it transforms (or *maps*) each element to a collection according to the function given as an argument, and then it combines (or *flattens*) these several lists into one.

**Listing 6.1. The flatMap function turns a collection of collections into a flat list.**

```
fun main() {
    val authors = library.flatMap { it.authors }
    println(authors) #1
    // [Isakova, Elizarov, Aigner, Jemerov, Eckel, Isakova, Liu]
    println(authors.toSet()) #2
    // [Isakova, Elizarov, Aigner, Jemerov, Eckel, Liu]
}
```

Each book can be written by multiple authors, and the `book.authors` property stores a list of authors. In [6.1](#), you use the `flatMap` function to combine the authors of all the books in a single, flat list. The `toSet` call removes duplicates from the resulting collection—so, in this example, Svetlana Isakova is listed only once in the output.

You may think of `flatMap` when you're stuck with a collection of collections of elements that have to be combined into one. Note that if you don't need to transform anything and just need to flatten such a collection of collections, you can use the `flatten` function: `listOfLists.flatten()`.

We've highlighted a few of the collection operation functions in the Kotlin standard library, but there are many more. We won't cover them all, for reasons of space, and also because showing a long list of functions is boring. Our general advice when you write code that works with collections is to

think of how the operation could be expressed as a general transformation, and to look for a standard library function that performs such a transformation—either by looking through your IDE’s autocomplete suggestions, or by consulting the Kotlin standard library reference (<https://kotlinlang.org/api/latest/jvm/stdlib/>). It’s likely that you’ll be able to find one and use it to solve your problem more quickly than with a manual implementation.

Now let’s take a closer look at the performance of code that chains collection operations. In the next section, you’ll see the different ways in which such operations can be executed.

## 6.2 Lazy collection operations: sequences

In the previous section, you saw several examples of chained collection functions, such as `map` and `filter`. These functions create intermediate collections *eagerly*, meaning the intermediate result of each step is stored in a temporary list. *Sequences* give you an alternative way to perform such computations that avoids the creation of intermediate temporary objects, similar to how Java 8’s streams do.

Here’s an example:

```
people.map(Person::name).filter { it.startsWith("A") }
```

The Kotlin standard library reference says that both `filter` and `map` return a list. That means this chain of calls will create two lists: one to hold the results of the `filter` function and another for the results of `map`. This isn’t a problem when the source list contains two elements, but it becomes much less efficient if you have a million.

To make this more efficient, you can convert the operation so it uses *sequences* instead of using collections directly:

```
people
    .asSequence() #1
    .map(Person::name) #2
    .filter { it.startsWith("A") } #2
    .toList() #3
```

The result of applying this operation is the same as in the previous example: a list of people's names that start with the letter A. But in the second example, no intermediate collections to store the elements are created, so performance for a large number of elements will be noticeably better.

The entry point for lazy collection operations in Kotlin is the `Sequence` interface. The interface represents just that: a sequence of elements that can be enumerated one by one. `Sequence` provides only one method, `iterator`, that you can use to obtain the values from the sequence.

The strength of the `Sequence` interface is in the way operations on it are implemented. The elements in a sequence are evaluated *lazily*. Therefore, you can use sequences to efficiently perform chains of operations on elements of a collection without creating collections to hold intermediate results of the processing. You'll also notice that we can use functions that we already know from regular list processing, such as `map` and `filter`, with sequences as well.

Any collection can be converted to a sequence by calling the extension function `asSequence`. To do the opposite conversion, from a sequence to a plain list, you call `toList`.

Why do you need to convert the sequence back to a collection? Wouldn't it be more convenient to use sequences instead of collections, if they're so much better? The answer is: sometimes. If you only need to iterate over the elements in a sequence, you can use the sequence directly. If you need to use other API methods, such as accessing the elements by index, then you need to convert the sequence to a list.



#### Note

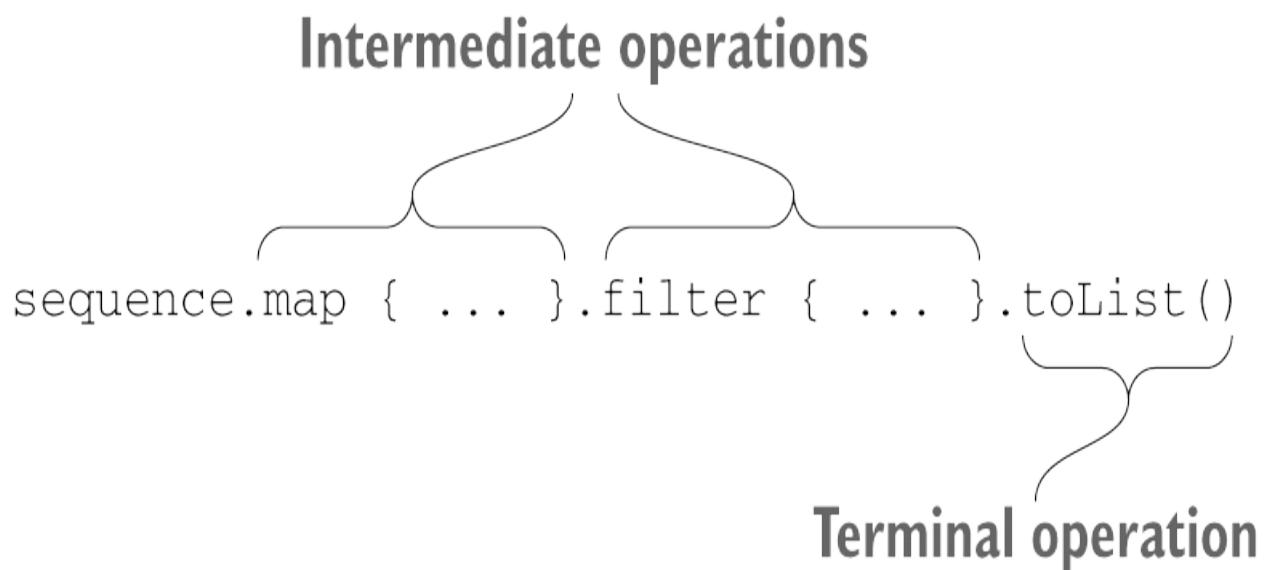
As a rule, use a sequence whenever you have a chain of operations on a *large* collection. In **Chapter 10**, we'll discuss why eager operations on regular collections are efficient in Kotlin, in spite of creating intermediate collections. But if the collection contains a large number of elements, the intermediate rearranging of elements costs a lot, so lazy evaluation is preferable.

Because operations on a sequence are lazy, in order to perform them, you need to iterate over the sequence's elements directly or by converting it to a collection. The next section explains that.

### 6.2.1 Executing sequence operations: intermediate and terminal operations

Operations on a sequence are divided into two categories: intermediate and terminal. An *intermediate operation* returns another sequence, which knows how to transform the elements of the original sequence. A *terminal operation* returns a result, which may be a collection, an element, a number, or any other object that's somehow obtained by the sequence of transformations of the initial collection (see [6.11](#)).

Figure 6.11. Intermediate and terminal operations on sequences



Intermediate operations are always lazy. Look at this example, where the terminal operation is missing:

```
fun main() {
    println(
        listOf(1, 2, 3, 4)
            .asSequence()
            .map {
                print("map($it) ")
            }
    )
}
```

```

        it * it
    }.filter {
        print("filter($it) ")
        it % 2 == 0
    }
}
// kotlin.sequences.FilteringSequence@506e1b77
}

```

Executing this code snippet doesn't print the expected result to the console—rather than seeing the result of these operations, you only see the Sequence object itself. The execution of the `map` and `filter` transformations are postponed and will be applied only when the result is obtained (that is, when the terminal operation is called):

```

fun main() {
    listOf(1, 2, 3, 4)
        .asSequence()
        .map {
            print("map($it) ")
            it * it
        }.filter {
            print("filter($it) ")
            it % 2 == 0
        }.toList()
}

```

The terminal operation causes all the postponed computations to be performed.

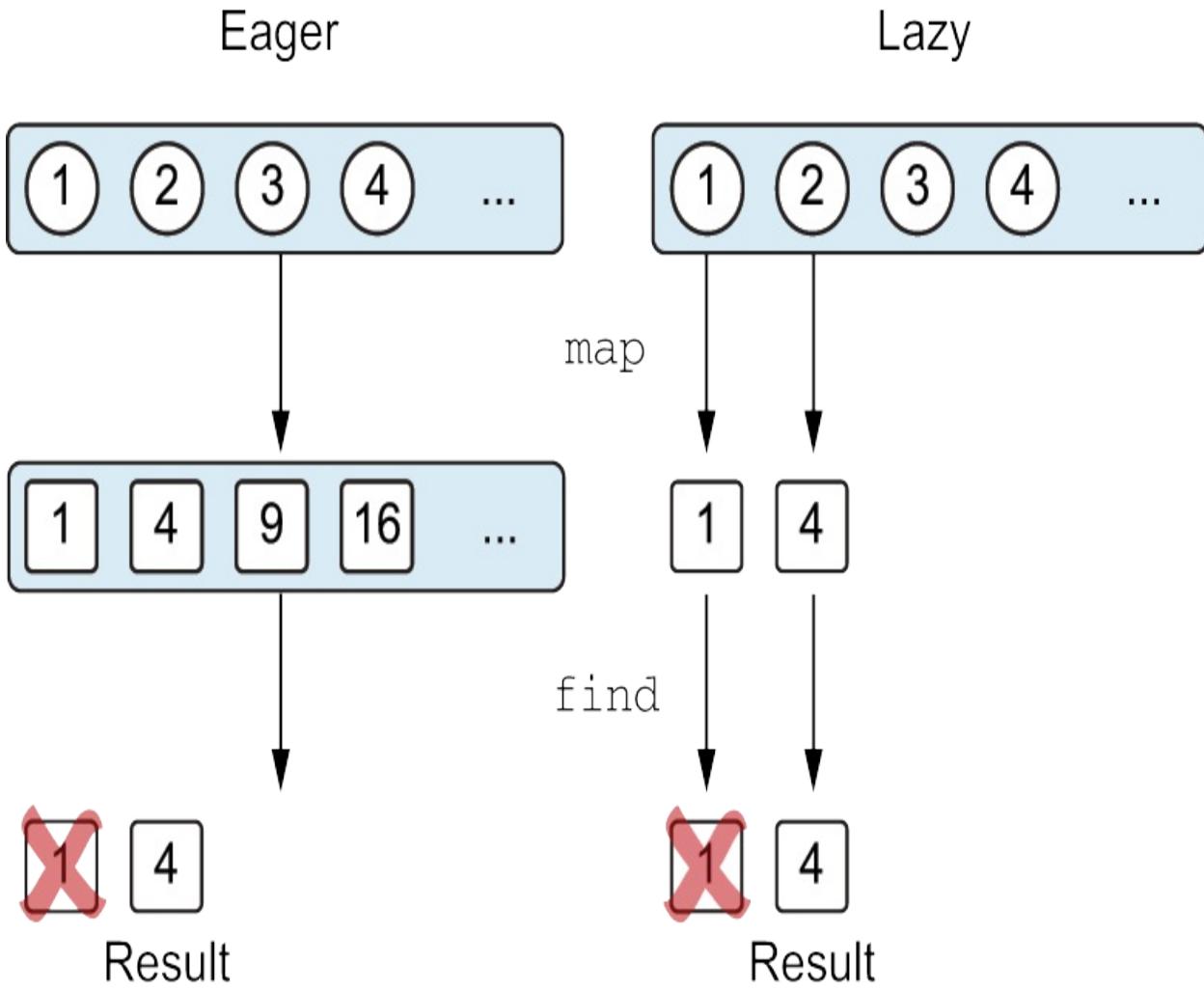
One more important thing to notice in this example is the order in which the computations are performed. The naive approach would be to call the `map` function on each element first and then call the `filter` function on each element of the resulting sequence. That's how `map` and `filter` work on collections, but not on sequences. For sequences, all operations are applied to each element sequentially: the first element is processed (mapped, then filtered), then the second element is processed, and so on.

This approach means some elements aren't transformed at all if the result is obtained before they are reached. Let's look at an example with `map` and `find` operations. First you map a number to its square, and then you find the first item that's greater than 3:

```
fun main() {
    println(
        listOf(1, 2, 3, 4)
            .asSequence()
            .map { it * it }
            .find { it > 3 }
    )
    // 4
}
```

If the same operations are applied to a collection instead of a sequence, then the result of `map` is evaluated first, transforming all elements in the initial collection. In the second step, an element satisfying the predicate is found in the intermediate collection. With sequences, the lazy approach means you can skip processing some of the elements. [6.12](#) illustrates the difference between evaluating this code in an eager (using collections) and lazy (using sequences) manner.

**Figure 6.12. Eager evaluation runs each operation on the entire collection; lazy evaluation processes elements one by one.**



In the first case, when you work with collections, the list is transformed into another list, so the `map` transformation is applied to each element, including 3 and 4. Afterward, the first element satisfying the predicate is found: the square of 2.

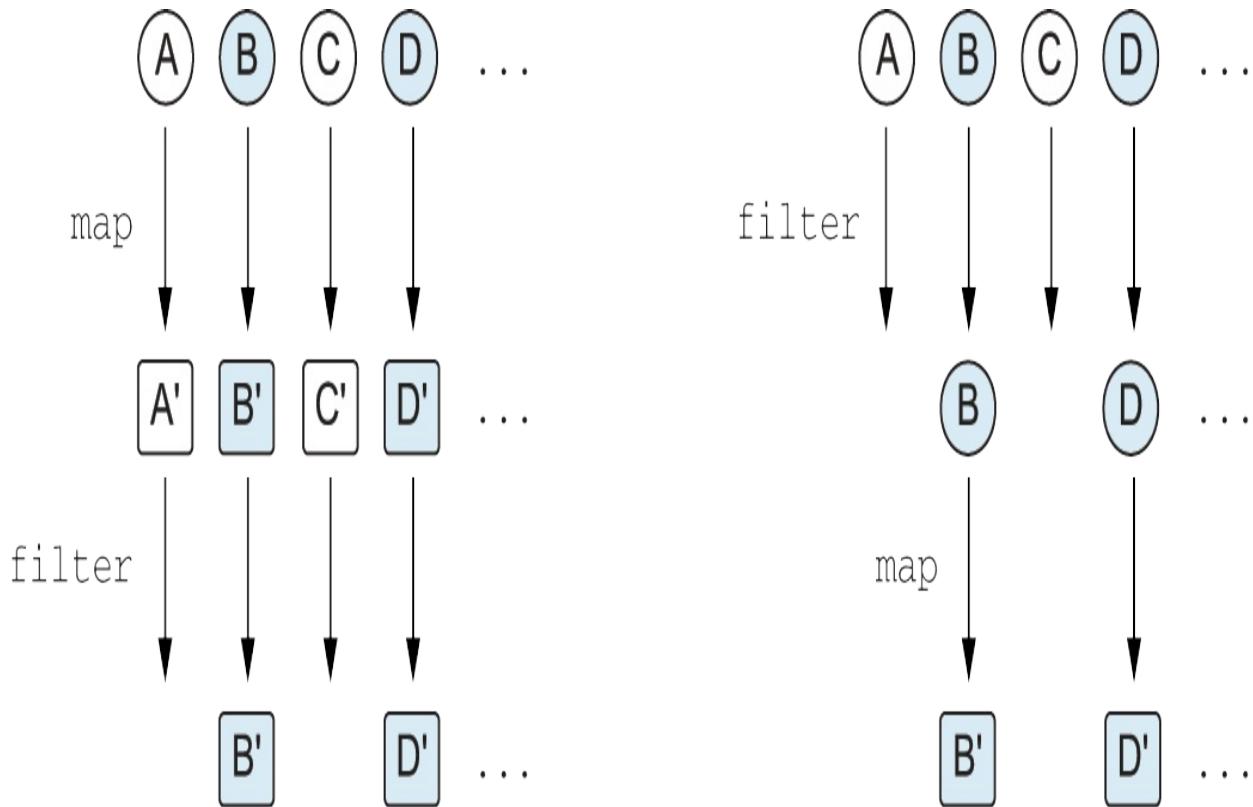
In the second case, the `find` call begins processing the elements one by one. You take a number from the original sequence, transform it with `map`, and then check whether it matches the predicate passed to `find`. When you reach 2, you see that its square is greater than 3 and return it as the result of the `find` operation. You don't need to look at 3 and 4, because the result was found before you reached them.

The order of the operations you perform on a collection can affect performance as well. Imagine that you have a collection of people, and you

want to print their names if they're shorter than a certain limit. You need to do two things: map each person to their name, and then filter out those names that aren't short enough. You can apply `map` and `filter` operations in any order in this case. Both approaches give the same result, but they differ in the total number of transformations that should be performed (see [6.13](#)).

```
fun main() {
    val people = listOf(
        Person("Alice", 29),
        Person("Bob", 31),
        Person("Charles", 31),
        Person("Dan", 21)
    )
    println(
        people
            .asSequence()
            .map(Person::name) #1
            .filter { it.length < 4 }.toList()
    )
    // [Bob, Dan]
    println(
        people
            .asSequence()
            .filter { it.name.length < 4 }
            .map(Person::name).toList() #2
    )
    // [Bob, Dan]
}
```

**Figure 6.13.** Applying `filter` first helps to reduce the total number of transformations.



If `map` goes first, each element is transformed, even if it is discarded in the next step, and never used again. If you apply `filter` first, inappropriate elements are filtered out as soon as possible and aren't transformed. As a rule of thumb, the earlier you can remove elements from your chain of operations (without compromising the logic of your code, of course), the more performant your code will be.

## 6.2.2 Creating sequences

The previous examples used the same method to create a sequence: you called `asSequence()` on a collection. Another possibility is to use the `generateSequence` function. This function calculates the next element in a sequence given the previous one. For example, here's how you can use `generateSequence` to calculate the sum of all natural numbers up to 100. You first generate a sequence of all natural numbers. You then use the `takewhile` function to take elements from that sequence while they're less than or equal to 100. Lastly, you use `sum` to calculate the sum of these numbers.

**Listing 6.2. Generating and using a sequence of natural numbers**

```

fun main() {
    val naturalNumbers = generateSequence(0) { it + 1 }
    val numbersTo100 = naturalNumbers.takeWhile { it <= 100 }
    println(numbersTo100.sum()) #1
    // 5050
}

```

Note that `naturalNumbers` (an infinite sequence) and `numbersTo100` (a finite sequence) in this example are both sequences with postponed computation. The actual numbers in those sequences won't be evaluated until you call the terminal operation (`sum` in this case).

Another common use case is a sequence of parents. If an element has parents of its own type, you may be interested in qualities of the sequence of all of its ancestors. Typical examples for this might be the lineage for a human being, or the parent folder structure for a given file (on the JVM, both files and folders are typically represented by the same type—`File`).

In the following example, you inquire whether the file is located in a hidden directory by generating a sequence of its parent directories and checking this attribute on each of the directories.

#### **Listing 6.3. Generating and using a sequence of parent directories**

```

import java.io.File

fun File.isInsideHiddenDirectory() =
    generateSequence(this) { it.parentFile }.any { it.isHidden }

fun main() {
    val file = File("/Users/svtk/.HiddenDir/a.txt")
    println(file.isInsideHiddenDirectory())
    // true
}

```

Once again, you generate a sequence by providing the first element and a way to get each subsequent element. By replacing `any` with `find`, you'll get the actual directory that is hidden, instead of just a Boolean value indicating that there is a hidden file somewhere in the path. Note that using sequences allows you to stop traversing the parents as soon as you find the required directory.

## 6.3 Summary

- Instead of manually iterating over elements in a collection, most common operations can be performed by combining existing functions from the standard library with your own lambdas. Kotlin comes with a wide variety of such functions.
- The `filter` and `map` functions form the basis for manipulating collections, and make it easy to extract elements that match a certain predicate or transform elements into a new form.
- The `reduce` and `fold` operations *aggregate* information from a collection, helping you compute a single value given a collection of items.
- Functions from the `associate` and `groupBy` families help you turn flat lists into maps, so you can structure your data by your own criteria.
- For data in collections that is related by its indices, the `chunked`, `windowed`, and `zip` functions make it possible to create subgroups of collection elements or merge together multiple collections.
- Using *predicates*, lambda functions that return Boolean, the `all`, `any`, `none`, and other sibling functions allow you to check whether certain invariants apply to your collections.
- To deal with nested collections, the `flatten` function can help you extract the nested items, while the `flatMap` function makes it possible to even perform a transformation in the same step.
- Sequences allow you to combine multiple operations on a collection *lazily* and without creating collections to hold intermediate results, making your code more efficient. You can use the same functions you use for collections to manipulate sequences.

# 7 Working with nullable values

## This chapter covers

- Nullable types
- Syntax for dealing with values that are potentially `null`
- Converting between nullable and non-nullable types
- Interoperability between Kotlin's concept of nullability and Java code

By now, you've seen a large part of Kotlin's syntax in action. You've moved beyond creating basic code in Kotlin and are ready to enjoy some of Kotlin's productivity features that can make your code more compact and readable. One of the essential features in Kotlin that helps improve the reliability of your code is its support for *nullable types*. Let's look at the details.

## 7.1 Avoiding NullPointerExceptions and handling the absence of values: Nullability

*Nullability* is a feature of the Kotlin type system that helps you avoid `NullPointerException` errors. As a user of a program, you've probably seen an error message similar to "An error has occurred:

`java.lang.NullPointerException,`" with no additional details. On Android, you may have seen another version of this message along the lines of "Unfortunately, the application X has stopped," which often also conceals a `NullPointerException` as a cause. Such errors occurring at runtime are troublesome for both users and developers.

The approach of modern languages, including Kotlin, is to convert these problems from runtime errors into compile-time errors. By supporting nullability as part of the type system, the compiler can detect many possible errors during compilation and reduce the possibility of having exceptions thrown at runtime.

In this section, we'll discuss nullable types in Kotlin: how Kotlin marks

values that are allowed to be `null`, and the tools Kotlin provides to deal with such values. Moving beyond that, we'll cover the details of mixing Kotlin and Java code with respect to nullable types.

### 7.1.1 Making possibly null variables explicit with nullable types

The first and probably most important difference between Kotlin's and Java's type systems is Kotlin's explicit support for *nullable types*. What does this mean? It's a way to indicate which variables or properties in your program are allowed to be `null`. If a variable can be `null`, calling a method on it isn't safe, because it can cause a `NullPointerException`. Kotlin disallows such calls and thereby prevents many possible exceptions. To see how this works in practice, let's look at the following Java function, which accepts a `String` and calls the `length` function on it:

```
/* Java */
int strLen(String s) {
    return s.length();
}
```

Is this function safe? Well, a seasoned developer would probably quickly spot that if the function is called with a `null` argument, it will throw a `NullPointerException`. Do you need to add a check for `null` to the function? It depends on the function's intended use.

Let's rewrite this function in Kotlin. The first question you must answer is, do you expect the function to be called with a `null` argument? We mean not only the `null` literal directly, as in `strLen(null)`, but also any variable or other expression that may have the value `null` at runtime.

If you don't expect it to happen, you declare this function in Kotlin as follows:

```
fun strLen(s: String) = s.length
```

Calling `strLen` with an argument that may be `null` isn't allowed and will be flagged as error at compile time:

```
fun main() {
```

```
    strLen(null)
    // ERROR: Null can not be a value of a non-null type String
}
```

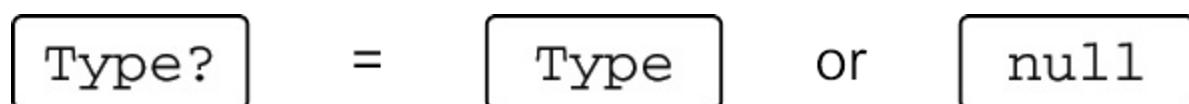
The parameter is declared as type `String`, and in Kotlin this means it must always contain a `String` instance. The compiler enforces that, so you can't pass an argument containing `null`. This gives you the guarantee that the `strLen` function will never throw a `NullPointerException` at runtime.

If you want to allow the use of this function with all arguments, including those that can be `null`, you need to mark it explicitly by putting a question mark after the type name:

```
fun strLenSafe(s: String?) = ...
```

You can put a question mark after any type, to indicate that the variables of this type can store `null` references: `String?`, `Int?`, `MyCustomType?`, and so on (see [7.1](#)).

**Figure 7.1. The question mark after a type name indicates that it is nullable. A variable of nullable type can store a null reference.**



To reiterate, a type without a question mark denotes that variables of this type can't store `null` references. This means all regular types are non-`null` by default, unless explicitly marked as nullable.

Once you have a value of a nullable type, the set of operations you can perform on it is restricted. For example, you can no longer call methods on it. The compiler will now complain about the call to `length` in the function body:

```
fun strLenSafe(s: String?) = s.length()
// ERROR: only safe (?..) or non-null asserted (!!..) calls are allowed
// on a nullable receiver of type kotlin.String?
```

You also can't assign a value of nullable type to a variable of a non-null type:

```
fun main() {  
    val x: String? = null  
    var y: String = x  
    //ERROR: Type mismatch: inferred type is String? but String was  
}
```

You can't pass a value of a nullable type as an argument to a function having a non-null parameter:

```
fun main() {  
    val x: String? = null  
    strLen(x)  
    //ERROR: Type mismatch: inferred type is String? but String was  
}
```

So what can you do with a value of nullable type? The most important thing is to compare it with `null`. And once you perform the comparison, the compiler remembers that and treats the value as being non-null in the scope where the check has been performed. For example, this code is perfectly valid.

#### **Listing 7.1. Handling `null` values using `if` checks**

```
fun strLenSafe(s: String?): Int =  
    if (s != null) s.length else 0 #1  
  
fun main() {  
    val x: String? = null  
    println(strLenSafe(x))  
    // 0  
    println(strLenSafe("abc"))  
    // 3  
}
```

If using `if` checks was the only tool for tackling nullability, your code would become verbose fairly quickly. Fortunately, Kotlin provides a number of other tools to help deal with nullable values in a more concise manner. But before we look at those tools, let's spend some time discussing the meaning of nullability and what variable types are.

### **7.1.2 Taking a closer look at the meaning of types**

Let's think about the most general questions: what are types, and why do variables have them? The Wikipedia article on types ([http://en.wikipedia.org/wiki/Data\\_type](http://en.wikipedia.org/wiki/Data_type)) gives a pretty good answer to what a type is: "A type is a classification ... that determines the possible values for that type, and the operations that can be done on values of that type."

Let's try to apply this definition to some of the Java types, starting with the `double` type. As you know, a `double` is a 64-bit floating-point number. You can perform standard mathematical operations on these values. All of those functions are equally applicable to all values of type `double`. Therefore, if you have a variable of type `double`, then you can be certain that any operation on its value that's allowed by the compiler will execute successfully.

Now let's contrast this with a variable of type `String`. In Java, such a variable can hold one of two kinds of values: an instance of the class `String` or `null`. Those kinds of values are completely unlike each other: even Java's own `instanceof` operator will tell you that `null` isn't a `String`. The operations that can be done on the value of the variable are also completely different: an actual `String` instance allows you to call any methods on the string, whereas a `null` value allows only a limited set of operations.

This means Java's type system isn't doing a good job in this case. Even though the variable has a declared type—`String`—you don't know what you can do with values of this variable unless you perform additional checks. Often, you skip those checks because you know from the general flow of data in your program that a value can't be `null` at a certain point. Sometimes you're wrong, and your program then crashes with a `NullPointerException`.

### **Other ways to cope with `NullPointerException` errors**

Java has some tools to help solve the problem of `NullPointerException`. For example, some people use annotations (such as `@Nullable` and `@NotNull`) to express the nullability of values. There are tools (for example, IntelliJ IDEA's built-in code inspections) that can use these annotations to detect places where a `NullPointerException` can be thrown. But such tools aren't part of the standard Java compilation process, so it's hard to ensure that they're applied consistently. It's also difficult to annotate the entire codebase,

including the libraries used by the project, so that all possible error locations can be detected. Our own experience at JetBrains shows that even widespread use of nullability annotations in Java doesn't completely solve the problem of NPEs.

Another path to solving this problem is to never use `null` values in code and to use a special wrapper type, such as the `Optional` type introduced in Java 8, to represent values that may or may not be defined. This approach has several downsides: the code gets more verbose, the extra wrapper instances affect performance at runtime, and it's not used consistently across the entire ecosystem. Even if you do use `Optional` everywhere in your own code, you'll still need to deal with `null` values returned from methods of the JDK, the Android framework, and other third-party libraries.

Nullable types in Kotlin provide a comprehensive solution to this problem. Distinguishing nullable and non-`null` types provides a clear understanding of what operations are allowed on the value and what operations can lead to exceptions at runtime and are therefore forbidden.



#### Note

Objects of nullable or non-`null` types at runtime are the same; a nullable type isn't a wrapper for a non-`null` type. Besides some automatically generated intrinsic checks ([https://en.wikipedia.org/wiki/Intrinsic\\_function](https://en.wikipedia.org/wiki/Intrinsic_function)) which have very little performance impact, working with nullable types in Kotlin has essentially no runtime overhead.

Now let's see how to work with nullable types in Kotlin and why dealing with them is by no means annoying. We'll start with the special operator for safely accessing a nullable value.

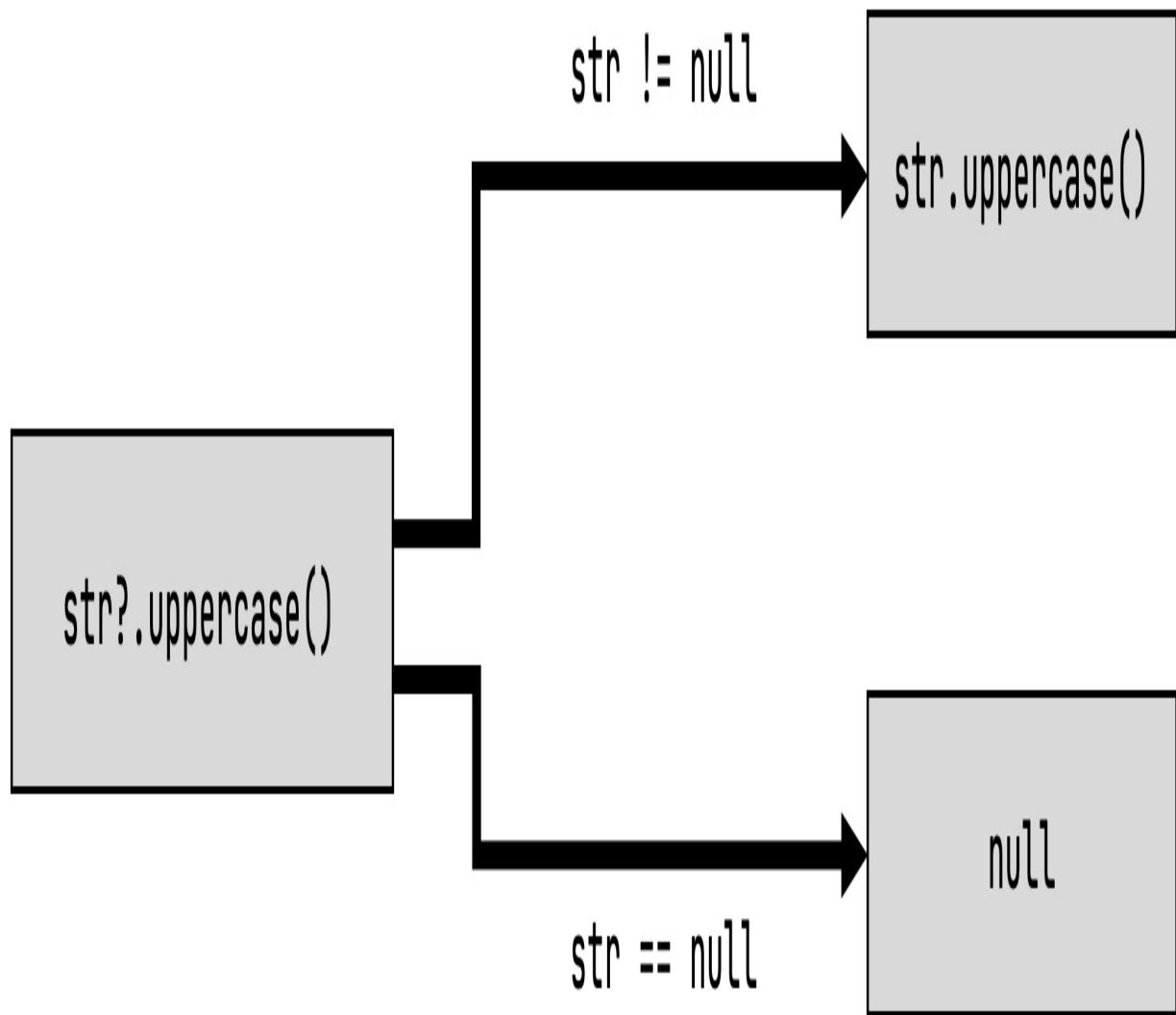
### 7.1.3 Combining null-checks and method calls with the safe call operator: "?."

One of the most useful tools in Kotlin's arsenal is the *safe-call* operator: `?.`, which allows you to combine a `null` check and a method call into a single

operation. For example, the expression `str?.uppercase()` is equivalent to the following, more cumbersome one: `if (str != null) str.uppercase() else null`.

In other words, if the value on which you're trying to call the method isn't `null`, the method call is executed normally. If it's `null`, the call is skipped, and `null` is used as the value instead. [7.2](#) illustrates this.

**Figure 7.2. The safe-call operator calls methods only on non-null values. If the value happens to be null, no call is made, and null is returned directly. This allows you to safely call methods without having to write a null check by hand.**



Note that the result type of such an invocation is nullable. Although `String.uppercase` returns a value of type `String`, the result type of an expression `s?.uppercase()` when `s` is nullable will be `String?`:

```

fun printAllCaps(str: String?) {
    val allCaps: String? = str?.uppercase() #1
    println(allCaps)
}

fun main() {
    printAllCaps("abc")
}

```

```
// ABC
printAllCaps(null)
// null
}
```

Safe calls can be used for accessing properties as well, not just for method calls. The following example shows a simple Kotlin class `Employee` with a nullable property `manager` and demonstrates the use of a safe-call operator for accessing that property in the `managerName` function.

**Listing 7.2. Using safe calls to deal with nullable properties**

```
class Employee(val name: String, val manager: Employee?)

fun managerName(employee: Employee): String? = employee.manager?.

fun main() {
    val ceo = Employee("Da Boss", null)
    val developer = Employee("Bob Smith", ceo)
    println(managerName(developer))
    // Da Boss
    println(managerName(ceo))
    // null
}
```

If you have an object graph in which multiple properties have nullable types, it's often convenient to use multiple safe calls in the same expression. Say you store information about a person, their company, and the address of the company using different classes. Both the company and its address may be omitted. With the `?.` operator, you can access the `country` property for a `Person` in one line, without any additional checks.

**Listing 7.3. Chaining multiple safe-call operators**

```
class Address(val streetAddress: String, val zipCode: Int,
             val city: String, val country: String)

class Company(val name: String, val address: Address?)

class Person(val name: String, val company: Company?)

fun Person.countryName(): String {
    val country = this.company?.address?.country #1
```

```
    return if (country != null) country else "Unknown"
}

fun main() {
    val person = Person("Dmitry", null)
    println(person.countryName())
    // Unknown
}
```

Sequences of calls with `null` checks are a common sight in Java code, and you've now seen how Kotlin makes them more concise. But [7.3](#) contains unnecessary repetition: you're comparing a value to `null` and returning either that value or something else if it's `null`. Let's see how Kotlin can help get rid of that repetition.

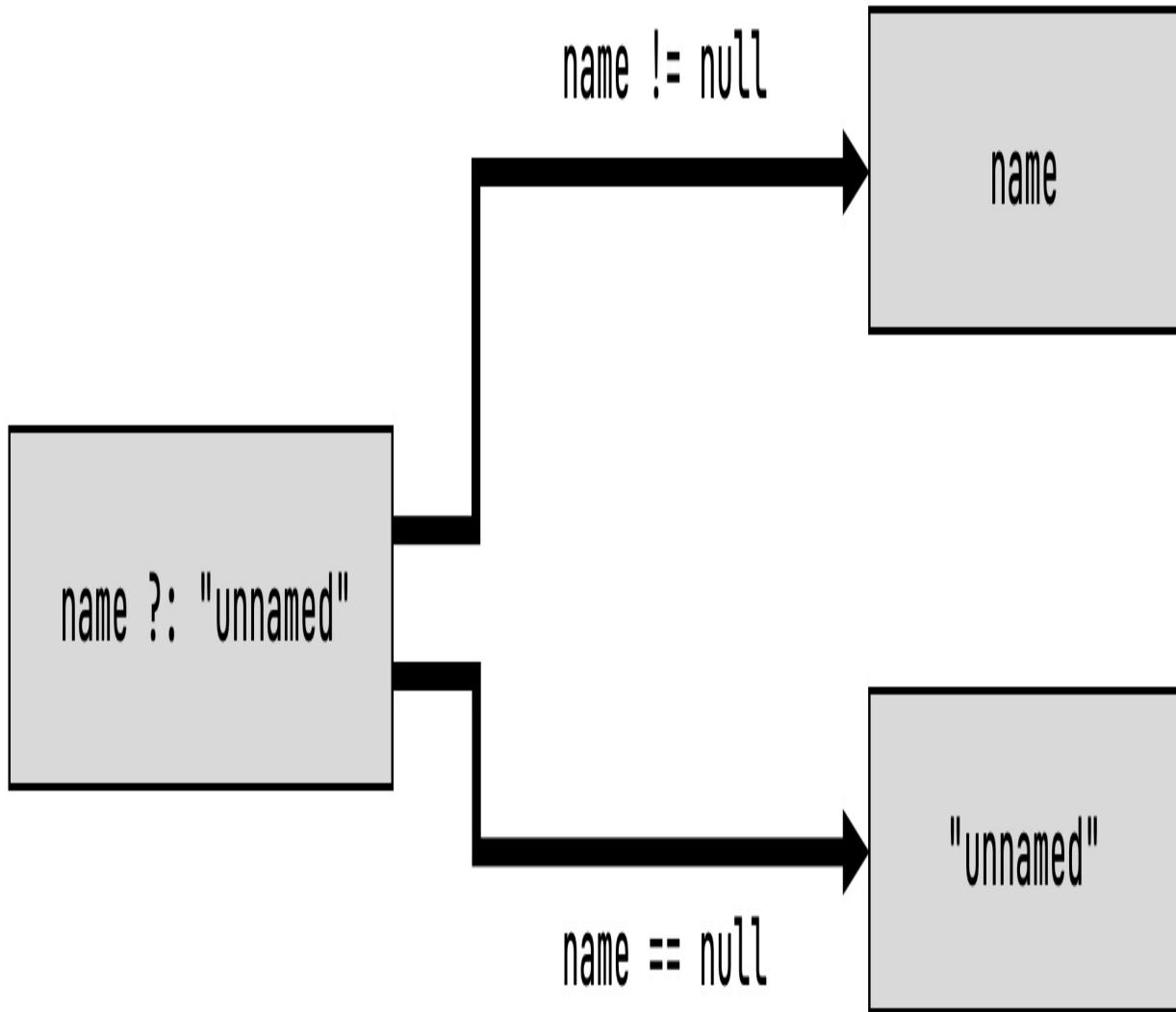
### 7.1.4 Providing default values in null-cases with the Elvis operator: "?:"

Kotlin has a handy operator to provide default values instead of `null`. It's called the *Elvis operator* (or the *null-coalescing operator*, if you prefer more serious-sounding names for things). It looks like this: `?:` (you can visualize it being Elvis if you turn your head sideways). Here's how it's used:

```
fun greet(name: String?) {
    val recipient: String = name ?: "unnamed" #1
    println("Hello, $recipient!")
}
```

The operator takes two values, and its result is the first value if it isn't `null` or the second value if the first one is `null`. [7.3](#) shows how it works.

**Figure 7.3.** The Elvis operator substitutes a specified value for `null`. This allows you to provide a default value should the left-hand expression happen to be `null`.



The Elvis operator is often used together with the safe-call operator to substitute a value other than `null` when the object on which the method is called is `null`. Here's how you can use this pattern to simplify [7.1](#).

#### **Listing 7.4. Using the Elvis operator to deal with `null` values**

```
fun strLenSafe(s: String?): Int = s?.length ?: 0

fun main() {
    println(strLenSafe("abc"))
    // 3
    println(strLenSafe(null))
    // 0
```

```
}
```

The implementation of the `countryName` function from [7.3](#) now also fits in a single, elegant expression.

```
fun Person.countryName() = company?.address?.country ?: "Unknown"
```

What makes the Elvis operator particularly handy in Kotlin is that operations such as `return` and `throw` work as expressions and therefore can be used on the operator's right side. In that case, if the value on the left side is `null`, the function will immediately return a value or throw the exception you specified. This is helpful for checking preconditions in a function.

Let's see how you can use this operator to implement a function to print a shipping label with the person's company address. [7.5](#) repeats the declarations of all the classes—in Kotlin, they're so concise that it's not a problem.

#### **Listing 7.5. Using `throw` together with the Elvis operator**

```
class Address(val streetAddress: String, val zipCode: Int,  
             val city: String, val country: String)  
  
class Company(val name: String, val address: Address?)  
  
class Person(val name: String, val company: Company?)  
  
fun printShippingLabel(person: Person) {  
    val address = person.company?.address  
    ?: throw IllegalArgumentException("No address") #1  
    with (address) { #2  
        println(streetAddress)  
        println("$zipCode $city, $country")  
    }  
}  
  
fun main() {  
    val address = Address("Elsestr. 47", 80687, "Munich", "German  
    val jetbrains = Company("JetBrains", address)  
    val person = Person("Dmitry", jetbrains)  
    printShippingLabel(person)  
    // Elsestr. 47  
    // 80687 Munich, Germany
```

```
    printShippingLabel(Person("Alexey", null))
    // java.lang.IllegalArgumentException: No address
}
```

The function `printShippingLabel` prints a label if everything is correct. If there's no address, it doesn't just throw a `NullPointerException` with a line number, but instead reports a meaningful error. If an address is present, the label consists of the street address, the ZIP code, the city, and the country. Note how the `with` function, which you saw in [5.4.1](#), is used to avoid repeating `address` four times in a row.

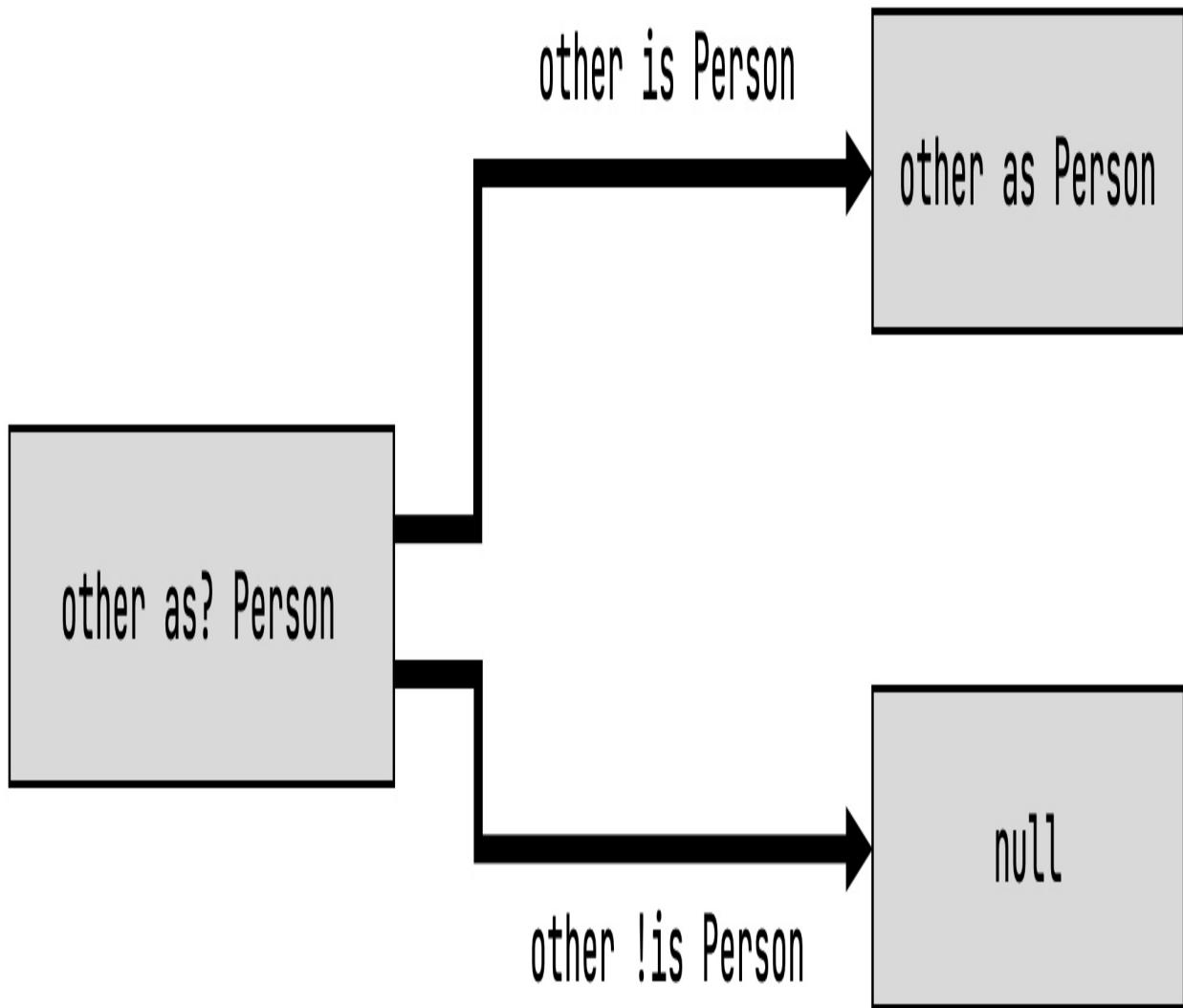
Now that you've seen the Kotlin way to perform "if not-null" checks, let's talk about the Kotlin safe version of `instanceof` checks: the *safe-cast operator* that often appears together with safe calls and Elvis operators.

### 7.1.5 Safely casting values without throwing exceptions: "as?"

In chapter 2, you saw the regular Kotlin operator for type casts: the `as` operator. Just like a regular Java type cast, `as` throws a `ClassCastException` if the value doesn't have the type you're trying to cast it to. Of course, you can combine it with an `is` check to ensure that it does have the proper type. But as a safe and concise language, doesn't Kotlin provide a better solution? Indeed it does.

The `as?` operator tries to cast a value to the specified type and returns `null` if the value doesn't have the proper type. [7.4](#) illustrates this.

**Figure 7.4. The safe-cast operator `as?` gives you the tools to safely work with the possibility that a cast may not succeed. It tries to cast a value to the given type and returns `null` if the type differs.**



One common pattern of using a safe cast is combining it with the Elvis operator. For example, this comes in handy for implementing the `equals` method.

#### **Listing 7.6. Using a safe cast to implement equals**

```
class Person(val firstName: String, val lastName: String) {
    override fun equals(other: Any?): Boolean {
        val otherPerson = other as? Person ?: return false #1

        return otherPerson.firstName == firstName && #2
                otherPerson.lastName == lastName
    }
}
```

```

        override fun hashCode(): Int =
            firstName.hashCode() * 37 + lastName.hashCode()
    }

fun main() {
    val p1 = Person("Dmitry", "Jemerov")
    val p2 = Person("Dmitry", "Jemerov")
    println(p1 == p2) #3
    // true
    println(p1.equals(42))
    // false
}

```

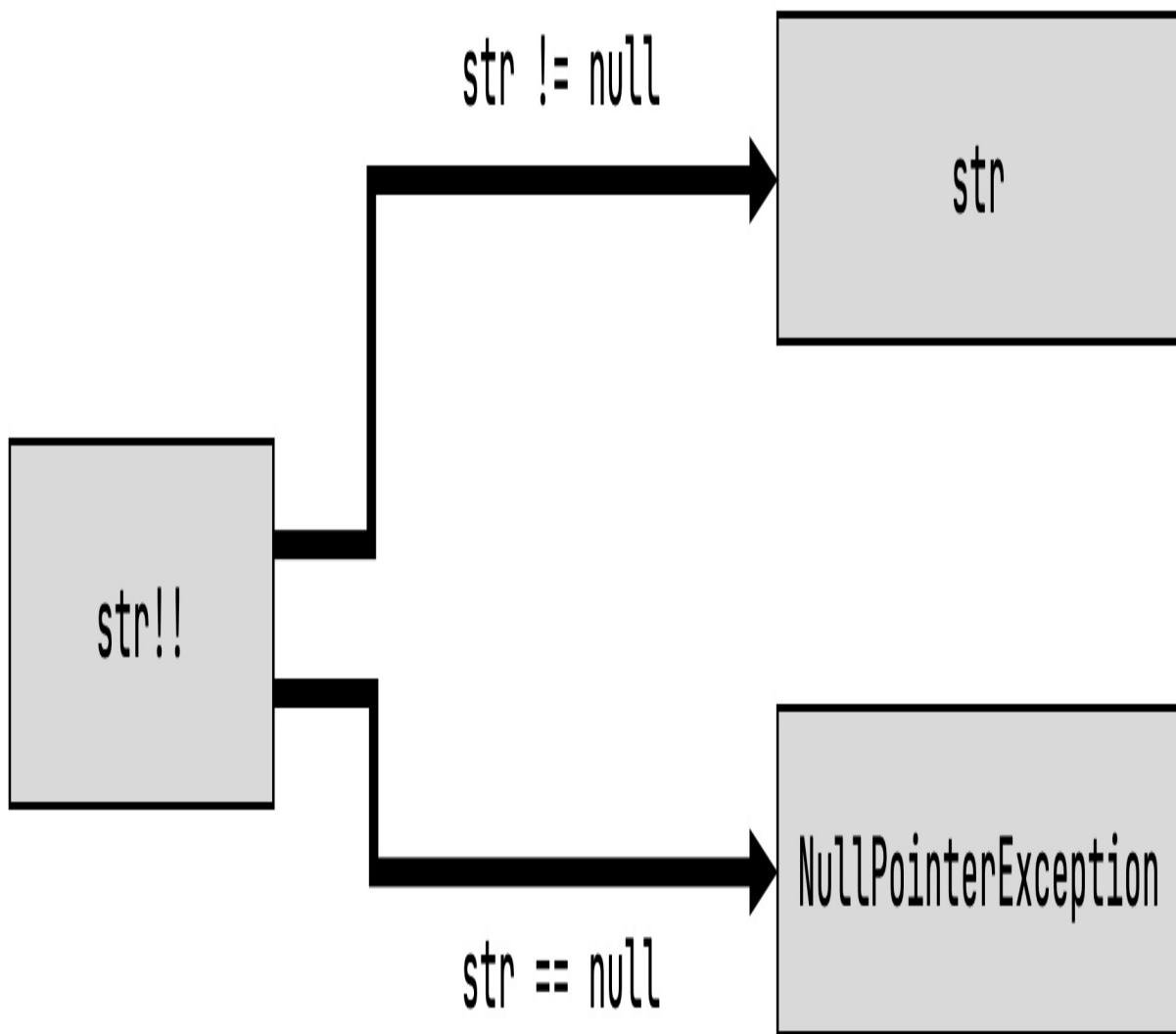
With this pattern, you can easily check whether the parameter has a proper type, cast it, and return `false` if the type isn't right—all in the same expression. Of course, smart casts also apply in this context: after you've checked the type and rejected `null` values, the compiler knows that the type of the `otherPerson` variable's value is `Person` and lets you use it accordingly.

The safe-call, safe-cast, and Elvis operators are useful and appear often in Kotlin code. But sometimes you don't need Kotlin's support in handling `null` values; you just need to tell the compiler that the value is in fact not `null`. Let's see how you can achieve that.

### 7.1.6 Making promises to the compiler with the not-null assertion operator: "!!"

The *not-null assertion* is the simplest and bluntest tool Kotlin gives you for dealing with a value of a nullable type. It's represented by a double exclamation mark and converts any value to a non-null type. For `null` values, an exception is thrown. The logic is illustrated in [7.5](#).

**Figure 7.5. By using a not-null assertion, you don't have to explicitly handle your value being `null`. Instead, it throws an exception when encountering a `null` value.**



Here's a trivial example of a function that uses the assertion to convert a nullable argument to a non-null one.

**Listing 7.7. Using a not-null assertion**

```
fun ignoreNulls(str: String?) {
    val strNotNull: String = str!! #1
    println(strNotNull.length)
}

fun main() {
    ignoreNulls(null)
```

```
// Exception in thread "main" java.lang.NullPointerException
// at <...>.ignoreNulls(07_NotnullAssertions.kt:2)
}
```

What happens if `str` is `null` in this function? Kotlin doesn't have much choice: it will throw an exception at runtime. But note that the place where the exception is thrown is the assertion itself, not a subsequent line where you're trying to use the value. Essentially, you're telling the compiler, "I know the value isn't `null`, and I'm ready for an exception if it turns out I'm wrong."



#### Note

You may notice that the double exclamation mark looks a bit rude: it's almost like you're yelling at the compiler. This is intentional. The designers of Kotlin are trying to nudge you toward a better solution that doesn't involve making assertions that can't be verified by the compiler.

But there are situations when not-`null` assertions are the appropriate solution for a problem. When you check for `null` in one function and use the value in another function, the compiler can't recognize that the use is safe. If you're certain the check is always performed in another function, you may not want to duplicate it before using the value; then you can use a not-`null` assertion instead.

This happens in practice with action classes, which you might encounter in UI frameworks. In an action class, there are separate methods for updating the state of an action (to enable or disable it) and for executing it. The checks performed in the `update` method ensure that the `execute` method won't be called if the conditions aren't met, but there's no way for the compiler to recognize that.

Let's look at a hypothetical example of an action class that uses a not-`null` assertion in this situation. The `CopyRowAction` action, which operates on a `SelectableTextList`, is supposed to copy the value of the selected row in a list to the clipboard. We've omitted all the unnecessary details, keeping only the code responsible for checking whether any row was selected (meaning

therefore the action can be performed) and obtaining the value for the selected row. We imply here that that `executeCopyRow` is called only when `isActionEnabled` is true. That also means that `list.selectedIndex` will never be null when `executeCopyRow` is invoked (even though the compiler doesn't know this):

**Listing 7.8. Using a not-null assertion in an action class**

```
class SelectableTextList(  
    val contents: List<String>,  
    var selectedIndex: Int? = null,  
)  
  
class CopyRowAction(val list: SelectableTextList) {  
    fun isActionEnabled(): Boolean =  
        list.selectedIndex != null  
  
    fun executeCopyRow() { #1  
        val index = list.selectedIndex!!  
        val value = list.contents[index]  
        // copy value to clipboard  
    }  
}
```

Note that if you don't want to use `!!` in this case, you can write `val index = list.selectedIndex! ?: return` to obtain the index as a non-null type. If you use that pattern, a nullable value of `list.selectedIndex` will cause an early return from the function, so `value` will always be non-null. Although the not-null check using the Elvis operator is redundant here, it may be a good protection against `isActionEnabled` becoming more complicated later.

There's one more caveat to keep in mind: when you use `!!` and it results in an exception, the stack trace identifies the line number in which the exception was thrown but not a specific expression. To make it clear exactly which value was null, it's best to avoid using multiple `!!` assertions on the same line:

```
person.company!! .address!! .country #1
```

If you get an exception in this line, you won't be able to tell whether it was `company` or `address` that held a null value.

So far, we've discussed mostly how to *access* the values of nullable types. But what should you do if you need to pass a nullable value as an argument to a function that expects a non-null value? The compiler doesn't allow you to do that without a check, because doing so is unsafe. The Kotlin language doesn't have any special support for this case, but there's a standard library function that can help you: it's called `let`.

### 7.1.7 Dealing with nullable expressions: The "let" function

The `let` function makes it easier to deal with nullable expressions. Together with the safe-call operator, it allows you to evaluate an expression, check the result for `null`, and store the result in a variable, all in a single, concise expression.

One of its most common uses is handling a nullable argument that should be passed to a function that expects a non-null parameter. Let's say the function `sendEmailTo` takes one parameter of type `String` and sends an email to that address. This function is written in Kotlin and requires a non-null parameter:

```
fun sendEmailTo(email: String) { /*...*/ }
```

You can't pass a value of a nullable type to this function:

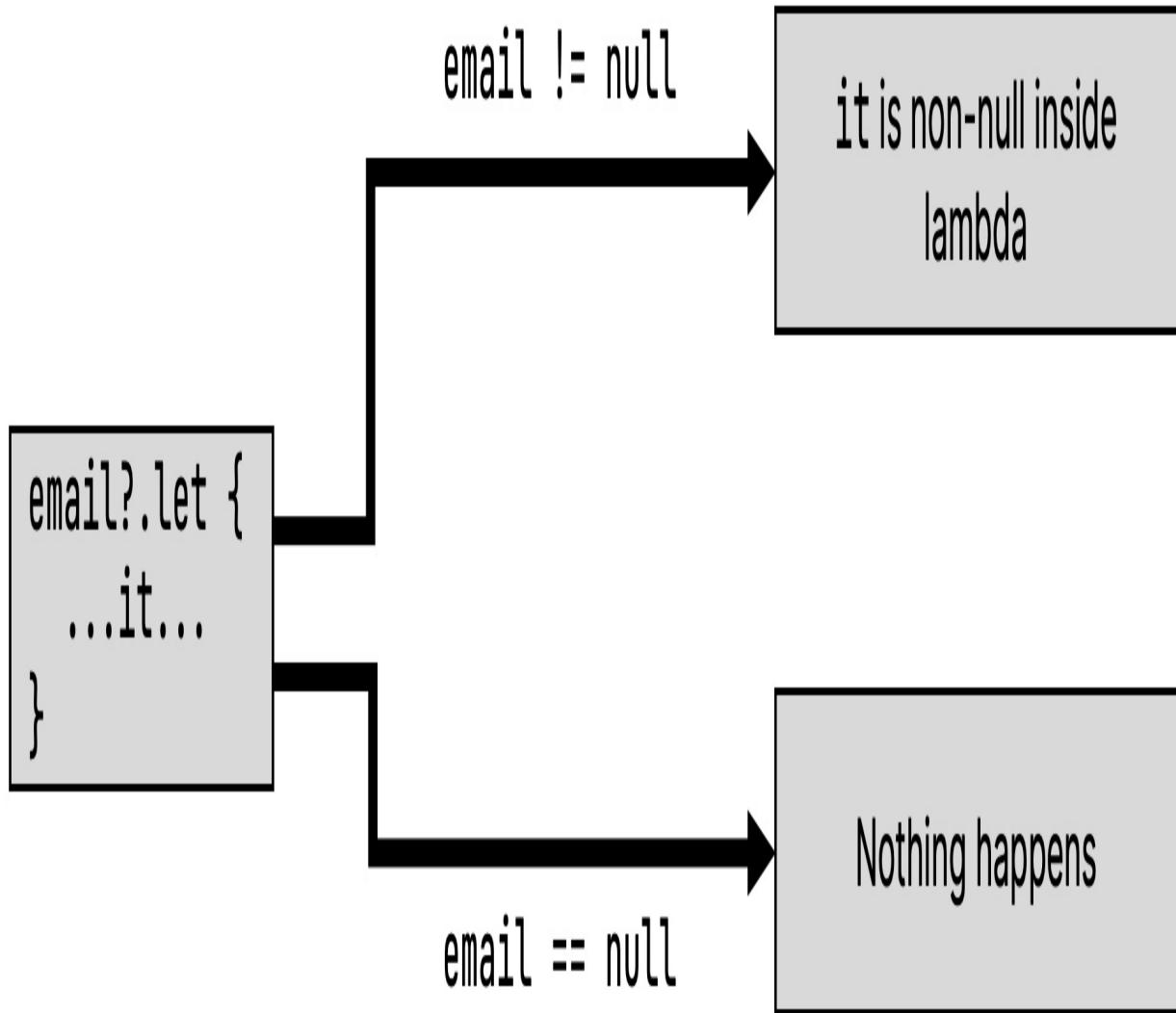
```
fun main() {
    val email: String? = "foo@bar.com"
    sendEmailTo(email)
    // ERROR: Type mismatch: inferred type is String? but String
}
```

You have to check explicitly whether this value isn't `null`:

```
if (email != null) sendEmailTo(email)
```

But you can go another way: use the `let` function, and call it via a safe call. All the `let` function does is turn the object on which it's called into a parameter of the lambda. In that way, it is similar to some of the other scope functions you got to know in [5.4](#). However, if you combine it with the safe call syntax, it effectively converts an object of a nullable type on which you call `let` into one of non-null type (see [7.6](#)).

**Figure 7.6.** Together with the safe-call operator, `let` allows you to specify a lambda that is only executed if your expression isn't null. This is particularly useful when you are working with the result of a chain of expressions that happens to be nullable.



The `let` function will be called only if the `email` value is non-null, so you use the `email` as a non-null argument of the lambda:

```
email?.let { email -> sendEmailTo(email) }
```

Switching to the short syntax using the autogenerated name `it`, the result is much more concise: `email?.let { sendEmailTo(it) }`. Here's a more complete example that shows this pattern.

### **Listing 7.9. Using let to call a function with a non-null parameter**

```
fun sendEmailTo(email: String) {  
    println("Sending email to $email")  
}  
  
fun main() {  
    var email: String? = "yole@example.com"  
    email?.let { sendEmailTo(it) }  
    // Sending email to yole@example.com  
    email = null  
    // email?.let { sendEmailTo(it) }  
}
```

Note that the `let` notation is especially convenient when you have to use the value of a longer expression if it's not `null`. You don't have to create a separate variable in this case. Compare this explicit `if` check

```
val person: Person? = getTheBestPersonInTheWorld()  
if (person != null) sendEmailTo(person.email)
```

to the same code without an extra variable:

```
getTheBestPersonInTheWorld()?.let { sendEmailTo(it.email) }
```

This function returns `null`, so the code in the lambda will never be executed:

```
fun getTheBestPersonInTheWorld(): Person? = null
```

When you need to check multiple values for `null`, you can use nested `let` calls to handle them. But in most cases, such code ends up fairly verbose and hard to follow. It's generally easier to use a regular `if` expression to check all the values together.

### **Comparing Kotlin's scope functions: When to use "with", "apply", "let", "run", and "also"**

During the last chapters, you've taken a detailed look at multiple functions with very similar signatures: `with`, `apply`, `let`, `run` and `also`.

All of these *scope functions* execute a block of code in the context of an object. They differ in how you refer to the object in question from inside the lambda, as well as their return value:

Function	Access to x via	Return value
x.let { ... }	it	Result of lambda
x.also { ... }	it	x
x.apply { ... }	this	x
x.run { ... }	this	Result of lambda
with(x) { ... }	this	Result of lambda

Their differences are quite subtle. Thus, it's worth once again pointing out the tasks each of them is particularly suited for, so you can compare them side by side:

- Use `let` together with the safe call operator `?.` to execute a block of code only when the object you are working with is not `null`. Use a standalone `let` to turn an expression into a variable, limited to the scope of its lambda.
- Use `apply` to configure properties of your object using a builder-style API, e.g. when creating an instance.
- Use `also` to execute additional actions that use your object, while passing the original object to further chained operations.
- Use `with` to group function calls on the same object, without having to repeat its name.
- Use `run` to configure an object *and* compute a custom result.

The usage of the different scope functions differs mainly in the details, so you might find yourself in a situation where more than one scope function seems like a good fit. For those cases, it makes sense to agree on conventions used in your team or for your project.

One other common situation is properties that are effectively non-null but can't be initialized with a non-null value in the constructor. Let's see how Kotlin allows you to deal with that situation.

## 7.1.8 Non-null types without immediate initialization: Late-initialized properties

Many frameworks initialize objects in dedicated methods called after the object instance has been created. For example, in Android, the activity initialization happens in the `onCreate` method. In JUnit, it is customary to put initialization logic in methods annotated with `@BeforeAll` or `@BeforeEach`.

But you can't leave a non-null property without an initializer in the constructor and only initialize it in a special method. Kotlin normally requires you to initialize all properties in the constructor, and if a property has a non-null type, you have to provide a non-null initializer value. If you can't provide that value, you have to use a nullable type instead. If you do that, every access to the property requires either a `null` check or the `!!` operator.

**Listing 7.10. Using non-null assertions to access a nullable property**

```
class MyService {
    fun performAction(): String = "Action Done!"
}

class MyTest {
    private var myService: MyService? = null #1

    @BeforeAll fun setUp() {
        myService = MyService() #2
    }

    @Test fun testAction() {
        assertEquals("Action Done!", myService!!.performAction())
    }
}
```

This looks ugly, especially if you access the property many times. To solve this, you can declare the `myService` property as *late-initialized*. This is done by applying the `lateinit` modifier.

### **Listing 7.11. Using a late-initialized property**

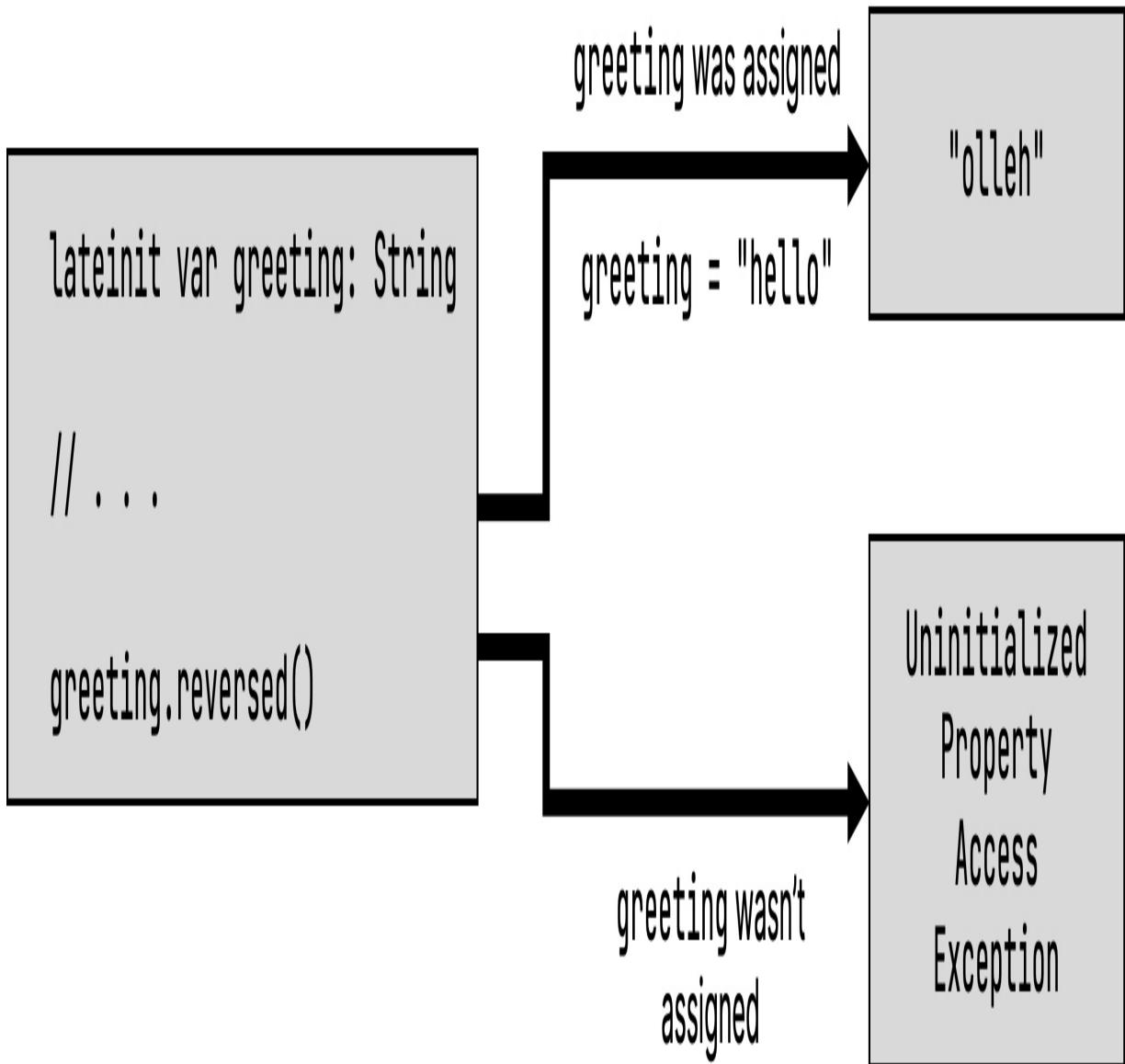
```
class MyService {  
    fun performAction(): String = "Action Done!"  
}  
  
class MyTest {  
    private lateinit var myService: MyService #1  
  
    @BeforeAll fun setUp() {  
        myService = MyService() #2  
    }  
  
    @Test fun testAction() {  
        assertEquals("Action Done!", myService.performAction()) #  
    }  
}
```

Note that a late-initialized property is always a `var`, because you need to be able to change its value outside of the constructor, and `val` properties are compiled into final fields that must be initialized in the constructor. But you no longer need to initialize it in a constructor, even though the property has a non-null type. If you access the property before it's been initialized, you get the following:

```
kotlin.UninitializedPropertyAccessException:  
    lateinit property myService has not been initialized
```

It clearly identifies what has happened and is much easier to understand than a generic `NullPointerException`.

**Figure 7.7. A `lateinit` property is has a non-null type, but doesn't need to be assigned a value right away. It is your responsibility not to access the variable before it was assigned a value.**



`lateinit` properties are commonly used in conjunction with Java dependency injection frameworks like Google Guice. In that scenario, the values of `lateinit` properties are set externally by the framework. To ensure compatibility with a broad range of Java frameworks, Kotlin generates a field with the same visibility as the `lateinit` property. If the property is declared as `public`, the field will be `public` as well.



Note

The `lateinit` modifier isn't restricted to properties of classes. You can also specify local variables inside a function body or lambda, as well as top-level properties, to be late-initialized.

Now let's look at how you can extend Kotlin's set of tools for dealing with `null` values by defining extension functions for nullable types.

### 7.1.9 Extending types without the safe-call operator: Extensions for nullable types

Defining extension functions for nullable types is one more powerful way to deal with `null` values. Rather than ensuring that a variable can't be `null` before a method call, you can allow the calls with `null` as a receiver, and deal with `null` in the function. This is only possible for extension functions; regular member calls are dispatched through the object instance and therefore can never be performed when the instance is `null`.

As an example, consider the functions `isEmpty` and `isBlank`, defined as extensions of `String` in the Kotlin standard library. The first one checks whether the string is an empty string "", and the second one checks whether it's empty or if it consists solely of whitespace characters. You'll generally use these functions to check that the string is non-trivial in order to do something meaningful with it. You may think it would be useful to handle `null` in the same way as trivial empty or blank strings. And, indeed, you can do so: the functions `isEmptyOrNull` and `isBlankOrNull` can be called with a receiver of type `String?`.

**Listing 7.12. Calling an extension function with a nullable receiver**

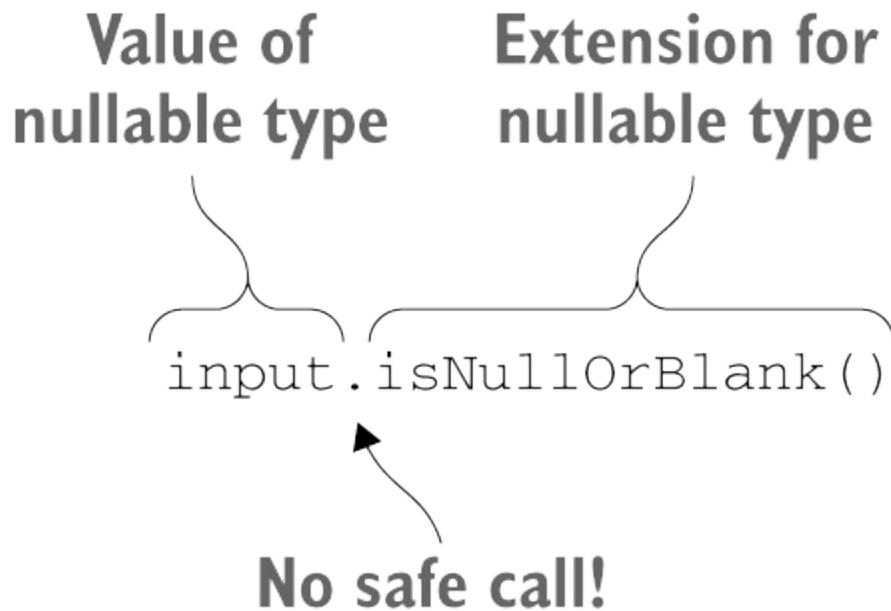
```
fun verifyUserInput(input: String?) {
    if (input.isNullOrEmpty()) { #1
        println("Please fill in the required fields")
    }
}

fun main() {
    verifyUserInput(" ")
    // Please fill in the required fields
    verifyUserInput(null)
```

```
// Please fill in the required fields #2  
}
```

You can call an extension function that was declared for a nullable receiver without safe access (see [7.8](#)). The function handles possible `null` values.

**Figure 7.8. Extensions for nullable types know how to handle the `null`-case for their receiver themselves. Therefore, they can be accessed without a safe call.**



The function `isNullOrEmpty` checks explicitly for `null`, returning `true` in this case, and then calls `isBlank`, which can be called on a non-null `String` only:

```
fun String?.isNullOrEmpty(): Boolean = #1  
    this == null || this.isBlank() #2
```

When you declare an extension function for a nullable type (ending with `?`), that means you can call this function on nullable values; and `this` in a function body can be `null`, so you have to check for that explicitly. In Java, `this` is always not-`null`, because it references the instance of a class you're in. In Kotlin, that's no longer the case: in an extension function for a nullable type, `this` can be `null`.

Note that the `let` function we discussed earlier can be called on a nullable

receiver as well, but it doesn't check whether the value is `null`. If you invoke it on a nullable type without using the safe-call operator, the lambda argument will also be nullable. It also means the passed block of code will always be executed, whether the value turns out to be `null` or not:

```
fun sendEmailTo(email: String) {  
    println("Sending email to $email")  
}  
  
fun main() {  
    val recipient: String? = null  
    recipient.let { sendEmailTo(it) } #1  
    //ERROR: Type mismatch: inferred type is String? but String was  
}
```

Therefore, if you want to check the arguments for being non-null with `let`, you have to use the safe-call operator `?.`, as you saw earlier: `recipient.let { sendEmailTo(it) }`.



#### Note

When you define your own extension function, you need to consider whether you should define it as an extension for a nullable type. By default, define it as an extension for a non-null type. You can safely change it later (no code will be broken) if it turns out it's used mostly on nullable values, and the `null` value can be reasonably handled.

This section showed you something unexpected. If you dereference a variable without an extra check, as in `s.isNullOrEmpty()`, it doesn't immediately mean the variable is non-null: the function can be an extension for a nullable type. Next, let's discuss another case that may surprise you: a type parameter can be nullable even without a question mark at the end.

### 7.1.10 Nullability of type parameters

By default, all type parameters of functions and classes in Kotlin are nullable. Any type, including a nullable type, can be substituted for a type parameter; in this case, declarations using the type parameter as a type are allowed to be `null`, even though the type parameter `T` doesn't end with a question mark.

Consider the following example.

**Listing 7.13. Dealing with a nullable type parameter**

```
fun <T> printHashCode(t: T) {  
    println(t?.hashCode()) #1  
}  
  
fun main() {  
    printHashCode(null) #2  
    // null  
}
```

In the `printHashCode` call, the inferred type for the type parameter `T` is a nullable type, `Any?`. Therefore, the parameter `t` is allowed to hold `null`, even without a question mark after `T`.

To make the type parameter non-null, you need to specify a non-null upper bound for it. That will reject a nullable value as an argument.

**Listing 7.14. Declaring a non-null upper bound for a type parameter**

```
fun <T: Any> printHashCode(t: T) { #1  
    println(t.hashCode())  
}  
  
fun main() {  
    printHashCode(null) #2  
    // Error: Type parameter bound for `T` is not satisfied  
    printHashCode(42)  
    // 42  
}
```

Chapter 11 will cover generics in Kotlin, and **Chapter 11** will cover this topic in more detail.

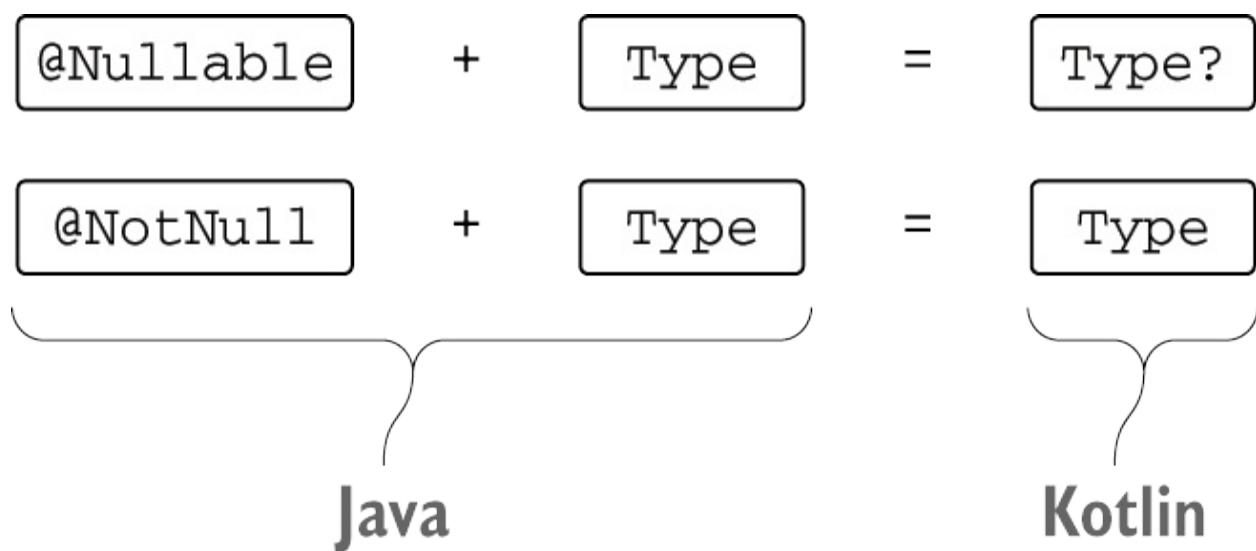
Note that type parameters are the only exception to the rule that a question mark at the end is required to mark a type as nullable, and types without a question mark are non-null. The next section shows another special case of nullability: types that come from the Java code.

## 7.1.11 Nullability and Java

The previous discussion covered the tools for working with `null` values in the Kotlin world. But Kotlin prides itself on its Java interoperability, and you know that Java doesn't support nullability in its type system. So what happens when you combine Kotlin and Java? Do you lose all safety, or do you have to check every value for `null`? Or is there a better solution? Let's find out.

First, as we mentioned, sometimes Java code contains information about nullability, expressed using annotations. When this information is present in the code, Kotlin uses it. Thus `@Nullable String` in Java is seen as `String?` by Kotlin, and `@NotNull String` is just `String` (see [7.9](#))

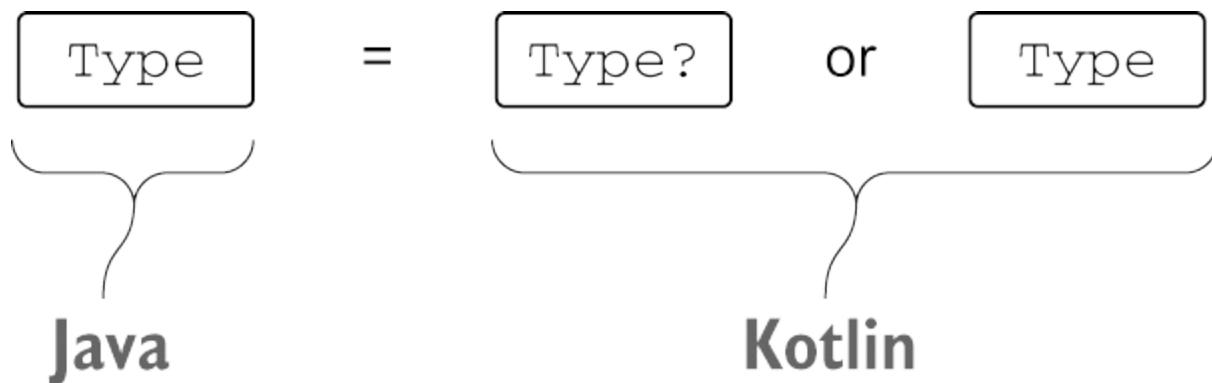
**Figure 7.9. Annotated Java types are represented as nullable and non-`null` types in Kotlin, according to the annotations. Those types can either explicitly store `null` values, or are explicitly non-`null`.**



Kotlin recognizes many different flavors of nullability annotations, including those from the JSR-305 standard (in the `javax.annotation` package), the Android ones (`android.support.annotation`), and those supported by JetBrains tools (`org.jetbrains.annotations`). The interesting question is what happens when the annotations aren't present. In that case, the Java type becomes a *platform type* in Kotlin.

## Platform types

**Figure 7.10. Java types without special annotations are represented in Kotlin as platform types. You can choose to either use them as a nullable type or as a non-null type.**



A platform type is essentially a type for which Kotlin doesn't have nullability information; you can work with it as either a nullable or a non-null type (see [7.10](#)). This means, just as in Java, you have full responsibility for the operations you perform with that type. The compiler will allow all operations. It also won't highlight any null-safe operations on such values as redundant, which it normally does when you perform a null-safe operation on a value of a non-null type. If you know the value can be null, you can compare it with null before use. If you know it's not null, you can use it directly. Just as in Java, you'll get a NullPointerException at the usage site if you get this wrong.

Let's say the class Person is declared in Java.

**Listing 7.15. A Java class without nullability annotations**

```
/* Java */
public class Person {
    private final String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Can `getName` return `null` or not? The Kotlin compiler knows nothing about nullability of the `String` type in this case, so you have to deal with it yourself. If you’re sure the name isn’t `null`, you can dereference it in a usual way, as in Java, without additional checks. But be ready to get an exception in this case.

**Listing 7.16. Accessing a Java class without `null` checks**

```
fun yellAt(person: Person) {
    println(person.name.uppercase() + "!!!") #1
}

fun main() {
    yellAt(Person(null))
    // java.lang.NullPointerException: person.name must not be nu
}
```

Your other option is to interpret the return type of `getName()` as nullable and access it safely.

**Listing 7.17. Accessing a Java class with `null` checks**

```
fun yellAtSafe(person: Person) {
    println((person.name ?: "Anyone").uppercase() + "!!!")
}

fun main() {
    yellAtSafe(Person(null))
    // ANYONE!!!
}
```

In this example, `null` values are handled properly, and no runtime exception is thrown.

Be careful while working with Java APIs. Most of the libraries aren’t annotated, so you may interpret all the types as non-`null`, but that can lead to errors. To avoid errors, you should check the documentation (and, if needed, the implementation) of the Java methods you’re using to find out when they can return `null`, and add checks for those methods.

**Why platform types?**

Wouldn't it be safer for Kotlin to treat all values coming from Java as nullable? Such a design would be possible, but it would require a large number of redundant `null` checks for values that can never be `null`, because the Kotlin compiler wouldn't be able to see that information.

The situation would be especially bad with generics—for example, every `ArrayList<String>` coming from Java would be an `ArrayList<String?>?` in Kotlin, and you'd need to check values for `null` on every access or use a cast, which would defeat the safety benefits. Writing such checks is extremely annoying, so the designers of Kotlin went with the pragmatic option and allowed the developers to take responsibility for correctly handling values coming from Java.

You can't declare a variable of a platform type in Kotlin; these types can only come from Java code. But you may see them in error messages and in the IDE:

```
val i: Int = person.name
// ERROR: Type mismatch: inferred type is String! but Int was exp
```

The `String!` notation is how the Kotlin compiler and Kotlin IDEs like IntelliJ IDEA and Android Studio denote platform types coming from Java code. You can't use this syntax in your own code, and usually this exclamation mark isn't connected with the source of a problem, so you can usually ignore it. It just emphasizes that the nullability of the type is unknown.

**Figure 7.11. When using type inference for a Java property, IntelliJ IDEA and Android Studio indicate that you are working with a platform type if Inlay Hints for Kotlin Types are enabled. The exclamation point allows you to spot these platform types at a glance.**

```
fun main() {  
    val s: String! = p.name  
}
```

As we said already, you may interpret platform types any way you like—as nullable or as non-null—so both of the following declarations are valid:

```
val s: String? = person.name #1  
val s1: String = person.name #2
```

In this case, just as with the method calls, you need to make sure you get the nullability right. If you try to assign a null value coming from Java to a non-null Kotlin variable, you'll get an exception at the point of assignment.

We've discussed how Java types are seen from Kotlin. Let's now talk about some pitfalls of creating mixed Kotlin and Java hierarchies.

## Inheritance

When overriding a Java method in Kotlin, you have a choice whether to

declare the parameters and the return type as nullable or non-null. For example, let's look at a `StringProcessor` interface in Java.

**Listing 7.18. A Java interface with a `String` parameter**

```
/* Java */
interface StringProcessor {
    void process(String value);
}
```

In Kotlin, both of the following implementations will be accepted by the compiler.

**Listing 7.19. Implementing the Java interface with different parameter nullability**

```
class StringPrinter : StringProcessor {
    override fun process(value: String) {
        println(value)
    }
}

class NullableStringPrinter : StringProcessor {
    override fun process(value: String?) {
        if (value != null) {
            println(value)
        }
    }
}
```

Note that it's important to get nullability right when implementing methods from Java classes or interfaces. Because the implementation methods can be called from non-Kotlin code, the Kotlin compiler will generate non-null assertions for every parameter that you declare with a non-null type. If the Java code does pass a `null` value to the method, the assertion will trigger, and you'll get an exception, even if you never access the parameter value in your implementation.

Let's summarize our discussion of nullability. We've discussed nullable and non-null types and the means of working with them: operators for safe operations (safe call `?..`, Elvis operator `?:`, and safe cast `as?`), as well as the operator for unsafe dereference (the not-null assertion `!!`). You've seen how

the library function `let` can help you accomplish concise non-`null` checks and how extensions for nullable types can help move a not-`null` check into a function. We've also discussed platform types that represent Java types in Kotlin.

## 7.2 Summary

- Kotlin's support of nullable types detects possible `NullPointerException` errors at compile time.
- Regular types are non-`null` by default, unless they are explicitly marked as nullable. A question mark after a type name indicates that it is nullable.
- Kotlin provides a variety of tools for dealing with nullable types concisely.
- Safe calls (`?.`) allow you to call methods and access properties on nullable objects.
- The Elvis operator (`?:`) makes it possible to provide a default value for an expression that may be `null`, return from execution, or throw an exception.
- You can use not-`null` assertions (`!!`) to promise the compiler that a given value is not `null` (but will have to expect an exception if you break that promise).
- The `let` scope function turns the object on which it is called into the parameter for a lambda. Together with the safe-call operator, it effectively converts an object of nullable type into one of non-`null` type.
- The `as?` operator provides an easy way to cast a value to a type and to handle the case when it has a different type.

# 8 Basic types, collections, and arrays

## This chapter covers

- Primitive and other basic types and their correspondence to the Java types
- Kotlin collections, arrays, and their nullability and interoperability stories

Beyond its support for nullability, Kotlin’s type system has several essential features to improve the reliability of your code, and implements many lessons learned from other typesystems, including Java’s. These decisions shape the way you work with everything in Kotlin code, from primitive values and basic types to the hierarchy of collections found in the Kotlin standard library. Kotlin introduces features that aren’t present in other type systems, such as *read-only collections*, and refines or doesn’t expose parts of the type system that have turned out to be problematic or unnecessary, such as first-class support for arrays. Let’s take a closer look, starting with the basic building blocks.

## 8.1 Primitive and other basic types

This section describes the basic types used in programs, such as `Int`, `Boolean`, and `Any`. Unlike Java, Kotlin doesn’t differentiate primitive types and wrappers. You’ll shortly learn why, and how it works under the hood. You’ll see the correspondence between Kotlin types and such Java types as `Object` and `Void`, as well.

### 8.1.1 Representing integers, floating-point numbers, characters and Booleans with primitive types

As you may know, Java makes a distinction between primitive types and reference types. A variable of a *primitive type* (such as `int`) holds its value directly. A variable of a *reference type* (such as `String`) holds a reference to

the memory location containing the object.

Values of primitive types can be stored and passed around more efficiently, but you can't call methods on such values or store them in collections. Java provides special wrapper types (such as `java.lang.Integer`) that encapsulate primitive types in situations when an object is needed. Thus, to define a collection of integers, you can't say `Collection<int>`; you have to use `Collection<Integer>` instead.

Kotlin doesn't distinguish between primitive types and wrapper types. You always use the same type (for example, `Int`):

```
val i: Int = 1
val list: List<Int> = listOf(1, 2, 3)
```

That's convenient. What's more, you can call methods on values of a number type. For example, consider this snippet, which uses the `coerceIn` standard library function to restrict the value to the specified range:

```
fun showProgress(progress: Int) {
    val percent = progress.coerceIn(0, 100)
    println("We're ${percent}% done!")
}

fun main() {
    showProgress(146)
    // We're 100% done!
}
```

If primitive and reference types are the same, does that mean Kotlin represents all numbers as objects? Wouldn't that be terribly inefficient? Indeed it would, so Kotlin doesn't do that.

At runtime, the number types are represented in the most efficient way possible. In most cases—for variables, properties, parameters, and return types—Kotlin's `Int` type is compiled to the Java primitive type `int`. The only case in which this isn't possible is generic classes, such as collections. A primitive type used as a type argument of a generic class is compiled to the corresponding Java wrapper type. For example, if the `Int` type is used as a type argument of the collection, then the collection will store instances of

`java.lang.Integer`, the corresponding wrapper type.

The full list of types that correspond to Java primitive types is:

- *Integer types*—Byte, Short, Int, Long
- *Floating-point number types*—Float, Double
- *Character type*—Char
- *Boolean type*—Boolean

## 8.1.2 Using the full bit range to represent positive numbers: Unsigned number types

There are situations where you need to utilize the full bit range of an integer number representing positive values, for example when you're working on the bit-and-byte level, manipulating the pixels in a bitmap, the bytes in a file, or other binary data.

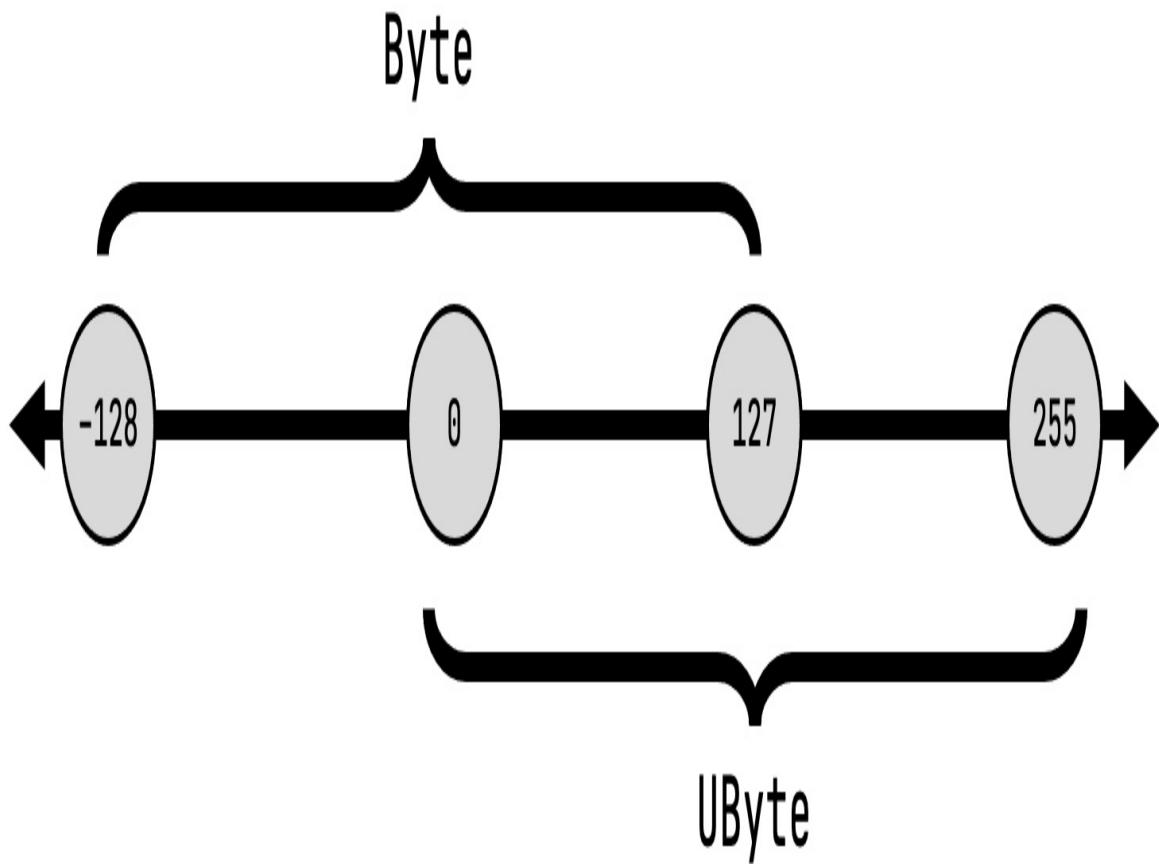
For situations like these, Kotlin extends the regular primitive types on the JVM with types for unsigned integer numbers. Specifically, there are four unsigned number types:

Type	Size	Value range
UByte	8 bit	0 - 255
UShort	16 bit	0 - 65535
UInt	32 bit	0 - $2^{32} - 1$
ULong	64 bit	0 - $2^{64} - 1$

Unsigned number types "shift" the value range compared to their signed counterparts, allowing you to store larger non-negative numbers in the same

amount of memory. A regular `Int` for example allows you to store numbers from roughly minus 2 billion to plus two billion. A `UInt` on the other hand can represent numbers between 0 and roughly 4 billion.

**Figure 8.1. Unsigned number types shift the value range, allowing you to store larger non-negative numbers in the same amount of memory: Where a regular, signed `Byte` can store values between -128 and 127, a `UByte` can store values between 0 and 255.**



Like other primitive types, unsigned numbers in Kotlin are only wrapped when required, and have the performance characteristics of primitive types otherwise.



**Note**

It may be tempting to use unsigned integers in situations where you want to express that non-negative integers are required. However, that's not the goal of Kotlin's unsigned number types. In cases where you don't explicitly need the full bit range, you are generally better served with regular integers, and checking that a non-negative value was passed to your function.

#### **Unsigned number types: implementation details**

If you take a look at the specification of the JVM (<http://mng.bz/nJa4>), you'll notice that the Java Virtual Machine itself does not specify or provide primitives for unsigned numbers. Kotlin can't change that, so it provides its own abstractions on top of the existing signed primitives.

It does so using a concept you learned about in [4.5: Inline classes](#). Each class representing an unsigned number is actually an inline class which uses its signed counterpart as a storage. That's right: under the hood, your `UInt` is just a regular `Int`. Because the Kotlin compiler takes care of replacing inline classes by the underlying property they wrap wherever possible, you can expect unsigned number types to perform equal to signed number types.

The Kotlin compiler can easily convert a type like `Int` to the corresponding primitive type on the JVM, because both types are capable of representing the same set of values (and neither can store a `null` reference). Likewise, when you use a Java declaration from Kotlin, Java primitive types become non-`null` types (not platform types), because they can't hold `null` values. Now let's discuss the nullable versions of the same types.

### **8.1.3 Nullable primitive types: `Int?`, `Boolean?`, and more**

Nullable types in Kotlin can't be represented by Java primitive types, because `null` can only be stored in a variable of a Java reference type. That means whenever you use a nullable version of a primitive type in Kotlin, it's compiled to the corresponding wrapper type.

To see the nullable types in use, let's go back to the opening example of the book and recall the `Person` class declared there. The class represents a person whose name is always known and whose age can be either known or

unspecified. Let's add a function that checks whether one person is older than another.

**Listing 8.1. Using nullable primitive types**

```
data class Person(val name: String,
                  val age: Int? = null) {

    fun isOlderThan(other: Person): Boolean? {
        if (age == null || other.age == null)
            return null
        return age > other.age
    }
}

fun main() {
    println(Person("Sam", 35).isOlderThan(Person("Amy", 42)))
    // false
    println(Person("Sam", 35).isOlderThan(Person("Jane")))
    // null
}
```

Note how the regular nullability rules apply here. You can't just compare two values of type `Int?`, because one of them may be `null`. Instead, you have to check that both values aren't `null`. After that, the compiler allows you to work with them normally.

The value of the `age` property declared in the class `Person` is stored as a `java.lang.Integer`. But this detail only matters if you're working with the class from Java. To choose the right type in Kotlin, you only need to consider whether `null` is a possible value for the variable or property.

As mentioned earlier, generic classes are another case when wrapper types come into play. If you use a primitive type as a type argument of a class, Kotlin uses the boxed representation of the type. For example, this creates a list of boxed `Integer` values, even though you've never specified a nullable type or used a `null` value:

```
val listOfInts = listOf(1, 2, 3)
```

This happens because of the way generics are implemented on the Java

virtual machine. The JVM doesn't support using a primitive type as a type argument, so a generic class (both in Java and in Kotlin) must always use a boxed representation of the type. As a consequence, if you need to efficiently store large collections of primitive types, you need to either use a third-party library like Eclipse Collections (<https://github.com/eclipse/eclipse-collections>) that provides support for such collections, or store them in arrays. We'll discuss arrays in detail at the end of this chapter.

Now let's look at how you can convert values between different primitive types.

### 8.1.4 Kotlin makes number conversions explicit

One important difference between Kotlin and Java is the way they handle numeric conversions. Kotlin doesn't automatically convert numbers from one type to the other, even when the type you're assigning your value to is larger, and could comfortably hold the value you're trying to assign. For example, the following code won't compile in Kotlin:

```
val i = 1
val l: Long = i #1
```

Instead, you need to apply the conversion explicitly:

```
val i = 1
val l: Long = i.toLong()
```

Conversion functions are defined for every primitive type (except Boolean): `toByte()`, `toShort()`, `toChar()` and so on. The functions support converting in both directions: extending a smaller type to a larger one, like `Int.toLong()`, and truncating a larger type to a smaller one, like `Long.toInt()`.

Kotlin makes the conversion explicit in order to avoid surprises, especially when comparing boxed values. The `equals` method for two boxed values checks the box type, not just the value stored in it. Thus, in Java, `Integer.valueOf(42).equals(Long.valueOf(42))` returns `false`. If Kotlin supported implicit conversions, you could write something like this:

```
val x = 1 #1
val list = listOf(1L, 2L, 3L) #2
x in list #3
```

This would evaluate to `false`, contrary to everyone's expectations. Thus the line `x in list` from this example doesn't compile. Kotlin requires you to convert the types explicitly so that only values of the same type are compared:

```
fun main() {
    val x = 1
    println(x.toLong() in listOf(1L, 2L, 3L))
    // true
}
```

If you use different number types in your code at the same time, you have to convert variables explicitly to avoid unexpected behavior.

### Primitive type literals

Kotlin supports the following ways to write number literals in source code, in addition to simple decimal numbers:

- Literals of type `Long` use the `L` suffix: `123L`.
- Literals of type `Double` use the standard representation of floating-point numbers: `0.12`, `2.0`, `1.2e10`, `1.2e-10`.
- Literals of type `Float` use the `f` or `F` suffix: `123.4f`, `.456F`, `1e3f`.
- Hexadecimal literals use the `0x` or `0X` prefix (such as `0xCAFEBABE` or `0xbcdL`).
- Binary literals use the `0b` or `0B` prefix (such as `0b0000000101`).
- Unsigned number literals use the `U` suffix: `123U`, `123UL`, `0x10cU`.

For character literals, you use mostly the same syntax as in Java. You write the character in single quotes, and you can also use escape sequences if you need to. The following are examples of valid Kotlin character literals: `'1'`, `'\t'` (the tab character), `'\u0009'` (the tab character represented using a Unicode escape sequence).

Note that when you're writing a number literal, you usually don't need to use conversion functions. One possibility is to use the special syntax to mark the

type of the constant explicitly, such as `42L` or `42.0f`. And even if you don't use it, the necessary conversion is applied automatically if you use a number literal to initialize a variable of a known type or pass it as an argument to a function. In addition, arithmetic operators are overloaded to accept all appropriate numeric types. For example, the following code works correctly without any explicit conversions:

```
fun printALong(l: Long) = println(l)

fun main() {
    val b: Byte = 1                      #1
    val l = b + 1L                        #2
    printALong(42)                       #3
    // 42
}
```

Note that the behavior of Kotlin arithmetic operators with regard to number-range overflow and underflow is exactly the same in Java; Kotlin doesn't introduce any extra overflow checks:

```
fun main() {
    println(Int.MAX_VALUE + 1)
    -2147483648 #1
    println(Int.MIN_VALUE - 1)
    2147483647 #2
}
```

### Conversion from String

The Kotlin standard library provides a set of extension functions to convert a string into a primitive type: `toInt`, `toByte`, `toBoolean`, and so on. Each of these functions tries to parse the contents of the string as the corresponding type and throws a `NumberFormatException` if the parsing fails:

```
fun main() {
    println("42".toInt())
    // 42
}
```

However, if you're expecting the conversion from string to primitive type to fail often, it can be cumbersome to always handle the `NumberFormatException` explicitly. For this case, each of these extension

functions also comes with a counterpart that returns `null` if the conversion fails: `toIntOrNull`, `toByteOrNull`, and so on:

```
fun main() {
    println("seven".toIntOrNull())
    // null
}
```

A special case is the conversion of strings to Boolean values. These conversion functions are defined on a nullable receiver, as we introduced them in [7.1.9](#). The `toBoolean` function returns `true` if the string it is called on is not `null`, and its content is equal to the word "true" (ignoring capitalization). Otherwise, it returns `false`:

```
fun main() {
    println("trUE".toBoolean())
    // true
    println("7".toBoolean())
    // false
    println(null.toBoolean())
    // false
}
```

For exact matches on the strings "true" and "false" during conversion, use the `toBooleanStrict` function, which only accepts these two values, and throws an exception otherwise.

Before we move on to other types, there are three more special types we need to mention: `Any`, `Unit`, and `Nothing`.

### 8.1.5 "Any" and "Any?": the root of the Kotlin type hierarchy

Similar to how `Object` is the root of the class hierarchy in Java, the `Any` type is the supertype of all non-nullable types in Kotlin. But in Java, `Object` is a supertype of all reference types only, and primitive types aren't part of the hierarchy. That means you have to use wrapper types such as `java.lang.Integer` to represent a primitive type value when `Object` is required. In Kotlin, `Any` is a supertype of all types, including the primitive types such as `Int`.

Just as in Java, assigning a value of a primitive type to a variable of type `Any` performs automatic boxing:

```
val answer: Any = 42 #1
```

Note that `Any` is a non-null type, so a variable of the type `Any` can't hold the value `null`. If you need a variable that can hold any possible value in Kotlin, including `null`, you must use the `Any?` type.

Under the hood, the `Any` type corresponds to `java.lang.Object`. The `Object` type used in parameters and return types of Java methods is seen as `Any` in Kotlin. (More specifically, it's viewed as a platform type, because its nullability is unknown.) When a Kotlin function uses `Any`, it's compiled to `Object` in the Java bytecode.

As you saw in chapter 4, all Kotlin classes have the following three methods: `toString`, `equals`, and `hashCode`. These methods are inherited from `Any`. Other methods defined on `java.lang.Object` (such as `wait` and `notify`) aren't available on `Any`, but you can call them if you manually cast the value to `java.lang.Object`.

### 8.1.6 The Unit type: Kotlin's "void"

The `Unit` type in Kotlin fulfills the same function as `void` in Java. It can be used as a return type of a function that has nothing interesting to return:

```
fun f(): Unit { ... }
```

Syntactically, it's the same as writing a function with a block body without a type declaration:

```
fun f() { ... } #1
```

In most cases, you won't notice the difference between `void` and `Unit`. If your Kotlin function has the `Unit` return type and doesn't override a generic function, it's compiled to a good-old `void` function under the hood. If you override it from Java, the Java function just needs to return `void`.

What distinguishes Kotlin's `Unit` from Java's `void`, then? `Unit` is a full-

fledged type, and, unlike `void`, it can be used as a type argument. Only one value of this type exists; it's also called `Unit` and is returned *implicitly*. This is useful when you override a function that returns a generic parameter and make it return a value of the `Unit` type:

```
interface Processor<T> {
    fun process(): T
}

class NoResultProcessor : Processor<Unit> {
    override fun process() {                      #1
        // do stuff
    }                                              #2
}
```

The signature of the interface requires the `process` function to return a value; and, because the `Unit` type does have a value, it's no problem to return it from the method. But you don't need to write an explicit `return` statement in `NoResultProcessor.process`, because `return Unit` is added implicitly by the compiler.

Contrast this with Java, where neither of the possibilities for solving the problem of using "no value" as a type argument is as nice as the Kotlin solution. One option is to use separate interfaces (such as `Callable` and `Runnable`) to represent interfaces that don't and do return a value. The other is to use the special `java.lang.Void` type as the type parameter. If you use the second option, you still need to put in an explicit `return null;` to return the only possible value matching that type, because if the return type isn't `void`, you must always have an explicit `return` statement.

You may wonder why we chose a different name for `Unit` and didn't call it `Void`. The name `Unit` is used traditionally in functional languages to mean "only one instance," and that's exactly what distinguishes Kotlin's `Unit` from Java's `void`. We could have used the customary `Void` name, but Kotlin has a type called `Nothing` that performs an entirely different function. Having two types called `Void` and `Nothing` would be confusing because the meanings are so close. So what's this `Nothing` type about? Let's find out.

### 8.1.7 The `Nothing` type: "This function never returns"

For some functions in Kotlin, the concept of a "return value" doesn't make sense because they never complete successfully. For example, many testing libraries have a function called `fail` that fails the current test by throwing an exception with a specified message. A function that has an infinite loop in it will also never complete successfully.

When analyzing code that calls such a function, it's useful to know that the function will never terminate normally. To express that, Kotlin uses a special return type called `Nothing`:

```
fun fail(message: String): Nothing {
    throw IllegalStateException(message)
}

fun main() {
    fail("Error occurred")
    // java.lang.IllegalStateException: Error occurred
}
```

The `Nothing` type doesn't have any values, so it only makes sense to use it as a function return type or as a type argument for a type parameter that's used as a generic function return type. In all other cases, declaring a variable where you can't store any value doesn't make sense.

Note that functions returning `Nothing` can be used on the right side of the Elvis operator to perform precondition checking:

```
val address = company.address ?: fail("No address")
println(address.city)
```

This example shows why having `Nothing` in the type system is extremely useful. The compiler knows that a function with this return type never terminates normally and uses that information when analyzing the code calling the function. In the previous example, the compiler infers that the type of `address` is non-null, because the branch handling the case when it's `null` always throws an exception, and won't continue the execution of the code that follows.

We've finished our discussion of the basic types in Kotlin: primitive types, `Any`, `Unit`, and `Nothing`. Now let's look at the collection types and how they

differ from their Java counterparts.

## 8.2 Collections and arrays

You've already seen many examples of code that uses various collection APIs, and since [3.1](#), you know that Kotlin builds on the Java collections library and augments it with features added through extension functions. There's more to the story of the collection support in Kotlin and the correspondence between Java and Kotlin collections, and now is a good time to look at the details.

### 8.2.1 Collections of nullable values and nullable collections

Earlier in this chapter, we discussed the concept of nullable types, but we only briefly touched on nullability of type arguments. But this is essential for a consistent type system: it's no less important to know whether a collection can hold `null` values than to know whether the value of a variable can be `null`. The good news is that Kotlin fully supports nullability for type arguments. Just as the type of a variable can have a `?` character appended to indicate that the variable can hold `null`, a type used as a type argument can be marked in the same way. To see how this works, let's look at an example of a function that takes an input text and tries to parse each line in the input string as a number.

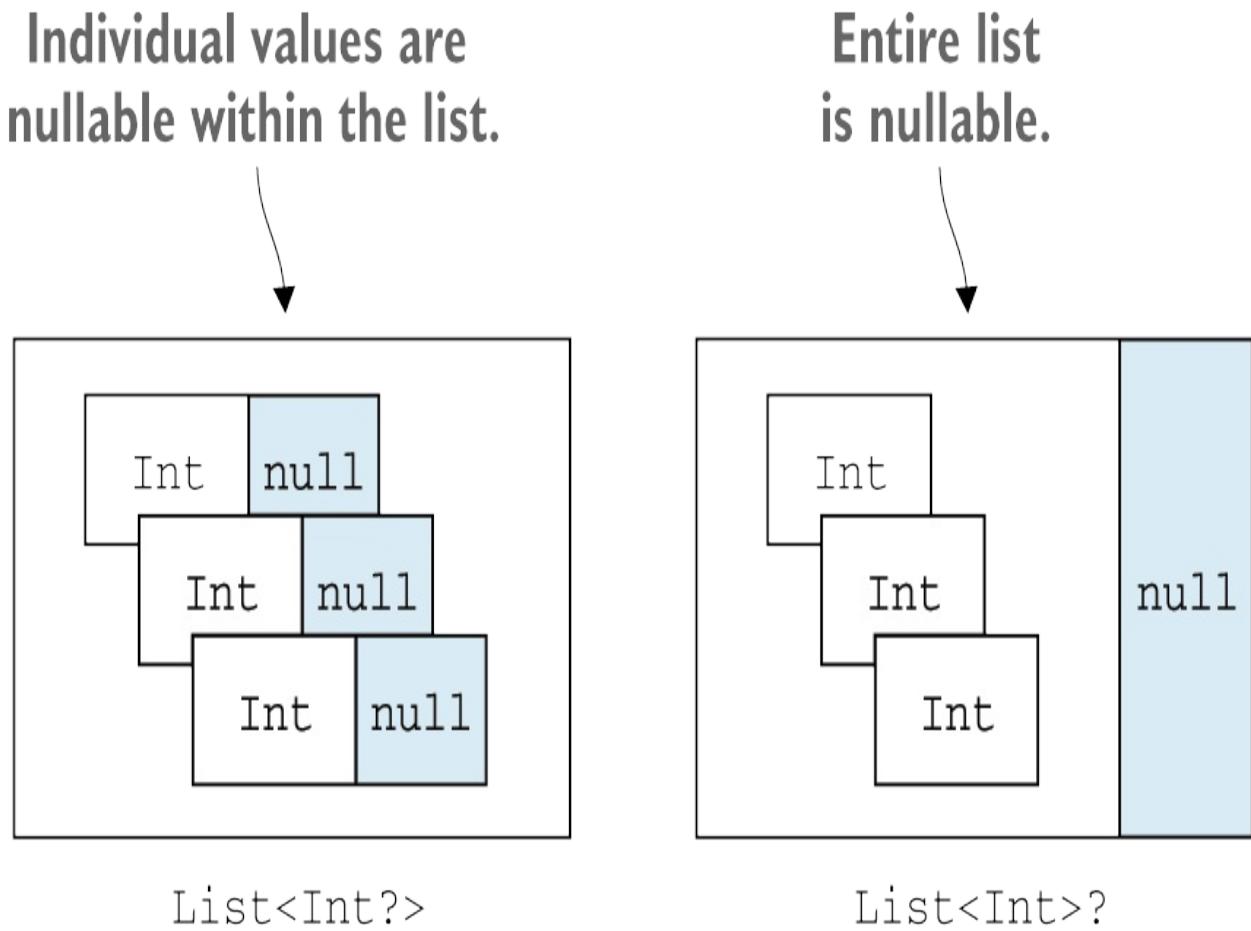
**Listing 8.2. Building a collection of nullable values**

```
fun readNumbers(text: String): List<Int?> {
    val result = mutableListOf<Int?>() #1
    for (line in text.lineSequence()) { #2
        val numberOrNull = line.toIntOrNull()
        result.add(numberOrNull) #3
    }
    return result
}
```

`List<Int?>` is a list that can hold values of type `Int?:` in other words, `Int` or `null`. You add an integer to the `result` list if the line can be parsed, or `null` otherwise.

Note how the nullability of the type of the variable itself is distinct from the nullability of the type used as a type argument. The difference between a list of nullable `Int`s and a nullable list of `Int`s is illustrated in [8.2](#).

**Figure 8.2.** Carefully consider how you intend to use your collection when thinking about nullability. Should the whole collection itself be nullable, or should individual elements inside the collection be nullable?



In the first case, the list itself is always not `null`, but each value in the list can be `null`. A variable of the second type may contain a `null` reference instead of a list instance, but the elements in the list are guaranteed to be non-`null`.

By the way: Given our knowledge of functional programming and lambdas, we can actually shrink this example by using the `map` function which we first saw in [6.1.1](#). It applies a given function—in this case, `toIntOrNull`—to each element in the input sequence, which we can then collect in a result list:

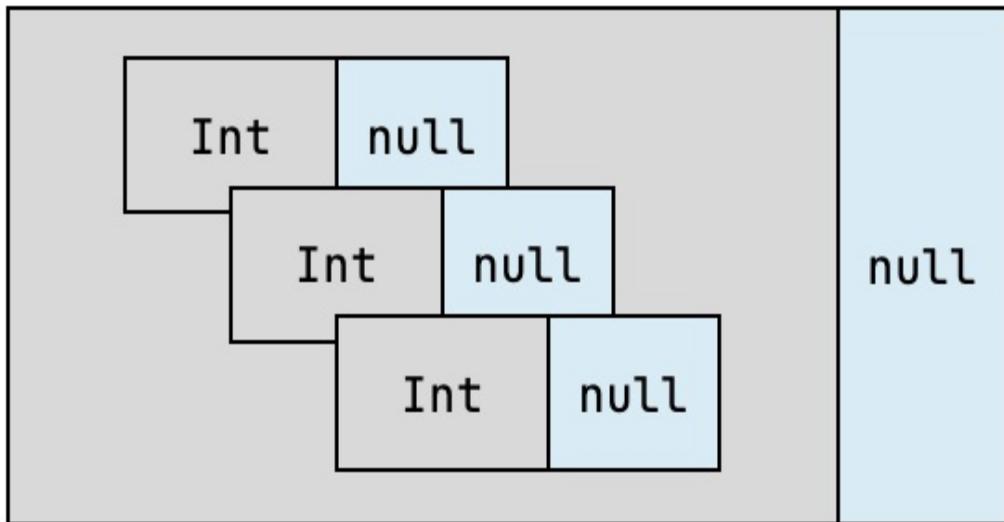
**Listing 8.3. Shortening the `readNumbers` method with "map"**

```
fun readNumbers2(text: String): List<Int?> =  
    text.lineSequence().map { it.toIntOrNull() }.toList()
```

You may also find yourself in a situation where you would like to declare a variable that holds a nullable list of nullable numbers. This allows you to express that individual elements in the list can be absent, but also that the list as a whole may be absent, as well. The Kotlin way to write this is `List<Int?>`, with two question marks. The inner question mark specifies that the *elements* of the list are nullable. The outer question mark specifies that the *list itself* is nullable. You need to apply `null` checks both when using the value of the variable and when using the value of every element in the list.

**Figure 8.3. A nullable collection of nullable integers can be null itself, or store elements which are potentially null.**

# Entire list and its individual values are nullable



`List<Int?>?`

To see how you can work with a list of nullable values, let's write a function to add all the valid numbers together and count the invalid numbers separately.

**Listing 8.4. Working with a collection of nullable values**

```
fun addValidNumbers(numbers: List<Int?>) {  
    var sumOfValidNumbers = 0  
    var invalidNumbers = 0  
    for (number in numbers) { #1  
        if (number != null) { #2  
            sumOfValidNumbers += number  
        } else {  
            invalidNumbers++  
        }  
    }  
}
```

```

        println("Sum of valid numbers: $sumOfValidNumbers")
        println("Invalid numbers: $invalidNumbers")
    }

fun main() {
    val input = """
        1
        abc
        42
    """.trimIndent() #3
    val numbers = readNumbers(input)
    addValidNumbers(numbers)
    // Sum of valid numbers: 43
    // Invalid numbers: 1
}

```

There isn't much special going on here. When you access an element of the list, you get back a value of type `Int?`, and you need to check it for `null` before you can use it in arithmetical operations.

Taking a collection of nullable values and filtering out `null` is such a common operation that Kotlin provides a standard library function `filterNotNull` to perform it. Here's how you can use it to greatly simplify the previous example.

#### **Listing 8.5. Using `filterNotNull` with a collection of nullable values**

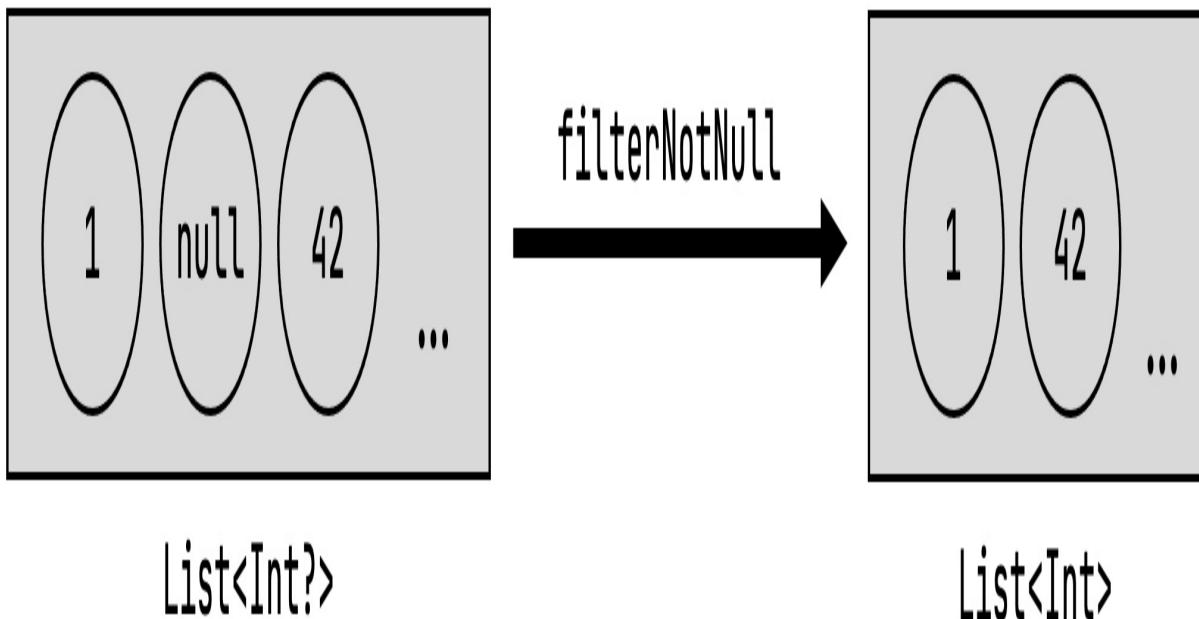
```

fun addValidNumbers(numbers: List<Int?>) {
    val validNumbers = numbers.filterNotNull()
    println("Sum of valid numbers: ${validNumbers.sum()}")
    println("Invalid numbers: ${numbers.size - validNumbers.size}")
}

```

Of course, the filtering also affects the type of the collection. The type of `validNumbers` is `List<Int>`, because the filtering ensures that the collection doesn't contain any `null` elements.

**Figure 8.4.** The `filterNotNull` function returns a new collection with all the `null` elements from the input collection removed. This new collection is also of non-nullable type, meaning you won't have to do any further `null`-handling down the line.



Now that you understand how Kotlin distinguishes between collections that hold nullable and non-null elements, let's look at another major distinction introduced by Kotlin: read-only versus mutable collections.

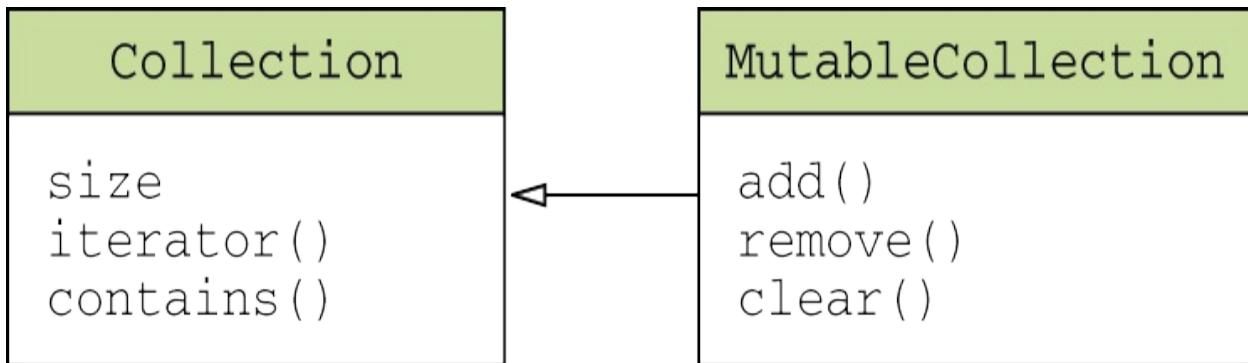
### 8.2.2 Read-only and mutable collections

An important trait that sets apart Kotlin's collection design from Java's is that it separates interfaces for accessing the data in a collection and for modifying the data. This distinction exists starting with the most basic interface for working with collections, `kotlin.collections.Collection`. Using this interface, you can iterate over the elements in a collection, obtain its size, check whether it contains a certain element, and perform other operations that read data from the collection. But this interface doesn't have any methods for adding or removing elements.

To modify the data in the collection, use the `kotlin.collections.MutableCollection` interface. It extends the regular `kotlin.collections.Collection` and provides methods for adding and removing the elements, clearing the collection, and so on. [8.5](#) shows the key

methods defined in the two interfaces.

**Figure 8.5. The `Collections` interface is ready-only. `MutableCollection` extends it and adds methods to modify a collection's contents.**



As a general rule, you should use read-only interfaces everywhere in your code. Use the mutable variants only if the code will modify the collection.

Just like the separation between `val` and `var`, the separation between read-only and mutable interfaces for collections makes it much easier to understand what's happening with data in your program. If a function takes a parameter that is a `Collection` but not a `MutableCollection`, you know it's not going to modify the collection, but only read data from it. And if a function requires you to pass a `MutableCollection`, you can assume that it's going to modify the data. If you have a collection that's part of the internal state of your component, you may need to make a copy of that collection before passing it to such a function. (This pattern is usually called a *defensive copy*.)

For example, you can clearly see that the following `copyElements` function will modify the target collection but not the source collection.

#### **Listing 8.6. Using read-only and mutable collection interfaces**

```
fun <T> copyElements(source: Collection<T>,
                      target: MutableCollection<T>) {
    for (item in source) { #1
        target.add(item) #2
    }
}
```

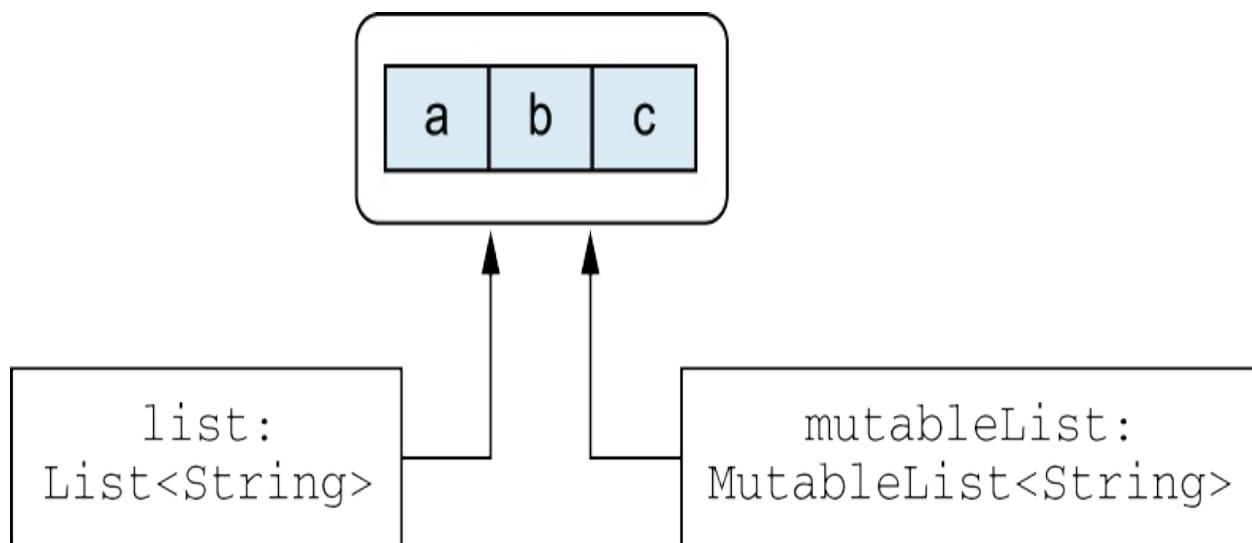
```
fun main() {  
    val source: Collection<Int> = arrayListOf(3, 5, 7)  
    val target: MutableCollection<Int> = arrayListOf(1)  
    copyElements(source, target)  
    println(target)  
    // [1, 3, 5, 7]  
}
```

You can't pass a variable of a read-only collection type as the target argument, even if its value is a mutable collection:

```
fun main() {  
    val source: Collection<Int> = arrayListOf(3, 5, 7)  
    val target: Collection<Int> = arrayListOf(1)  
    copyElements(source, target) #1  
    // Error: Type mismatch: inferred type is Collection<Int>  
    // but MutableCollection<Int> was expected  
}
```

A key thing to keep in mind when working with collection interfaces is that *read-only collections aren't necessarily immutable*. If you're working with a variable that has a read-only interface type, this can be just one of the many references to the same collection. Other references can have a mutable interface type, as illustrated in [8.6](#).

**Figure 8.6.** Two different references, one read-only and one mutable, pointing to the same collection object. Code accessing `list` can't change the underlying collection, but may still have to deal with changes done by code working with the `mutableList`.



If one part of your code holds a reference to the collection which is mutable, then another part of your code holding a read-only "view" on that same collection can't rely on the fact that the collection isn't modified by the first part simultaneously. When the collection is modified while your code is working on it, it may lead to `ConcurrentModificationException` errors and other problems.

Therefore, it's essential to understand that *read-only collections aren't always thread-safe*: what your function may receive as a "view" on a collection may actually be a mutable collection under the hood. So, if you're working with data in a multithreaded environment, you need to ensure that your code properly synchronizes access to the data or uses data structures that support concurrent access.



#### Note

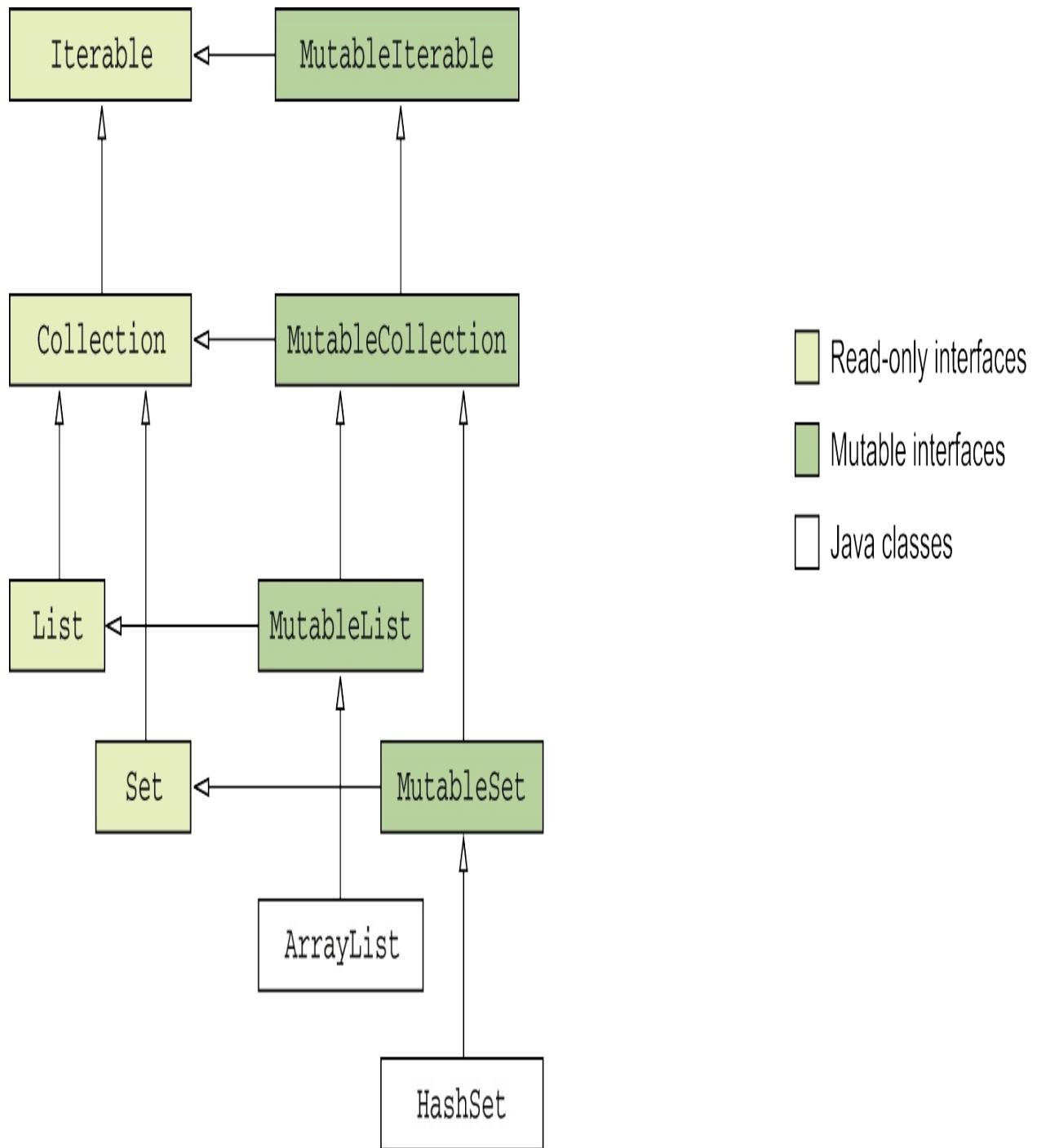
While immutable collections aren't available in the standard library, the `kotlinx.collections.immutable` library (<https://github.com/Kotlin/kotlinx.collections.immutable>) provides immutable collection interfaces and implementation prototypes for Kotlin.

How does the separation between read-only and mutable collections work? Didn't we say earlier that Kotlin collections are the same as Java collections? Isn't there a contradiction? Let's see what really happens here.

### 8.2.3 Kotlin collections and Java collections are deeply related

It's true that every Kotlin collection is an instance of the corresponding Java collection interface. No conversion is involved when moving between Kotlin and Java; there's no need for wrappers or copying data. But every Java collection interface has two *representations* in Kotlin: a read-only one and a mutable one, as you can see in [8.7](#).

**Figure 8.7. The hierarchy of the Kotlin collection interfaces. The Java classes `ArrayList` and `HashSet`, among others, extend Kotlin mutable interfaces.**



All collection interfaces shown in [8.7](#) are declared in Kotlin. The basic structure of the Kotlin read-only and mutable interfaces is parallel to the structure of the Java collection interfaces in the `java.util` package. In addition, each mutable interface extends the corresponding read-only interface. Mutable interfaces correspond directly to the interfaces in the `java.util` package, whereas the read-only versions lack all the mutating

methods.

[8.7](#) also contains the Java classes `java.util.ArrayList` and `java.util.HashSet` to show how Java standard classes are treated in Kotlin. Kotlin sees them as if they inherited from the Kotlin's `MutableList` and `MutableSet` interfaces, respectively. Other implementations from the Java collection library (`LinkedList`, `SortedSet`, and so on) aren't presented here, but from the Kotlin perspective they have similar supertypes. This way, you get both compatibility and clear separation of mutable and read-only interfaces.

In addition to the collections, the `Map` class (which doesn't extend `Collection` or `Iterable`) is also represented in Kotlin as two distinct versions: `Map` and `MutableMap`.

[8.1](#) shows the functions you can use to create collections of different types.

**Table 8.1. Collection-creation functions**

Collection type	Read-only	Mutable
List	<code>listOf</code> , <code>List</code>	<code>mutableListOf</code> , <code>MutableList</code> , <code>arrayListOf</code> , <code>buildList</code>
Set	<code>setOf</code>	<code>mutableSetOf</code> , <code>hashSetOf</code> , <code>linkedSetOf</code> , <code>sortedSetOf</code> , <code>buildSet</code>
Map	<code>mapOf</code>	<code>mutableMapOf</code> , <code>hashMapOf</code> , <code>linkedMapOf</code> , <code>sortedMapOf</code> , <code>buildMap</code>

---

Note that `setOf()` and `mapOf()` return instances of the `Set` and `Map` read-only interfaces, but that are mutable *under the hood*. (On the JVM, collections can be wrapped in a call to `Collections.unmodifiable` to make changes impossible. However, since this introduces indirection overhead, Kotlin doesn't do this for your collections automatically.) But you shouldn't rely on that: it's possible that a future version of Kotlin will use truly immutable implementation classes as return values of `setOf` and `mapOf`.

When you need to call a Java method and pass a collection as an argument, you can do so directly without any extra steps. For example, if you have a Java method that takes a `java.util.Collection` as a parameter, you can pass any `Collection` or `MutableCollection` value as an argument to that parameter.

This has important consequences with regard to mutability of collections. Because Java doesn't distinguish between read-only and mutable collections, Java code *can modify the collection* even if it's declared as a read-only `Collection` on the Kotlin side. The Kotlin compiler can't fully analyze what's being done to the collection in the Java code, and therefore there's no way for Kotlin to reject a call passing a read-only collection to Java code that modifies it. For example, the following two snippets of code form a compilable cross-language Kotlin/Java program:

```
/* Java */
// CollectionUtils.java
public class CollectionUtils {
    public static List<String> uppercaseAll(List<String> items) {
        for (int i = 0; i < items.size(); i++) {
            items.set(i, items.get(i).toUpperCase());
        }
        return items;
    }
}

// Kotlin
// collections.kt
fun printInUppercase(list: List<String>) { #1
    println(CollectionUtils.uppercaseAll(list)) #2
    println(list.first()) #3
}
```

```
fun main() {
    val list = listOf("a", "b", "c")
    printInUppercase(list)
    // [A, B, C]
    // A
}
```

Therefore, if you’re writing a Kotlin function that takes a collection and passes it to Java, *it’s your responsibility to use the correct type for the parameter*, depending on whether the Java code you’re calling will modify the collection.

Note that this caveat also applies to collections with non-null element types. If you pass such a collection to a Java method, the method can put a null value into it; there’s no way for Kotlin to forbid that or even to detect that it has happened without compromising performance. Because of that, you need to take special precautions when you pass collections to Java code that can modify them, to make sure the Kotlin types correctly reflect all the possible modifications to the collection.

Now, let’s take a closer look at how Kotlin deals with collections declared in Java code.

### 8.2.4 Collections declared in Java are seen as platform types in Kotlin

If you recall the discussion of nullability earlier in this chapter, you’ll remember that types defined in Java code are seen as *platform types* in Kotlin. For platform types, Kotlin doesn’t have the nullability information, so the compiler allows Kotlin code to treat them as either nullable or non-null. In the same way, variables of collection types declared in Java are also seen as platform types. A collection with a platform type is essentially a collection of unknown mutability—the Kotlin code can treat it as either read-only or mutable. Usually this doesn’t matter, because, in effect, all the operations you may want to perform just work.

The difference becomes important when you’re overriding or implementing a Java method that has a collection type in its signature. Here, as with platform

types for nullability, you need to decide which Kotlin type you’re going to use to represent a Java type coming from the method you’re overriding or implementing.

You need to make multiple choices in this situation, all of which will be reflected in the resulting parameter type in Kotlin:

- Is the collection nullable?
- Are the elements in the collection nullable?
- Will your method modify the collection?

To see the difference, consider the following cases. In the first example, a Java interface represents an object that processes text in a file.

**Listing 8.7. A Java interface with a collection parameter**

```
/* Java */
interface FileContentProcessor {
    void processContents(File path,
        byte[] binaryContents,
        List<String> textContents);
}
```

A Kotlin implementation of this interface needs to make the following choices:

- The list will be nullable, because some files are binary and their contents can’t be represented as text.
- The elements in the list will be non-null, because lines in a file are never null.
- The list will be read-only, because it represents the contents of a file, and those contents aren’t going to be modified.

Here’s how this implementation looks.

**Listing 8.8. Kotlin implementation of `FileContentProcessor`**

```
class FileIndexer : FileContentProcessor {
    override fun processContents(path: File,
        binaryContents: ByteArray?,
```

```
    textContents: List<String>?) {  
        // ...  
    }  
}
```

Contrast this with another interface. Here the implementations of the interface parse some data from a text form into a list of objects, append those objects to the output list, and report errors detected when parsing by adding the messages to a separate list.

**Listing 8.9. Another Java interface with a collection parameter**

```
/* Java */  
interface DataParser<T> {  
    void parseData(String input,  
                  List<T> output,  
                  List<String> errors);  
}
```

The choices in this case are different:

- `List<String>` will be non-null, because the callers always need to receive error messages.
- The elements in the list will be nullable, because not every item in the output list will have an associated error message.
- `List<String>` will be mutable, because the implementing code needs to add elements to it.

Here's how you can implement that interface in Kotlin.

**Listing 8.10. Kotlin implementation of `DataParser`**

```
class PersonParser : DataParser<Person> {  
    override fun parseData(input: String,  
                          output: MutableList<Person>,  
                          errors: MutableList<String?>) {  
        // ...  
    }  
}
```

Note how the same Java type—`List<String>`—is represented by two different Kotlin types: a `List<String>?` (nullable list of strings) in one case and a `MutableList<String?>` (mutable list of nullable strings) in the other. To make these choices correctly, you must know the exact contract the Java interface or class needs to follow. This is usually easy to understand based on what your implementation needs to do.

Now that we've discussed collections, it's time to look at arrays. As we've mentioned before, you should prefer using collections to arrays by default. But because many Java APIs still use arrays, we'll cover how to work with them in Kotlin.

### 8.2.5 Creating arrays of objects and primitive types for interoperability and performance reasons

You have already encountered arrays quite early in your Kotlin journey, because an array can be part of the signature of the Kotlin `main` function. Here's a reminder of how it looks:

**Listing 8.11. Using arrays**

```
fun main(args: Array<String>) {
    for (i in args.indices) { #1
        println("Argument $i is: ${args[i]}") #2
    }
}
```

An array in Kotlin is a class with a type parameter, and the element type is specified as the corresponding type argument.

To create an array in Kotlin, you have the following possibilities:

- The `arrayOf` function creates an array containing the elements specified as arguments to this function.
- The `arrayOfNulls` function creates an array of a given size containing `null` elements. Of course, it can only be used to create arrays where the element type is nullable.
- The `Array` constructor takes the size of the array and a lambda, and

initializes each array element by calling the lambda. This is how you can initialize an array with a non-null element type without passing each element explicitly.

As a simple example, here's how you can use the `Array` function to create an array of strings from "a" to "z".

**Listing 8.12. Creating an array of characters**

```
fun main() {  
    val letters = Array<String>(26) { i -> ('a' + i).toString() }  
    println(letters.joinToString(""))  
    // abcdefghijklmnopqrstuvwxyz  
}
```

The lambda takes the index of the array element and returns the value to be placed in the array at that index. Here you calculate the value by adding the index to the 'a' character and converting the result to a string. The array element type is shown for clarity; you can omit it in real code because the compiler can infer it:

```
fun main() {  
    val letters = Array(26) { i -> ('a' + i).toString() }  
    println(letters.joinToString())  
    // a, b, c, d, e, f, g, h, i, j  
}
```



**Note**

This type of construction isn't actually exclusive to arrays. Kotlin also provides `List` and `MutableList` functions that also instantiate their elements based on a size parameter and an initialization lambda.

Having said that, one of the most common cases for creating an array in Kotlin code is when you need to call a Java method that takes an array, or a Kotlin function with a `vararg` parameter. In those situations, you often have the data already stored in a collection, and you just need to convert it into an array. You can do this using the `toTypedArray` method.

### **Listing 8.13. Passing a collection to a vararg method**

```
fun main() {
    val strings = listOf("a", "b", "c")
    println("%s/%s/%s".format(*strings.toTypedArray())) #1
    // a/b/c
}
```

As with other types, *type arguments of array types always become object types*. Therefore, if you declare something like an `Array<Int>`, it will become an array of boxed integers (its Java type will be `java.lang.Integer[]`). If you need to create an array of values of a primitive type without boxing, you must use one of the specialized classes for arrays of primitive types.

To represent arrays of primitive types, Kotlin provides a number of separate classes, one for each primitive type. For example, an array of values of type `Int` is called `IntArray`. For other types, Kotlin provides `ByteArray`, `CharArray`, `BooleanArray`, and so on. All of these types are compiled to regular Java primitive type arrays, such as `int[]`, `byte[]`, `char[]`, and so on. Therefore, values in such an array are stored without boxing, in the most efficient manner possible.



#### **Note**

Just like other number type arrays that prevent boxing, Kotlin also allows you to create arrays of unsigned types, such as `UByteArray`, `UShortArray`, `UIIntArray`, and `ULongArray`. At the time of writing, unsigned arrays and operations on them are not yet stable.

To create an array of a primitive type, you have the following options:

- The constructor of the type takes a `size` parameter and returns an array initialized with default values for the corresponding primitive type (usually zeros).
- The factory function (`intArrayOf` for `IntArray`, and so on for other array types) takes a variable number of values as arguments and creates an array holding those values.
- Another constructor takes a `size` and a lambda used to initialize each

element.

Here's how the first two options work for creating an integer array holding five zeros:

```
val fiveZeros = IntArray(5)
val fiveZerosToo = intArrayOf(0, 0, 0, 0, 0)
```

Here's how you can use the constructor accepting a lambda:

```
fun main() {
    val squares = IntArray(5) { i -> (i+1) * (i+1) }
    println(squares.joinToString())
    // 1, 4, 9, 16, 25
}
```

Alternatively, if you have an array or a collection holding boxed values of a primitive type, you can convert them to an array of that primitive type using the corresponding conversion function, such as `toIntArray`.

Next, let's look at some of the things you can do with arrays. In addition to the basic operations (getting the array's length and getting and setting elements), the Kotlin standard library supports the same set of extension functions for arrays as for collections. All the functions you saw in chapter 6 (`filter`, `map`, and so on) work for arrays as well, including the arrays of primitive types. (Note that the return values of these functions are lists, not arrays.)

Let's see how to rewrite [8.11](#) using the `forEachIndexed` function and a lambda. The lambda passed to that function is called for each element of the array and receives two arguments, the index of the element and the element itself.

#### **Listing 8.14. Using `forEachIndexed` with an array**

```
fun main(args: Array<String>) {
    args.forEachIndexed { index, element ->
        println("Argument $index is: $element")
    }
}
```

Now you know how to use arrays in your code. Working with them is as simple as working with collections in Kotlin.

## 8.3 Summary

- Types representing basic numbers (such as `Int`) look and function like regular classes but are usually compiled to Java primitive types. Kotlin's unsigned number classes, that don't have an exact equivalent on the JVM, are transformed via inline classes to behave and perform like primitive types.
- Nullable primitive types (such as `Int?`) correspond to boxed primitive types in Java (such as `java.lang.Integer`).
- The `Any` type is a supertype of all other types and is analogous to Java's `Object`. `Unit` is an analogue of `void`.
- The `Nothing` type is used as a return type of functions that don't terminate normally.
- Types coming from Java are interpreted as platform types in Kotlin, allowing the developer to treat them as either nullable or non-null.
- Kotlin uses the standard Java classes for collections and enhances them with a distinction between read-only and mutable collections.
- You need to carefully consider nullability and mutability of parameters when you extend Java classes or implement Java interfaces in Kotlin.
- Kotlin's `Array` class looks like a regular generic class, but is compiled to a Java array.
- Arrays of primitive types are represented by special classes such as `IntArray`.

# 9 Operator overloading and other conventions

## This chapter covers

- Operator overloading
- Conventions: special-named functions supporting various operations
- Delegated properties

Kotlin has a number of features where specific language constructs are implemented by calling functions that you define in your own code. You already may be familiar with these types of constructs from Java, where objects that implement the `java.lang.Iterable` interface can be used in `for` loops, and objects that implement the `java.lang.AutoCloseable` interface can be used in `try-with-resources` statements.

In Kotlin, such features are tied to functions with specific names (and not bound to some special interfaces in the standard library, like they are in Java). For example, if your class defines a special method named `plus`, then, by convention, you can use the `+` operator on instances of this class. Because of that, in Kotlin we refer to this technique as *conventions*. In this chapter, we'll look at different conventions supported by Kotlin and how they can be used.

Kotlin uses the principle of conventions, instead of relying on types as Java does, because this allows developers to adapt existing Java classes to the requirements of Kotlin language features. Kotlin code can't modify third-party classes so that they would implement additional interfaces. On the other hand, defining new methods for a class is possible through the mechanism of extension functions. You can define any convention methods as extensions and thereby adapt any existing Java class without modifying its code.

As a running example in this chapter, we'll use a simple `Point` class, representing a point on a screen. Such classes are available in most UI frameworks, and you can easily adapt the definitions shown here to your

environment:

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

Let's begin by defining some arithmetic operators on the `Point` class.

## 9.1 Overloading arithmetic operators makes operations for arbitrary classes more convenient

The most straightforward example of the use of conventions in Kotlin is arithmetic operators. In Java, the full set of arithmetic operations can be used only with primitive types, and additionally the `+` operator can be used with `String` values. But these operations could be convenient in other cases as well. For example, if you're working with numbers through the `BigInteger` class, it would be more elegant to sum them using `+` than to call the `add` method explicitly. To add an element to a collection, you may want to use the `+=` operator. Kotlin allows you to do that, and in this section we'll show you how it works.

### 9.1.1 Plus, times, divide, and more: Overloading binary arithmetic operations

The first operation you're going to support is adding two points together. This operation sums up the points' X and Y coordinates. Here's how you can implement it.

**Listing 9.1. Defining the plus operator**

```
data class Point(val x: Int, val y: Int) {
    operator fun plus(other: Point): Point { #1
        return Point(x + other.x, y + other.y) #2
    }
}

fun main() {
    val p1 = Point(10, 20)
    val p2 = Point(30, 40)
    println(p1 + p2) #3
    // Point(x=40, y=60)
```

}

Note how you use the `operator` keyword to declare the `plus` function. All functions used to overload operators need to be marked with that keyword. This makes it explicit that you intend to use the function as an implementation of the corresponding convention and that you didn't define a function that accidentally had a matching name.

After you declare the `plus` function with the `operator` modifier, you can sum up your objects using just the `+` sign. Under the hood, the `plus` function is called as shown in [9.1](#).

**Figure 9.1. The `+` operator is transformed into a `plus` function call.**



As an alternative to declaring the operator as a member, you can define the operator as an extension function.

**Listing 9.2. Defining an operator as an extension function**

```
operator fun Point.plus(other: Point): Point {  
    return Point(x + other.x, y + other.y)  
}
```

The implementation is exactly the same. Future examples will use the extension function syntax because it's a common pattern to define convention extension functions for external library classes, and the same syntax will work nicely for your own classes as well.

Compared to some other languages, defining and using overloaded operators in Kotlin is simpler, because you can't define your own operators. For cases where you want to be able to use a function between two operands, i.e. `a myOp b`, Kotlin offers *infix functions*, which you have seen in [3.4](#), and will revisit again in **Chapter 13**. These allow you to leverage the main syntax benefit of custom operators—having operands on each side of the function call—without introducing arbitrary symbol combinations whose meaning you will have to painstakingly remember.

Kotlin has a limited set of operators that you can overload, and each one corresponds to the name of the function you need to define in your class. [9.1](#) lists all the binary operators you can define and the corresponding function names.

**Table 9.1. Overloadable binary arithmetic operators**

Expression	Function name
$a * b$	<code>times</code>
$a / b$	<code>div</code>
$a \% b$	<code>mod</code>
$a + b$	<code>plus</code>
$a - b$	<code>minus</code>

Operators for your own types always use the same precedence as the standard numeric types. For example, if you write  $a + b * c$ , the multiplication will always be executed before the addition, even if you've defined those operators yourself. The operators `*`, `/`, and `%` have the same precedence, which is higher than the precedence of the `+` and `-` operators.

### Operator functions and Java

Kotlin operators are easy to call from Java: because every overloaded operator is defined as a Kotlin function (with the `operator` modifier), you call them as regular functions using the full name. When you call Java from Kotlin, you can use the operator syntax for any methods with names matching the Kotlin conventions. Because Java doesn't define any syntax for marking operator functions, the requirement to use the `operator` modifier

doesn't apply, and the matching name and number of parameters are the only constraints. If a Java class defines a method with the behavior you need but gives it a different name, you can define an extension function with the correct name that would delegate to the existing Java method.

When you define an operator, you don't need to use the same types for the two operands. For example, let's define an operator that will allow you to scale a point by a certain number. You can use it to translate points between different coordinate systems.

**Listing 9.3. Defining an operator with different operand types**

```
operator fun Point.times(scale: Double): Point {
    return Point((x * scale).toInt(), (y * scale).toInt())
}

fun main() {
    val p = Point(10, 20)
    println(p * 1.5)
    // Point(x=15, y=30)
}
```

Note that Kotlin operators don't automatically support *commutativity* (the ability to swap the left and right sides of an operator). If you want users to be able to write `1.5 * p` in addition to `p * 1.5`, you need to define a separate operator for that: `operator fun Double.times(p: Point): Point`.

The return type of an operator function can also be different from either of the operand types. For example, you can define an operator to create a string by repeating a character a number of times.

**Listing 9.4. Defining an operator with a different return type**

```
operator fun Char.times(count: Int): String {
    return toString().repeat(count) #1
}

fun main() {
    println('a' * 3)
    // aaa
}
```

This operator takes a `char` as the left operand and an `Int` as the right operand and has `String` as the result type. Such combinations of operand and result types are perfectly acceptable.

Note that you can overload operator functions like regular functions: you can define multiple methods with different parameter types for the same method name.

#### No special operators for bitwise operations

Kotlin doesn't define any bitwise operators for standard number types (both signed as unsigned, as you have gotten to know them in [8.1](#)); consequently, it doesn't allow you to define them for your own types. Instead, it uses regular functions supporting the infix call syntax which you saw in [3.4](#). You can define similar functions that work with your own types.

Here's the full list of functions provided by Kotlin for performing bitwise operations:

- `shl`—Signed shift left
- `shr`—Signed shift right
- `ushr`—Unsigned shift right
- `and`—Bitwise and
- `or`—Bitwise or
- `xor`—Bitwise xor
- `inv`—Bitwise inversion

The following example demonstrates the use of some of these functions:

```
fun main() {  
    println(0x0F and 0xF0)  
    // 0  
    println(0x0F or 0xF0)  
    // 255  
    println(0x1 shl 4)  
    // 16  
}
```

Now let's discuss the operators like `+=` that merge two actions: assignment and the corresponding arithmetic operator.

## 9.1.2 Applying an operation and immediately assigning its value: Overloading compound assignment operators

Normally, when you define an operator such as plus, Kotlin supports not only the + operation but += as well. Operators such as +=, -=, and so on are called *compound assignment operators*. Here's an example:

```
fun main() {  
    var point = Point(1, 2)  
    point += Point(3, 4)  
    println(point)  
    // Point(x=4, y=6)  
}
```

This is the same as writing `point = point + Point(3, 4)`. Of course, that works only if the variable is mutable.

In some cases, it makes sense to define the += operation that would modify an object referenced by the variable on which it's used, but not reassign the reference. One such case is adding an element to a mutable collection:

```
fun main() {  
    val numbers = mutableListOf<Int>()  
    numbers += 42  
    println(numbers[0])  
    // 42  
}
```

If you define a function named plusAssign with the Unit return type and mark it with the operator keyword, Kotlin will call it when the += operator is used. Other binary arithmetic operators have similarly named counterparts: minusAssign, timesAssign, and so on.

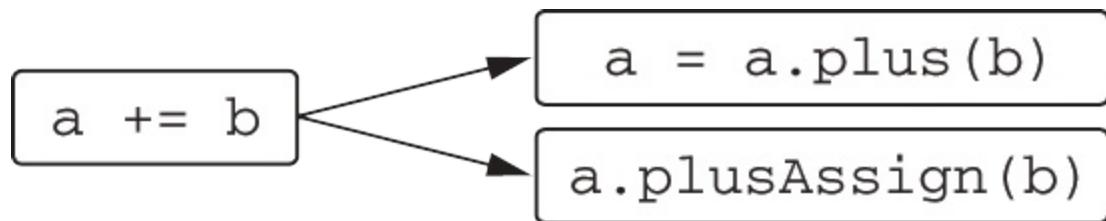
The Kotlin standard library defines a function plusAssign on a mutable collection, and the previous example uses it:

```
operator fun <T> MutableCollection<T>.plusAssign(element: T) {  
    this.add(element)  
}
```

When you write += in your code, theoretically both plus and plusAssign

functions can be called (see [9.2](#)). If this is the case, and both functions are defined and applicable, the compiler reports an error. One possibility to resolve it is replacing your use of the operator with a regular function call. Another is to replace a var with a val, so that the plusAssign operation becomes inapplicable. But in general, it's best to design new classes consistently: try not to add both plus and plusAssign operations at the same time. If your class is immutable, like Point in one of the earlier examples, you should provide only operations that return a new value (such as plus). If you design a mutable class, like a builder, provide only plusAssign and similar operations.

**Figure 9.2. The `+=` operator can be transformed into either the `plus` or the `plusAssign` function call.**



The Kotlin standard library supports both approaches for collections. The + and - operators always return a new collection. The += and -= operators work on mutable collections by modifying them in place, and on read-only collections by returning a modified copy. (This means += and -= can only be used with a read-only collection if the variable referencing it is declared as a var.) As operands of those operators, you can use either individual elements or other collections with a matching element type:

```
fun main() {
    val list = mutableListOf(1, 2)
    list += 3 #1
    val newList = list + listOf(4, 5) #2
    println(list)
    // [1, 2, 3]
    println(newList)
    [1, 2, 3, 4, 5]
}
```

So far, we've discussed overloading of *binary* operators—operators that are applied to two values, such as `a + b`. In addition, Kotlin allows you to

overload *unary* operators, which are applied to a single value, as in `-a`.

### 9.1.3 Operators with only one operand: Overloading unary operators

The procedure for overloading a unary operator is the same as you saw previously: declare a function (member or extension) with a predefined name, and mark it with the modifier operator. Let's look at an example.

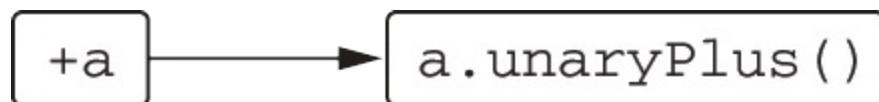
**Listing 9.5. Defining unary arithmetic operator**

```
operator fun Point.unaryMinus(): Point { #1
    return Point(-x, -y) #2
}

fun main() {
    val p = Point(10, 20)
    println(-p)
    // Point(x=-10, y=-20)
}
```

Functions used to overload unary operators don't take any arguments. As shown in [9.3](#), the unary plus operator works the same way. [9.2](#) lists all the unary operators you can overload.

**Figure 9.3. The unary + operator is transformed into a `unaryPlus` function call.**



**Table 9.2. Overloadable unary arithmetic operators**

Expression	Function name
<code>+a</code>	<code>unaryPlus</code>
<code>-a</code>	<code>unaryMinus</code>

!a	not
++a, a++	inc
--a, a--	dec

When you define the `inc` and `dec` functions to overload increment and decrement operators, the compiler automatically supports the same semantics for pre- and post-increment operators as for the regular number types. Consider the following example, which overloads the `++` operator for the `BigDecimal` class.

**Listing 9.6. Defining an increment operator**

```
import java.math.BigDecimal

operator fun BigDecimal.inc() = this + BigDecimal.ONE

fun main() {
    var bd = BigDecimal.ZERO
    println(bd++) #1
    // 0
    println(bd)
    // 1
    println(++bd) #2
    // 2
}
```

The postfix operation `++` first returns the current value of the `bd` variable and after that increases it, whereas the prefix operation works the other way round. The printed values are the same as you'd see if you used a variable of type `Int`, and you didn't need to do anything special to support this.

## 9.2 Overloading comparison operators make it easy to check relationships between objects

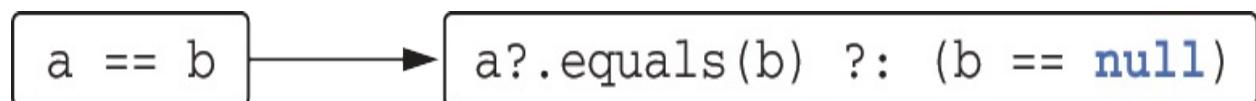
Just as with arithmetic operators, Kotlin lets you use comparison operators (`==`, `!=`, `>`, `<`, and so on) with any object, not just with primitive types. Instead of calling `equals` or `compareTo`, as in Java, you can use comparison operators directly, which is intuitive and concise. In this section, we'll look at the conventions used to support these operators.

### 9.2.1 Equality operators: `equals ("==")`

We touched on the topic of equality in section [Object equality: `equals\(\)`](#). You saw that using the `==` operator in Kotlin is translated into a call of the `equals` method. This is just one more application of the conventions principle we've been discussing.

Using the `!=` operator is also translated into a call of `equals`, with the obvious difference that the result is inverted. Note that unlike all other operators, `==` and `!=` can be used with nullable operands, because those operators check equality to `null` under the hood. The comparison `a == b` checks whether `a` isn't `null`, and, if it's not, calls `a.equals(b)` (see [9.4](#)). Otherwise the result is true only if both arguments are `null` references.

**Figure 9.4.** An equality check `==` is transformed into an `equals` call and a `null` check.



For the `Point` class, the implementation of `equals` is automatically generated by the compiler, because you've marked it as a data class ([4.3.2](#) explained the details). But if you did implement it manually, here's what the code could look like.

**Listing 9.7.** Implementing the `equals` method

```
class Point(val x: Int, val y: Int) {
    override fun equals(other: Any?): Boolean { #1
        if (other === this) return true #2
        if (other !is Point) return false #3
        return other.x == x && other.y == y #4
    }
}
```

```
fun main() {
    println(Point(10, 20) == Point(10, 20))
    // true
    println(Point(10, 20) != Point(5, 5))
    // true
    println(null == Point(1, 2))
    // false
}
```

You use the *identity equals* operator (`==`) to check whether the parameter to `equals` is the same object as the one on which `equals` is called. The identity equals operator does exactly the same thing as the `==` operator in Java: it checks that both of its arguments reference the same object (or have the same value, if they have a primitive type). Using this operator is a common optimization when implementing `equals`. Note that the `==` operator can't be overloaded.

The `equals` function is marked as `override`, because, unlike other conventions, the method implementing it is defined in the `Any` class (equality comparison is supported for all objects in Kotlin). That also explains why you don't need to mark it as `operator`: the base method in `Any` is marked as such, and the `operator` modifier on a method applies also to all methods that implement or override it. Also note that `equals` can't be implemented as an extension, because the implementation inherited from the `Any` class would always take precedence over the extension.

This example shows that using the `!=` operator is also translated into a call of the `equals` method. The compiler automatically negates the return value, so you don't need to do anything for this to work correctly.

What about other comparison operators?

### 9.2.2 Ordering operators: `compareTo ("<", ">", "<=" and ">=")`

In Java, classes can implement the `Comparable` interface in order to be used in algorithms that compare values, such as finding a maximum or sorting. The `compareTo` method defined in that interface is used to determine whether

one object is larger than another. But in Java, there's no shorthand syntax for calling this method. Only values of primitive types can be compared using `<` and `>`; all other types require you to write `element1.compareTo(element2)` explicitly.

Kotlin supports the same `Comparable` interface. But the `compareTo` method defined in that interface can be called by convention, and uses of comparison operators (`<`, `>`, `<=`, and `>=`) are translated into calls of `compareTo`, as shown in [9.5](#). The return type of `compareTo` has to be `Int`. The expression `p1 < p2` is equivalent to `p1.compareTo(p2) < 0`. Other comparison operators work exactly the same way.

**Figure 9.5. Comparison of two objects is transformed into comparing the result of the `compareTo` call with zero.**



Because there's no obviously right way to compare two-dimensional points with one another, let's use the good-old `Person` class to show how the method can be implemented. The implementation will use address book ordering (compare by last name, and then, if the last name is the same, compare by first name).

**Listing 9.8. Implementing the `compareTo` method**

```
class Person(  
    val firstName: String, val lastName: String  
) : Comparable<Person> {  
  
    override fun compareTo(other: Person): Int {  
        return compareValuesBy(this, other, #1  
            Person::lastName, Person::firstName)  
    }  
}  
  
fun main() {  
    val p1 = Person("Alice", "Smith")  
    val p2 = Person("Bob", "Johnson")  
    println(p1 < p2)  
    // false
```

```
}
```

In this case, you implement the `Comparable` interface so that the `Person` objects can be compared not only by Kotlin code but also by Java functions, such as the functions used to sort collections. Just as with `equals`, the `operator` modifier is applied to the function in the base interface, so you don't need to repeat the keyword when you override the function.

Note how you can use the `compareValuesBy` function from the Kotlin standard library to implement the `compareTo` method easily and concisely. This function receives a list of selector functions that calculate values to be compared. The function calls each selector in order for both objects and compares the return values. If the values are different, it returns the result of the comparison. If they're the same, it proceeds to the next selector function, or returns 0 if there are no more functions to call. These selectors can be passed as lambdas or, as you do here, as property references.

Note, however, that a direct implementation comparing fields by hand would be faster, although it would contain more code. As always, you should prefer the concise version and worry about performance only if you know the implementation will be called frequently.

All Java classes that implement the `Comparable` interface can be compared in Kotlin using the concise operator syntax:

```
fun main() {
    println("abc" < "bac")
    // true
}
```

You don't need to add any extensions to make that work.

## 9.3 Conventions used for collections and ranges

Some of the most common operations for working with collections are getting and setting elements by index, as well as checking whether an element belongs to a collection. All of these operations are supported via operator syntax: To get or set an element by index, you use the syntax `a[b]`

(called the *indexed access operator*). The `in` operator can be used to check whether an element is in a collection or range and also to iterate over a collection. You can add those operations for your own classes that act as collections. Let's now look at the conventions used to support those operations.

### 9.3.1 Accessing elements by index: the "get" and "set" conventions

You already know that in Kotlin, you can access the elements in a map similarly to how you access arrays in Java—via square brackets:

```
val value = map[key]
```

You can use the same operator to change the value for a key in a mutable map:

```
mutableMap[key] = newValue
```

Now it's time to see how this works. In Kotlin, the indexed access operator is one more convention. Reading an element using the indexed access operator is translated into a call of the `get` operator method, and writing an element becomes a call to `set`. The methods are already defined for the `Map` and `MutableMap` interfaces. Let's see how to add similar methods to your own class.

You'll allow the use of square brackets to reference the coordinates of the point: `p[0]` to access the X coordinate and `p[1]` to access the Y coordinate. Here's how to implement and use it.

#### **Listing 9.9. Implementing the get convention**

```
operator fun Point.get(index: Int): Int { #1
    return when(index) {
        0 -> x #2
        1 -> y
        else ->
            throw IndexOutOfBoundsException("Invalid coordinate $ #3
    }
}
```

```
fun main() {
    val p = Point(10, 20)
    println(p[1])
    // 20
}
```

All you need to do is define a function named `get` and mark it as operator. Once you do that, expressions like `p[1]`, where `p` has type `Point`, will be translated into calls to the `get` method, as shown in [9.6](#).

**Figure 9.6.** Access via square brackets is transformed into a `get` function call.



Note that the parameter of `get` can be any type, not just `Int`. For example, when you use the indexing operator on a map, the parameter type is the key type of the map, which can be an arbitrary type. You can also define a `get` method with multiple parameters. For example, if you're implementing a class to represent a two-dimensional array or matrix, you can define a method such as `operator fun get(rowIndex: Int, colIndex: Int)` and call it as `matrix[row, col]`. You can define multiple overloaded `get` methods with different parameter types, if your collection can be accessed with different key types.

In a similar way, you can define a function that lets you change the value at a given index using the bracket syntax. The `Point` class is immutable, so it doesn't make sense to define such a method for `Point`. Let's define another class to represent a mutable point and use that as an example.

#### **Listing 9.10. Implementing the set convention**

```
data class MutablePoint(var x: Int, var y: Int)

operator fun MutablePoint.set(index: Int, value: Int) { #1
    when(index) {
        0 -> x = value #2
        1 -> y = value
        else ->
            throw IndexOutOfBoundsException("Invalid coordinate $")
```

```

        }
    }

fun main() {
    val p = MutablePoint(10, 20)
    p[1] = 42
    println(p)
    // MutablePoint(x=10, y=42)
}

```

**Figure 9.7. Assignment through square brackets is transformed into a set function call.**



This example is also simple: to allow the use of the indexed access operator in assignments, you just need to define a function named `set`. The last parameter to `set` receives the value used on the right side of the assignment, and the other arguments are taken from the indices used inside the brackets, as you can see in [9.7](#).

### 9.3.2 Checking whether an object belongs to a collection: the "in" convention

One other operator supported by collections is the `in` operator, which is used to check whether an object belongs to a collection. The corresponding function is called `contains`. Let's implement it so that you can use the `in` operator to check whether a point belongs to a rectangle.

**Listing 9.11. Implementing the `in` convention**

```

data class Rectangle(val upperLeft: Point, val lowerRight: Point)

operator fun Rectangle.contains(p: Point): Boolean {
    return p.x in upperLeft.x until lowerRight.x && #1
        p.y in upperLeft.y until lowerRight.y #2
}

fun main() {
    val rect = Rectangle(Point(10, 20), Point(50, 50))
    println(Point(20, 30) in rect)
}

```

```

    // true
    println(Point(5, 5) in rect)
    // false
}

```

The object on the right side of `in` becomes the object on which the `contains` method is called, and the object on the left side becomes the argument passed to the method (see [9.8](#)).

In the implementation of `Rectangle.contains`, you use the `until` standard library function to build an *open range* and then you use the `in` operator on a range to check that a point belongs to it.

**Figure 9.8. The `in` operator is transformed into a `contains` function call.**



An *open range* is a range that doesn't include its ending point. For example, if you build a regular (closed) range using `10..20`, this range includes all numbers from 10 to 20, including 20. An open range `10 until 20` includes numbers from 10 to 19 but doesn't include 20. A rectangle class is usually defined in such a way that its bottom and right coordinates aren't part of the rectangle, so the use of open ranges is appropriate here.

### 9.3.3 Creating ranges from objects: The "rangeTo" convention

To create a range, you use the `..` syntax: for instance, `1..10` enumerates all the numbers from 1 to 10. You met ranges in [2.4.4](#), but now let's discuss the convention that helps create one. The `..` operator is a concise way to call the `rangeTo` function (see [9.9](#)).

**Figure 9.9. The `..` operator is transformed into a `rangeTo` function call.**



The `rangeTo` function returns a range. You can define this operator for your

own class. But if your class implements the `Comparable` interface, you don't need that: you can create a range of any comparable elements by means of the Kotlin standard library. The library defines the `rangeTo` function that can be called on any comparable element:

```
operator fun <T: Comparable<T>> T.rangeTo(that: T): ClosedRange<T
```

This function returns a range that allows you to check whether different elements belong to it.

As an example, let's build a range of dates using the `LocalDate` class (defined in the Java 8 standard library).

#### **Listing 9.12. Working with a range of dates**

```
import java.time.LocalDate

fun main() {
    val now = LocalDate.now()
    val vacation = now..now.plusDays(10) #1
    println(now.plusWeeks(1) in vacation) #2
    // true
}
```

The expression `now..now.plusDays(10)` is transformed into `now.rangeTo(now.plusDays(10))` by the compiler. The `rangeTo` function isn't a member of `LocalDate` but rather is an extension function on `Comparable`, as shown earlier.

The `rangeTo` operator has lower priority than arithmetic operators. But it's better to use parentheses for its arguments to avoid confusion:

```
fun main() {
    val n = 9
    println(0..(n + 1)) #1
    // 0..10
}
```

Also note that the expression `0..n.forEach {}` won't compile, because you have to surround a range expression with parentheses to call a method on it:

```
fun main() {
    (0..n).forEach { print(it) } #1
    // 0123456789
}
```

Now let's discuss how conventions allow you to iterate over a collection or a range.

### 9.3.4 Making it possible to loop over your types: The "iterator" convention

As we discussed in chapter 2, `for` loops in Kotlin use the same `in` operator as range checks. But its meaning is different in this context: it's used to perform iteration. This means a statement such as `for (x in list) { ... }` will be translated into a call of `list.iterator()`, on which the `hasNext` and `next` methods are then repeatedly called, just like in Java.

Note that in Kotlin, it's also a convention, which means the `iterator` method can be defined as an extension. That explains why it's possible to iterate over a regular string: the Kotlin standard library defines an extension function `iterator` on `CharSequence`, a superclass of `String`:

```
operator fun CharSequence.iterator(): CharIterator #1

fun main() {
    for (c in "abc") { }
}
```

You can define the `iterator` function as a method in your own classes, or as an extension function for third-party classes that you are using. For example, you could define an extension function that makes it possible to iterate over `LocalDate` objects. Because the `iterator` function should return an object implementing the `Iterator<LocalDate>` interface, you use an object declaration (as you've gotten to know it in [4.4.1](#)) to specify implementations for the `hasNext` and `next` functions expected by the interface:

**Listing 9.13. Implementing a date range iterator**

```
import java.time.LocalDate
```

```

operator fun ClosedRange<LocalDate>.iterator(): Iterator<LocalDat
    object : Iterator<LocalDate> { #1
        var current = start

        override fun hasNext() =
            current <= endInclusive #2

        override fun next(): LocalDate {
            val thisDate = current
            current = current.plusDays(1) #3
            return thisDate #4
        }
    }

fun main() {
    val newYear = LocalDate.ofYearDay(2042, 1)
    val daysOff = newYear.minusDays(1)..newYear
    for (dayOff in daysOff) { println(dayOff) } #5
    // 2041-12-31
    // 2042-01-01
}

```

Note how you define the `iterator` method on a custom range type: you use `LocalDate` as a type argument. The `rangeTo` library function, shown in the previous section, returns an instance of `ClosedRange`, and the `iterator` extension on `ClosedRange<LocalDate>` allows you to use an instance of the range in a `for` loop.

## 9.4 Making destructuring declarations possible with component functions

When we discussed data classes in [4.3.2](#), we mentioned that some of their features would be revealed later. Now that you’re familiar with the principle of conventions, we can look at the final feature: *destructuring declarations*. This feature allows you to unpack a single composite value and use it to initialize several separate local variables.

Here’s how it works:

```

fun main() {
    val p = Point(10, 20)
    val (x, y) = p #1

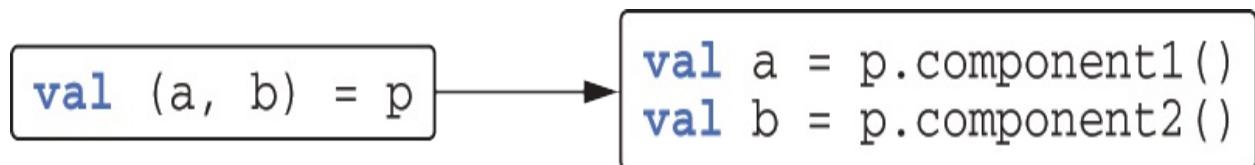
```

```
    println(x)
    // 10
    println(y)
    // 20
}
```

A destructuring declaration looks like a regular variable declaration, but it has multiple variables grouped in parentheses.

Under the hood, the destructuring declaration once again uses the principle of conventions. To initialize each variable in a destructuring declaration, a function named `componentN` is called, where `N` is the position of the variable in the declaration. In other words, the previous example would be transformed as shown in [9.10](#).

**Figure 9.10. Destructuring declarations are transformed into `componentN` function calls.**



For a data class, the compiler generates a `componentN` function for every property declared in the primary constructor. The following example shows how you can declare these functions manually for a non-data class:

```
class Point(val x: Int, val y: Int) {
    operator fun component1() = x
    operator fun component2() = y
}
```

One of the main use cases where destructuring declarations are helpful is returning multiple values from a function. If you need to do that, you can define a data class to hold the values you need to return and use it as the return type of the function. The destructuring declaration syntax makes it easy to unpack and use the values after you call the function. To demonstrate, let's write a simple function to split a filename into a name and an extension.

**Listing 9.14. Using a destructuring declaration to return multiple values**

```

data class NameComponents(val name: String, #1
                        val extension: String)

fun splitFilename(fullName: String): NameComponents {
    val result = fullName.split('.', limit = 2)
    return NameComponents(result[0], result[1]) #2
}

fun main() {
    val (name, ext) = splitFilename("example.kt") #3
    println(name)
    // example
    println(ext)
    // kt
}

```

You can improve this example even further if you note that `componentN` functions are also defined on arrays and collections. This is useful when you’re dealing with collections of a known size—and this is such a case, with `split` returning a list of two elements.

**Listing 9.15. Using a destructuring declaration with a collection**

```

data class NameComponents(
    val name: String,
    val extension: String)

fun splitFilename(fullName: String): NameComponents {
    val (name, extension) = fullName.split('.', limit = 2)
    return NameComponents(name, extension)
}

```

Of course, it’s not possible to define an infinite number of such `componentN` functions so the syntax would work with an arbitrary number of items, but that wouldn’t be useful, either. The standard library allows you to use this syntax to access the first five elements of a container.

A simpler way to return multiple values from a function is to use the `Pair` and `Triple` classes from the standard library. While this may require less code than defining your own class, you’re also giving up valuable expressiveness in your code, because `Pair` and `Triple` don’t make it clear what is contained in the returned object.

## 9.4.1 Destructuring declarations and loops

Destructuring declarations work not only as top-level statements in functions but also in other places where you can declare variables—for example, in loops. One good use for that is enumerating entries in a map. Here's a small example using this syntax to print all entries in a given map.

**Listing 9.16. Using a destructuring declaration to iterate over a map**

```
fun printEntries(map: Map<String, String>) {  
    for ((key, value) in map) { #1  
        println("$key -> $value")  
    }  
}  
  
fun main() {  
    val map = mapOf("Oracle" to "Java", "JetBrains" to "Kotlin")  
    printEntries(map)  
    // Oracle -> Java  
    // JetBrains -> Kotlin  
}
```

This simple example uses two Kotlin conventions: one to iterate over an object and another to destructure declarations. The Kotlin standard library contains an extension function `iterator` on a map that returns an iterator over map entries. Thus, unlike Java, you can iterate over a map directly. It also contains extensions functions `component1` and `component2` on `Map.Entry`, returning its key and value, respectively. In effect, the previous loop is translated to the equivalent of the following code:

```
for (entry in map.entries) {  
    val key = entry.component1()  
    val value = entry.component2()  
    // ...  
}
```

You can also use destructuring declarations when a lambda receives a composite value like a data class or a map. In this example, you are yet again printing all entries in a given map, but use the `.forEach` function which you already got to know in [5.1.4](#):

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

These examples again illustrate the importance of extension functions to support Kotlin's conventions.

## 9.4.2 Ignoring destructured values with "\_"

When you're destructuring an object with many components, there's a chance you might not actually need all of them. In this example, you're destructuring a `Person` class, but really only use the `firstName` and `age` fields:

**Listing 9.17. Intro person**

```
data class Person(
    val firstName: String,
    val lastName: String,
    val age: Int,
    val city: String,
)

fun introducePerson(p: Person) {
    val (firstName, lastName, age, city) = p
    println("This is $firstName, aged $age.")
}
```

In this case, declaring a local `lastName` and `city` variable doesn't provide any value for our code. Rather, it clutters the body of the function with unused variables—something that is generally best avoided.

Since we're not forced to destructure the whole object, we can leave trailing destructuring declarations (in this case, `city`) out of the destructuring declaration. Instead, you only destructure the first three elements:

```
val (firstName, lastName, age) = p
```

To get rid of the `lastName` declaration, you have to take a slightly different route. Were we to just remove it (leaving us with `(firstName, age)`), we would falsely assign the contents of `Person.lastName` to the `age` variable

(remember that under the hood, this destructure declaration only calls the `component1` and `component2` functions, regardless of the name you give them). To deal with this case, Kotlin allows you to assign unused declarations during destructure by assigning them to the reserved `_` character.

Equipped with this knowledge, you can make the implementation for `introducePerson` more concise—renaming `lastName` to `_`, and removing `city` during the destructure entirely:

```
fun introducePerson(p: Person) {  
    val (firstName, _, age) = p #1  
    println("This is $firstName, aged $age.")  
}
```

### Limitations and drawbacks of destructure in Kotlin

Kotlin's implementation of destructure declarations is *positional*—that means, the result of a destructure operation depends entirely on the positions of the arguments. For the `Person` data class from [9.17](#), this means that variables during destructure will always be assigned the values in the same order as they appear in the constructor:

```
val (firstName, lastName, age, city) = p
```

The names of the variables to which the result of the destructure is assigned do not matter—because destructure declarations iterate through the `componentN` functions one after the other, this code works just as well:

```
val (f, l, a, c) = p
```

This can lead to subtle problems when during refactoring, you change the order of properties in a data class:

```
data class Person(  
    val lastName: String, #1  
    val firstName: String, #1  
    val age: Int,  
    val city: String,  
)
```

Now, the same code snippet from above still works, but falsely assigns the

value of `lastName` to `firstName`, and vice versa:

```
val (firstName, lastName, age, city) = p
```

This behavior means destructuring declarations are best only used for small container classes (such as key-value pairs or index-value pairs), or classes that are very unlikely to change in the future. They should be avoided for more complex entities.

A potential solution to this issue is the introduction of *name-based destructuring*, a topic that at the time of writing is being considered for Kotlin's value classes (<http://mng.bz/v17r>), which are planned to be added in a future version of Kotlin.

## 9.5 Reusing property accessor logic: delegated properties

To conclude this chapter, let's look at one more feature that relies on conventions and is one of the most unique and powerful in Kotlin: *delegated properties*. This feature lets you easily implement properties that work in a more complex way than storing values in backing fields, without duplicating the logic in each accessor. For example, properties can store their values in database tables, in a browser session, in a map, and so on.

The foundation for this feature is *delegation*: a design pattern where an object, instead of performing a task, delegates that task to another helper object. The helper object is called a *delegate*. You saw this pattern earlier, in [4.3.3](#), when we were discussing class delegation. Here this pattern is applied to a property, which can also delegate the logic of its accessors to a helper object. You could implement that by hand—or use a better solution: take advantage of Kotlin's language support. You'll see examples for both in a moment, but first, let's have a look at a general explanation.

### 9.5.1 Basic syntax and inner workings of delegated properties

The general syntax of a delegated property is this:

```
var p: Type by Delegate()
```

The property `p` delegates the logic of its accessors to another object: in this case, a new instance of the `Delegate` class. The object is obtained by evaluating the expression following the `by` keyword, which can be anything that satisfies the rules of the convention for property delegates.

Let's take a look at what happens under the hood for a class that defines a delegated property, such as this:

```
class Foo {  
    var p: Type by Delegate()  
}
```

The compiler creates a hidden helper property, initialized with the instance of the delegate object, to which the initial property `p` delegates. For simplicity, let's call it `delegate`:

```
class Foo {  
    private val delegate = Delegate() #1  
  
    var p: Type #2  
        set(value: Type) = delegate.setValue(..., value)  
        get() = delegate.getValue(...)  
}
```

By convention, the `Delegate` class must have `getValue` and `setValue` operator functions, although the latter is required only for mutable properties (i.e. when defining `var delegate = ...`). Additionally, they can (but don't have to) also provide an implementation for the `provideDelegate` function, in which you can perform validation logic or change the way the delegate is instantiated when it is first created. As usual, they can be implemented as members or extensions. To simplify the explanation, we omit their parameters; the exact signatures will be covered later in this chapter. In a simple form, the `Delegate` class might look like the following:

```
class Delegate {  
    operator fun getValue(...) { ... } #1  
  
    operator fun setValue(..., value: Type) { ... } #2  
  
    operator fun provideDelegate(): Delegate { ... } #3
```

```
}
```

```
class Foo {
    var p: Type by Delegate() #4
}

fun main() {
    val foo = Foo() #5
    val oldValue = foo.p #6
    foo.p = newValue #7
}
```

You use `foo.p` as a regular property, but under the hood the methods on the helper property of the `Delegate` type are called. To investigate how this mechanism is used in practice, we'll begin by looking at one example of the power of delegated properties: library support for lazy initialization. Afterward, we'll explore how you can define your own delegated properties and when this is useful.

## 9.5.2 Using delegated properties: lazy initialization and "by `lazy()`"

*Lazy initialization* is a common pattern that entails creating part of an object on demand, when it's accessed for the first time. This is helpful when the initialization process consumes significant resources and the data isn't always required when the object is used.

For example, consider a `Person` class that lets you access a list of the emails written by a person. The emails are stored in a database and take a long time to access. You want to load the emails on first access to the property and do so only once. Let's say you have the following function `loadEmails`, which retrieves the emails from the database:

```
class Email { /*...*/ }
fun loadEmails(person: Person): List<Email> {
    println("Load emails for ${person.name}")
    return listOf(/*...*/)
}
```

Here's how you can implement lazy loading using an additional `_emails`

property that stores null before anything is loaded and the list of emails afterward. The emails property itself uses a custom accessor as you got to know them in [2.2.2](#):

**Listing 9.18. Implementing lazy initialization using a backing property**

```
class Person(val name: String) {  
    private var _emails: List<Email>? = null #1  
  
    val emails: List<Email>  
        get() {  
            if (_emails == null) {  
                _emails = loadEmails(this) #2  
            }  
            return _emails!! #3  
        }  
}  
  
fun main() {  
    val p = Person("Alice")  
    p.emails #4  
    // Load emails for Alice  
    p.emails  
}
```

Here you use the so-called *backing property* technique. You have one property, `_emails`, which stores the value, and another, `emails`, which provides read access to it. You need to use two properties because the properties have different types: `_emails` is nullable, whereas `emails` is non-null. Their naming follows a simple convention: When your class has two properties representing the same concept, the private property is prefixed with an underscore (`_emails`), while the public property has no prefix (`emails`).

This technique can be used fairly often, so it's worth getting familiar with it.

But the code is somewhat cumbersome: imagine how much longer it would become if you had several lazy properties. What's more, it doesn't always work correctly: the implementation isn't thread-safe. If two threads both access the `emails` property, there's no mechanism in place to prevent the expensive `loadEmails` function from being called twice. At best, this only

wastes some resources, but at worst, you end up with an inconsistent state in your application. Surely Kotlin provides a better solution.

The code becomes much simpler with the use of a delegated property, which can encapsulate both the backing property used to store the value and the logic ensuring that the value is initialized only once. The delegate you can use here is returned by the `lazy` standard library function.

**Listing 9.19. Implementing lazy initialization using a delegated property**

```
class Person(val name: String) {  
    val emails by lazy { loadEmails(this) }  
}
```

The `lazy` function returns an object that has a method called `getValue` with the proper signature, so you can use it together with the `by` keyword to create a delegated property. The argument of `lazy` is a lambda that it calls to initialize the value. The `lazy` function is thread-safe by default; and if you need to, you can specify additional options to tell it which lock to use or to bypass the synchronization entirely if the class is never used in a multithreaded environment.

In the next section, we'll dive into details of how the mechanism of delegated properties works and discuss the conventions in play here.

### 9.5.3 Implementing your own delegated properties

To see how delegated properties are implemented, let's take another example: the task of notifying listeners when a property of an object changes. This is useful in many different cases: for example, when objects are presented in a UI and you want to automatically update the UI when the objects change.

This is typically called an "observable". Let's see how we could implement it in Kotlin. First, let's look at a variant that doesn't use delegated properties. Then, let's refactor the code to using delegated properties.

The `Observable` class manages a list of `Observers`. When `notifyObservers` is called, it calls the `onChange` function for each `Observer` with the old and

new property values. An `Observer` only needs to provide an implementation for this `onChange` method, so it would be suitable to use a functional interface as you've seen them in [5.3](#):

```
fun interface Observer {
    fun onChange(name: String, oldValue: Any?, newValue: Any?)
}

open class Observable {
    val observers = mutableListOf<Observer>()
    fun notifyObservers(propName: String, oldValue: Any?, newValue: Any?) {
        for (obs in observers) {
            obs.onChange(propName, oldValue, newValue)
        }
    }
}
```

Now let's write a `Person` class. You'll define a read-only property (the person's name, which typically doesn't change) and two writable properties: the age and the salary. The class will notify its observers when either the age or the salary of the person is changed.

**Listing 9.20. Implementing observer notifications for changed properties manually**

```
class Person(val name: String, age: Int, salary: Int): Observable {
    var age: Int = age
        set(newValue) {
            val oldValue = field #1
            field = newValue
            notifyObservers( #2
                "age", oldValue, newValue
            )
        }

    var salary: Int = salary
        set(newValue) {
            val oldValue = field
            field = newValue
            notifyObservers(
                "salary", oldValue, newValue
            )
        }
}

fun main() {
```

```

val p = Person("Seb", 28, 1000)
p.observers += Observer { propName, oldValue, newValue -> #3
    println(
        """
            Property $propName changed from $oldValue to $newValue
        """.trimIndent()
    )
}
p.age = 29
// Property age changed from 28 to 29!
p.salary = 1500
// Property salary changed from 1000 to 1500!
}

```

Note how this code uses the `field` identifier to access the *backing field* of the `age` and `salary` properties, as we discussed in [4.2.4](#).

There's quite a bit of repeated code in the setters. Let's try to extract a class that will store the value of the property and fire the necessary notification.

**Listing 9.21. Implementing observer notifications for changed properties with a helper class**

```

class ObservableProperty(val propName: String, var propValue: Int)
    fun getValue(): Int = propValue
    fun setValue(newValue: Int) {
        val oldValue = propValue
        propValue = newValue
        observable.notifyObservers(propName, oldValue, newValue)
    }
}

class Person(val name: String, age: Int, salary: Int): Observable
    val _age = ObservableProperty("age", age, this)
    var age: Int
        get() = _age.getValue()
        set(newValue) {
            _age.setValue(newValue)
        }

    val _salary = ObservableProperty("salary", age, this)
    var salary: Int
        get() = _salary.getValue()
        set(newValue) {
            _salary.setValue(newValue)
        }

```

```
}
```

Now you're close to understanding how delegated properties work in Kotlin. You've created a class that stores the value of the property and automatically notifies observers when it's modified. You removed the duplication in the logic, but instead quite a bit of boilerplate is required to create the `ObservableProperty` instance for each property and to delegate the getter and setter to it. Kotlin's delegated property feature lets you get rid of that boilerplate. But before you can do that, you need to change the signatures of the `ObservableProperty` methods to match those required by Kotlin conventions.

**Listing 9.22. `ObservableProperty` as a property delegate**

```
import kotlin.reflect.KProperty

class ObservableProperty(var propValue: Int, val observable: Obse
operator fun getValue(thisRef: Any?, prop: KProperty<*>): Int =
operator fun setValue(thisRef: Any?, prop: KProperty<*>, newVal
    val oldValue = propValue
    propValue = newValue
    observable.notifyObservers(prop.name, oldValue, newValue)
}
}
```

Compared to the previous version, this code has the following changes:

- The `getValue` and `setValue` functions are now marked as `operator`, as required for all functions used through conventions.
- You add two parameters to those functions: one to receive the instance for which the property is get or set (`thisRef`), and the second to represent the property itself (`prop`). The property is represented as an object of type `KProperty`. We'll look at it in more detail in **Chapter 12**; for now, all you need to know is that you can access the name of the property as `KProperty.name`.
- You remove the `name` property from the primary constructor because you can now access the property name through `KProperty`.

You can finally use the magic of Kotlin's delegated properties. See how

much shorter the code becomes?

**Listing 9.23. Using delegated properties for making properties observable**

```
class Person(val name: String, age: Int, salary: Int) : Observable
    var age by ObservableProperty(age, this)
    var salary by ObservableProperty(salary, this)
}
```

Through the `by` keyword, the Kotlin compiler does automatically what you did manually in the previous version of the code. Compare this code to the previous version of the `Person` class: the generated code when you use delegated properties is very similar. The object to the right of `by` is called the *delegate*. Kotlin automatically stores the delegate in a hidden property and calls `getValue` and `setValue` on the delegate when you access or modify the main property.

Instead of implementing the observable property logic by hand, you can use the Kotlin standard library. It turns out the standard library already contains its own `ObservableProperty` class. However, the standard library class doesn't know anything about the `Observable` interface that you defined yourself earlier, so you need to pass it a lambda that tells it how to report the changes in the property value. Here's how you can do that.

**Listing 9.24. Using `Delegates.observable` to implement property change notification**

```
import kotlin.properties.Delegates

class Person(val name: String, age: Int, salary: Int) : Observable
    private val onChange = { property: KProperty<*>, oldValue: Any
        notifyObservers(property.name, oldValue, newValue)
    }

    var age by Delegates.observable(age, onChange)
    var salary by Delegates.observable(salary, onChange)
}
```

The expression to the right of `by` doesn't have to be a new instance creation. It can also be a function call, another property, or any other expression, as long as the value of this expression is an object on which the compiler can call `getValue` and `setValue` with the correct parameter types. As with other

conventions, `getValue` and `setValue` can be either methods declared on the object itself or extension functions.

Note that to keep the examples simple, we've only shown you how to work with delegated properties of type `Int`. The delegated-properties mechanism is fully generic and works with any other type, too.

### 9.5.4 Delegated-property are translated to hidden properties with custom accessors

Let's summarize the rules for how delegated properties work. Suppose you have a class with a delegated property:

```
class C {  
    var prop: Type by MyDelegate()  
}  
  
val c = C()
```

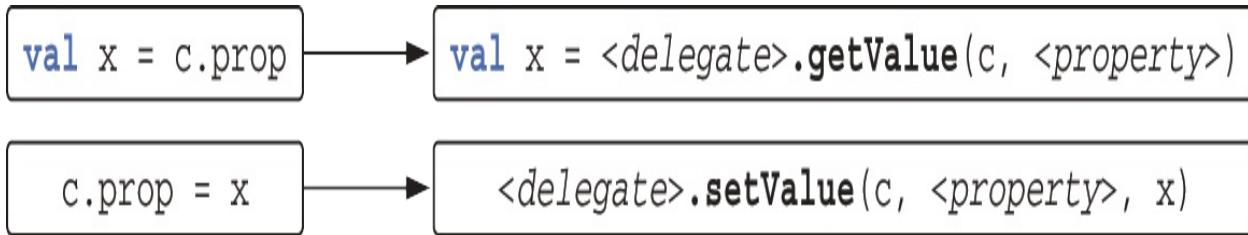
The instance of `MyDelegate` will be stored in a hidden property, which we'll refer to as `<delegate>`. The compiler will also use an object of type `KProperty` to represent the property. We'll refer to this object as `<property>`.

The compiler generates the following code:

```
class C {  
    private val <delegate> = MyDelegate()  
  
    var prop: Type  
        get() = <delegate>.getValue(this, <property>)  
        set(value: Type) = <delegate>.setValue(this, <property>, v  
}
```

Thus, inside every property accessor, the compiler generates calls to the corresponding `getValue` and `setValue` methods, as shown in [9.11](#).

**Figure 9.11.** When you access a property, the `getValue` and `setValue` functions on `<delegate>` are called.



The mechanism is fairly simple, yet it enables many interesting scenarios. You can customize where the value of the property is stored (in a map, in a database table, or in the cookies of a user session) and also what happens when the property is accessed (to add validation, change notifications, and so on). All of this can be accomplished with compact code. Let's look at one more use for delegated properties in the standard library and then see how you can use them in your own frameworks.

### 9.5.5 Accessing dynamic attributes by delegating to maps

Another common pattern where delegated properties come into play is objects that have a dynamically defined set of attributes associated with them. (Other languages, such as C#, call such objects *expando objects*.) For example, consider a contact-management system that allows you to store arbitrary information about your contacts. Each person in the system has a few required properties (such as a name) that are handled in a special way, as well as any number of additional attributes that can be different for each person (youngest child's birthday, for example).

One way to implement such a system is to store all the attributes of a person in a map and provide properties for accessing the information that requires special handling. Here's an example.

**Listing 9.25. Defining a property that stores its value in a map**

```

class Person {
    private val _attributes = mutableMapOf<String, String>()

    fun setAttribute(attrName: String, value: String) {
        _attributes[attrName] = value
    }

    var name: String

```

```

        get() = _attributes["name"]!! #1
        set(value) {
            _attributes["name"] = value #2
        }
    }

fun main() {
    val p = Person()
    val data = mapOf("name" to "Seb", "company" to "JetBrains")
    for ((attrName, value) in data)
        p.setAttribute(attrName, value)
    println(p.name)
    // Seb
    p.name = "Sebastian"
    println(p.name)
    // Sebastian
}

```

Here you use a generic API to load the data into the object (in a real project, this could be JSON deserialization or something similar) and then a specific API to access the value of one property. Changing this to use a delegated property is trivial; you can put the map directly after the by keyword.

#### **Listing 9.26. Using a delegated property which stores its value in a map**

```

class Person {
    private val _attributes = mutableMapOf<String, String>()

    fun setAttribute(attrName: String, value: String) {
        _attributes[attrName] = value
    }

    var name: String by _attributes #1
}

```

This works because the standard library defines `getValue` and `setValue` extension functions on the standard `Map` and `MutableMap` interfaces. The name of the property is automatically used as the key to store the value in the map. As in [9.25](#), `p.name` hides the call of `_attributes.getValue(p, prop)`, which in turn is implemented as `_attributes[prop.name]`.

## **9.5.6 How a real-life framework might use delegated properties**

Changing the way the properties of an object are stored and modified is extremely useful for framework developers. This section shows a an example case of how delegated properties improves framework development and usage, and dives into the details of it works.

Let's say your database contains the table `Users` with two columns: name of string type and age of integer type. You can define the classes `Users` and `User` in Kotlin. Then all the user entities stored in the database can be loaded and changed in Kotlin code via instances of the `User` class.

**Listing 9.27. Accessing database columns using delegated properties**

```
object Users : IdTable() { #1
    val name = varchar("name", length = 50).index() #2
    val age = integer("age")
}

class User(id: EntityID) : Entity(id) { #3
    var name: String by Users.name #4
    var age: Int by Users.age
}
```

The `Users` object describes a database table; it's declared as an object because it describes the table as a whole, so you only need one instance of it. Properties of the object represent columns of the table.

The `Entity` class, the superclass of `User`, contains a mapping of database columns to their values for the entity. The properties for the specific `User` have the values `name` and `age` specified in the database for this user.

Using the framework is especially convenient because accessing the property automatically retrieves the corresponding value from the mapping in the `Entity` class, and modifying it marks the object as dirty so that it can be saved to the database when needed. You can write `user.age += 1` in your Kotlin code, and the corresponding entity in the database will be automatically updated.

Now you know enough to understand how a framework with such an API can be implemented. Each of the entity attributes (`name`, `age`) is implemented as a delegated property, using the column object (`Users.name`, `Users.age`) as the

delegate:

```
class User(id: EntityID) : Entity(id) {  
    var name: String by Users.name #1  
    var age: Int by Users.age  
}
```

Let's look at the explicitly specified types of columns:

```
object Users : IdTable() {  
    val name: Column<String> = varchar("name", 50).index()  
    val age: Column<Int> = integer("age")  
}
```

For the `Column` class, the framework defines the `getValue` and `setValue` methods, satisfying the Kotlin convention for delegates:

```
operator fun <T> Column<T>.getValue(o: Entity, desc: KProperty<*>  
{  
    // retrieve the value from the database  
}  
operator fun <T> Column<T>.setValue(o: Entity, desc: KProperty<*>  
{  
    // update the value in the database  
}
```

You can use the `Column` property (`Users.name`) as a delegate for a delegated property (`name`). When you write `user.age += 1` in your code, the code will perform something similar to `user.ageDelegate.setValue(user.ageDelegate.getValue() + 1)` (omitting the parameters for the property and object instances). The `getValue` and `setValue` methods take care of retrieving and updating the information in the database.

The full implementation of the classes in this example can be found in the source code for the Exposed framework (<https://github.com/JetBrains/Exposed>). We'll return to this framework in **Chapter 13**, to explore the DSL design techniques used there.

## 9.6 Summary

- Kotlin allows you to overload some of the standard mathematical operations by defining functions with the corresponding names. You can't define your own operators, but you can use infix functions as a more expressive alternative.
- You can use comparison operators (==, !=, >, <, and so on) with any object. They are mapped to calls of the equals and compareTo methods.
- By defining functions named get, set, and contains, you can support the [ ] and in operators to make your class similar to Kotlin collections.
- Creating ranges and iterating over collections and arrays also work through conventions.
- Destructuring declarations let you initialize multiple variables by unpacking a single object, which is handy for returning multiple values from a function. They work with data classes automatically, and you can support them for your own classes by defining functions named componentN.
- Delegated properties allow you to reuse logic controlling how property values are stored, initialized, accessed, and modified, which is a powerful tool for building frameworks.
- The lazy standard library function provides an easy way to implement lazily initialized properties.
- The Delegates.observable function lets you add an observer of property changes.
- Delegated properties can use any map as a property delegate, providing a flexible way to work with objects that have variable sets of attributes.

# 10 Higher-order functions: lambdas as parameters and return values

## This chapter covers

- Function types
- Higher-order functions and their use for structuring code
- Inline functions
- Non-local returns and labels
- Anonymous functions

You were introduced to lambdas in [5](#), where we explored the general concept, and dove deeper into the standard library functions that use lambdas in [6](#). Lambdas are a great tool for building abstractions, and of course their power isn't restricted to collections and other classes in the standard library. In this chapter, you'll learn how to create *higher-order functions*—your own functions that take lambdas as arguments or return them. You'll see how higher-order functions can help simplify your code, remove code duplication, and build nice abstractions. You'll also become acquainted with *inline functions*—a powerful Kotlin feature that removes the performance overhead associated with using lambdas and enables more flexible control flow within lambdas.

## 10.1 Declaring functions that return or receive other functions: higher-order functions

The key new idea of this chapter is the concept of *higher-order functions*. By definition, a higher-order function is a function that takes another function as an argument or returns one. In Kotlin, functions can be represented as values either by using lambdas or via function references. Therefore, a higher-order function is any function to which you can pass a lambda or a function reference as an argument, or a function which returns one, or both. For example, the `filter` standard library function takes a predicate function as an

argument and is therefore a higher-order function:

```
list.filter { x > 0 }
```

In [6](#), you saw many other higher-order functions declared in the Kotlin standard library: `map`, `with`, and so on. Now you'll learn how you can declare such functions in your own code. To do this, you must first be introduced to *function types*.

### 10.1.1 Function types specify the parameter types and return values of a lambda

In order to declare a function that takes a lambda as an argument, you need to know how to declare the type of the corresponding parameter. Before we get to this, let's look at a simpler case and store a lambda in a local variable. You already saw how you can do this without declaring the type, relying on Kotlin's type inference:

```
val sum = { x: Int, y: Int -> x + y }
val action = { println(42) }
```

In this case, the compiler infers that both the `sum` and `action` variables have function types.

**Figure 10.1. Optional inlay hints in IntelliJ IDEA and Android Studio help visualize the inferred function types of lambdas like `sum` and `action`.**

```
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

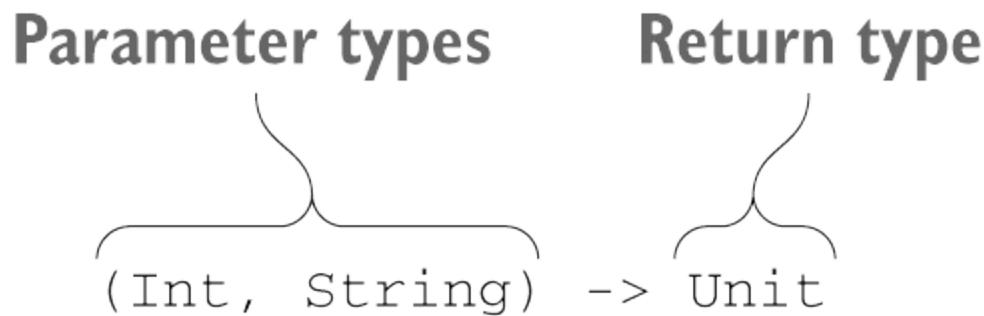
```
val action: () -> Unit = { println(42) }
```

Now let's see what an explicit type declaration for these variables looks like:

```
val sum: (Int, Int) -> Int = { x, y -> x + y } #1
val action: () -> Unit = { println(42) } #2
```

To declare a function type, you put the function parameter types in parentheses, followed by an arrow and the return type of the function (see [10.2](#)).

**Figure 10.2. Function-type syntax in Kotlin**



As you remember from [8.1.6](#), the `Unit` type is used to specify that a function returns no meaningful value. The `Unit` return type can be omitted when you declare a regular function, but a function type declaration always requires an explicit return type, so you can't omit `Unit` in this context.

Note how you can omit the types of the parameters `x`, `y` in the lambda expression `{ x, y -> x + y }`. Because they're specified in the function type as part of the variable declaration, you don't need to repeat them in the lambda itself.

Just like with any other function, the return type of a function type can be marked as nullable:

```
var canReturnNull: (Int, Int) -> Int? = { null }
```

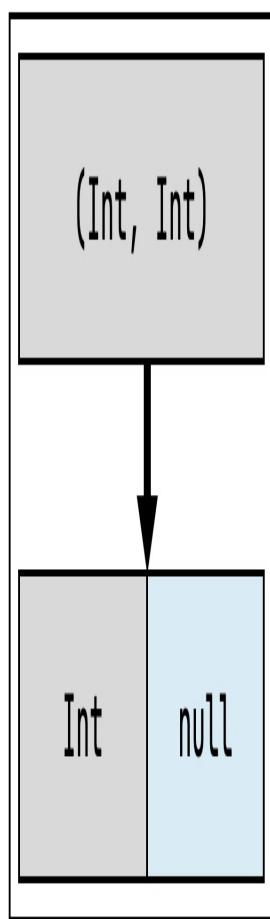
You can also define a nullable variable of a function type. To specify that the variable itself, rather than the return type of the function, is nullable, you need to enclose the entire function type definition in parentheses and put the question mark after the parentheses:

```
var funOrNull: ((Int, Int) -> Int)? = null
```

Note the subtle difference between this example and the previous one. If you omit the parentheses, you'll declare a function type with a nullable return type, and not a nullable variable of a function type. [10.3](#) illustrates this.

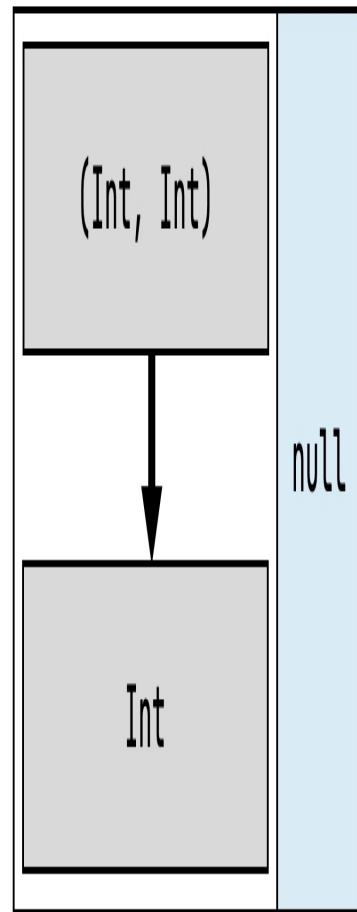
**Figure 10.3. The parentheses decide whether a function type has a nullable return type, or is nullable itself.**

Function that takes 2 integers  
and returns a nullable integer



$(\text{Int}, \text{Int}) \rightarrow \text{Int?}$

Nullable function that takes 2 integers  
and returns a non-null integer



$((\text{Int}, \text{Int}) \rightarrow \text{Int})?$

## 10.1.2 Calling functions passed as arguments

Now that you know how to specify a functional type in Kotlin in a local variable, let's discuss how to implement a higher-order function. The first example is as simple as possible and uses the same type declaration as the `sum` lambda you saw earlier. The function performs an arbitrary operation on two numbers, 2 and 3, and prints the result.

**Listing 10.1. Defining a simple higher-order function**

```
fun twoAndThree(operation: (Int, Int) -> Int) { #1
    val result = operation(2, 3) #2
    println("The result is $result")
}

fun main() {
    twoAndThree { a, b -> a + b }
    // The result is 5
    twoAndThree { a, b -> a * b }
    // The result is 6
}
```

The syntax for calling the function passed as an argument is the same as calling a regular function: you put the parentheses after the function name, and you put the parameters inside the parentheses.

### Parameter names of function types

You can specify names for parameters of a function type:

```
fun twoAndThree(
    operation: (operandA: Int, operandB: Int) -> Int #1
) {
    val result = operation(2, 3)
    println("The result is $result")
}

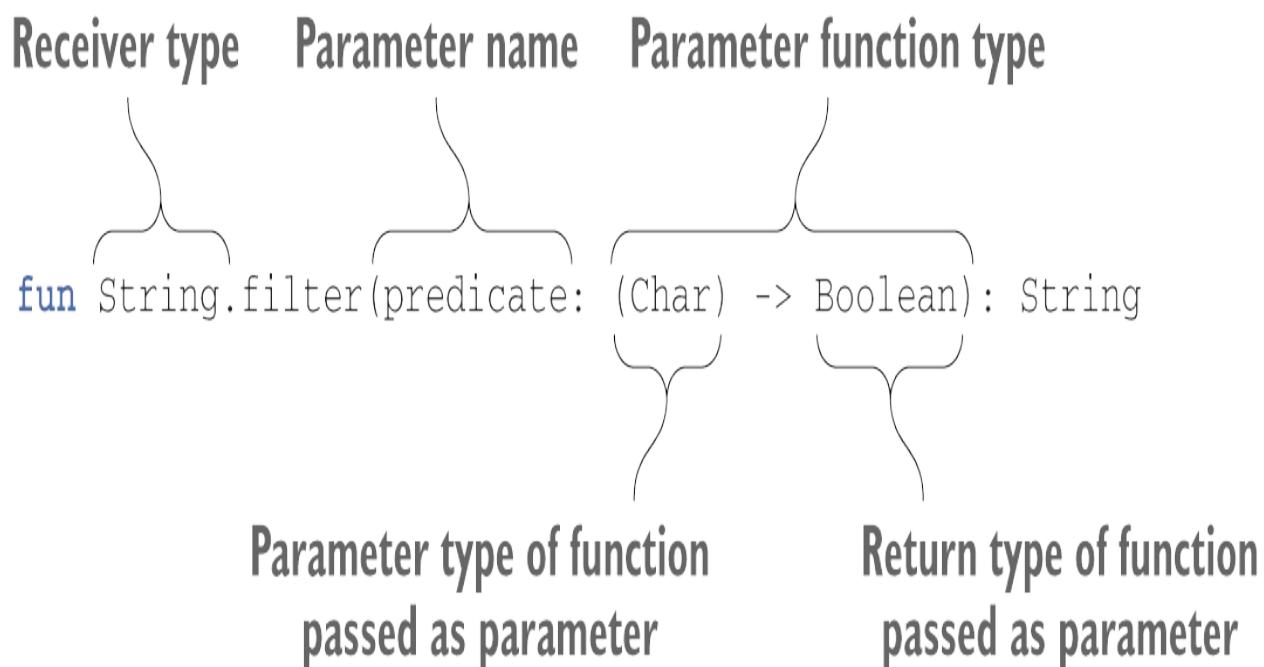
fun main() {
    twoAndThree { operandA, operandB -> operandA + operandB } #2
    // The result is 5
    twoAndThree { alpha, beta -> alpha + beta } #3
}
```

```
// The result is 5  
}
```

Parameter names don't affect type matching. When you declare a lambda, you don't have to use the same parameter names as the ones used in the function type declaration. But the names improve readability of the code and can be used in the IDE for code completion.

As a more interesting example, let's reimplement one of the most commonly used standard library functions: the `filter` function. You've used `filter` earlier, in [6.1.1](#), but now it's time to engage with its inner workings. To keep things simple, you'll implement the `filter` function on `String`, but the generic version that works on a collection of any elements is similar. The declaration of the `filter` function is shown in [10.4](#).

**Figure 10.4. Declaration of the `filter` function, taking a predicate as a parameter**



The `filter` function takes a predicate as a parameter. The type of `predicate` is a function that takes a character parameter and returns a Boolean result. When `predicate` returns `true` for given character, it needs to be present in the resulting string. If it returns `false`, it should not be included. Here's how the function can be implemented.

The `filter` function implementation is straightforward. It checks whether each character satisfies the predicate. For those characters that do, it uses the `append` function of the `StringBuilder` provided by `buildString` (as you have gotten to know it in [5.4.2](#)), gradually building up the result, and then returning it. This is particularly simple because of the iterator convention that you've seen previously in [9.3.4](#), allowing you to iterate over the `String` just like any other Kotlin collection.

Because both the extension function and the `buildString` function define a receiver, you use a labeled `this` expression to access the outer receiver of the `filter` function (the input string) rather than the receiver of the `buildString` lambda (a `StringBuilder` instance). You'll take a closer look at the labeled `this` expression in [10.6](#).

**Listing 10.2. Implementing a simple version of the `filter` function**

```
fun String.filter(predicate: (Char) -> Boolean): String {  
    return buildString {  
        for (char in this@filter) { #1  
            if (predicate(char)) append(char) #2  
        }  
    }  
}  
  
fun main() {  
    println("ab1c".filter { it in 'a'..'z' }) #3  
    // abc  
}
```



**IntelliJ IDEA tip**

IntelliJ IDEA supports smart stepping into lambda code in the debugger. If you step through the previous example, you'll see how execution moves between the body of the `filter` function and the lambda you pass through it, as the function processes each element in the input list.

### 10.1.3 Java lambdas are automatically converted to Kotlin function types

As you already discovered in [5.2.1](#), you can pass a Kotlin lambda to any Java method that expects a functional interface through automatic *SAM* (single abstract method) conversion. This means that your Kotlin code can rely on Java libraries and call higher-order functions defined in Java without problem. Likewise, Kotlin functions that use function types can be called easily from Java. Java lambdas are automatically converted to values of function types:

**Listing 10.3. ProcessTheAnswer.kt**

```
/* Kotlin declaration */
fun processTheAnswer(f: (Int) -> Int) {
    println(f(42))
}

/* Java call */
processTheAnswer(number -> number + 1);
// 43
```

In Java, you can easily use extension functions from the Kotlin standard library that expect lambdas as arguments. Note, however, that they don't look as nice as in Kotlin—you have to pass a receiver object as a first argument explicitly:

```
/* Java */
import kotlin.collections.CollectionsKt;

// ...
public static void main(String[] args) {
    List<String> strings = new ArrayList();
    strings.add("42");
    CollectionsKt.forEach(strings, s -> { #1
        System.out.println(s);
        return Unit.INSTANCE; #2
    });
}
```

In Java, your function or lambda can return `Unit`. But because the `Unit` type has a value in Kotlin, you need to return it explicitly. You can't pass a lambda returning `void` as an argument of a function type that returns `Unit`, like `(String) -> Unit` in the previous example.

## Function types: implementation details

On the JVM, Kotlin function types are declared as regular interfaces: a variable of a function type is an implementation of a `FunctionN` interface. The Kotlin standard library defines a series of interfaces, corresponding to different numbers of function arguments: `Function0<R>` (this function takes no arguments, and only specifies its return type), `Function1<P1, R>` (this function takes one argument), and so on. Each interface defines a single `invoke` method, and calling it will execute the function. A variable of a function type is an instance of a class implementing the corresponding `FunctionN` interface, with the `invoke` method containing the body of the lambda. Under the hood, that means [10.3](#) looks approximately like this:

```
fun processTheAnswer(f: Function1<Int, Int>) {  
    println(f.invoke(42))  
}
```

### 10.1.4 Parameters with function types can provide defaults or be nullable

When you declare a parameter of a function type, you can also specify its default value. To see where this can be useful, let's go back to the `joinToString` function that we discussed in [3.2](#). Here's the implementation we ended up with.

**Listing 10.4. `joinToString` with hard-coded `toString` conversion**

```
fun <T> Collection<T>.joinToString(  
    separator: String = ", ",  
    prefix: String = "",  
    postfix: String = ""  
) : String {  
    val result = StringBuilder(prefix)  
  
    for ((index, element) in this.withIndex()) {  
        if (index > 0) result.append(separator)  
        result.append(element) #1  
    }  
  
    result.append(postfix)  
    return result.toString()
```

```
}
```

This implementation is flexible, but it doesn't let you control one key aspect of the conversion: how individual values in the collection are converted to strings. The code uses `StringBuilder.append(o: Any?)`, which always converts the object to a string using the `toString` method. This is good in a lot of cases, but not always. You now know that you can pass a lambda to specify how values are converted into strings. But requiring all callers to pass that lambda would be cumbersome, because most of them are OK with the default behavior. To solve this, you can define a parameter of a function type and specify a default value for it as a lambda.

**Listing 10.5. Specifying a default value for a parameter of a function type**

```
fun <T> Collection<T>.joinToString(
    separator: String = ", ",
    prefix: String = "",
    postfix: String = "",
    transform: (T) -> String = { it.toString() } #1
): String {
    val result = StringBuilder(prefix)

    for ((index, element) in this.withIndex()) {
        if (index > 0) result.append(separator)
        result.append(transform(element)) #2
    }

    result.append(postfix)
    return result.toString()
}

fun main() {
    val letters = listOf("Alpha", "Beta")
    println(letters.joinToString()) #3
    // Alpha, Beta
    println(letters.joinToString { it.lowercase() }) #4
    // alpha, beta
    println(letters.joinToString(separator = "! ", postfix = "! "
        transform = { it.uppercase() })) #5
    // ALPHA! BETA!
}
```

Note that this function is generic: it has a type parameter `T` denoting the type

of the element in a collection. The `transform` lambda will receive an argument of that type.

Declaring a default value of a function type requires no special syntax—you just put the value as a lambda after the = sign. The examples show different ways of calling the function: omitting the lambda entirely (so that the default `toString()` conversion is used), passing it outside of the parentheses (because it is the last argument of the `joinToString` function), and passing it as a named argument.

An alternative approach is to declare a parameter of a nullable function type. Note that you can't call the function passed in such a parameter directly: Kotlin will refuse to compile such code, because it detects the possibility of null pointer exceptions in this case. One option is to check for `null` explicitly:

```
fun foo(callback: (() -> Unit)?) {  
    // ...  
    if (callback != null) {  
        callback()  
    }  
}
```

A shorter version makes use of the fact that a function type is an implementation of an interface with an `invoke` method. As a regular method, `invoke` can be called through the safe-call syntax: `callback?.invoke()`.

Here's how you can use this technique to rewrite the `joinToString` function.

#### **Listing 10.6. Using a nullable parameter of a function type**

```
fun <T> Collection<T>.joinToString(  
    separator: String = ", ",  
    prefix: String = "",  
    postfix: String = "",  
    transform: ((T) -> String)? = null #1  
) : String {  
    val result = StringBuilder(prefix)  
    for ((index, element) in this.withIndex()) {  
        if (index > 0) result.append(separator)  
        val str = transform?.invoke(element) #2
```

```

        ?: element.toString() #3
    result.append(str)
}

result.append(postfix)
return result.toString()
}

```

This example is also a good time to remind yourself once more of the function type syntax discussed in [10.2](#): `transform` is a parameter of a nullable function type, but has a non-null return type: If `transform` is not `null`, it is guaranteed to return a non-null value of type ``String``.

Now you know how to write functions that take functions as arguments. Let's look next at the other kind of higher-order functions: functions that return other functions.

### 10.1.5 Returning functions from functions

The requirement to return a function from another function doesn't come up as often as passing functions to other functions, but it's still useful. For instance, imagine a piece of logic in a program that can vary depending on the state of the program or other conditions—for example, calculating the cost of shipping depending on the selected shipping method. You can define a function that chooses the appropriate logic variant and returns it as another function. Here's how this looks as code.

#### **Listing 10.7. Defining a function that returns another function**

```

enum class Delivery { STANDARD, EXPEDITED }

class Order(val itemCount: Int)

fun getShippingCostCalculator(delivery: Delivery): (Order) -> Double {
    if (delivery == Delivery.EXPEDITED) {
        return { order -> 6 + 2.1 * order.itemCount } #2
    }
    return { order -> 1.2 * order.itemCount } #2
}

fun main() {

```

```
    val calculator = getShippingCostCalculator(Delivery.EXPEDITED
    println("Shipping costs ${calculator(Order(3))}") #4
    // Shipping costs 12.3
}
```

To declare a function that returns another function, you specify a function type as its return type. In [10.7](#), `getShippingCostCalculator` returns a function that takes an `Order` and returns a `Double`. To return a function, you write a return expression followed by a lambda, a member reference, or another expression of a function type, such as a local variable.

Let's see another example where returning functions from functions is useful. Suppose you're working on a GUI contact-management application, and you need to determine which contacts should be displayed, based on the state of the UI. Let's say the UI allows you to type a string and then shows only contacts with names starting with that string; it also lets you hide contacts that don't have a phone number specified. You'll use the `ContactListFilters` class to store the state of the options.

```
class ContactListFilters {
    var prefix: String = ""
    var onlyWithPhoneNumber: Boolean = false
}
```

When a user types D to see the contacts whose first or last name starts with D, the `prefix` value is updated. We've omitted the code that makes the necessary changes. (A full UI application would be too much code for the book, so we show a simplified example.)

To decouple the contact-list display logic from the filtering UI, you can define a function that creates a predicate used to filter the contact list. This predicate checks the prefix and also checks that the phone number is present if required.

#### **Listing 10.8. Using functions that return functions in UI code**

```
data class Person(
    val firstName: String,
    val lastName: String,
    val phoneNumber: String?
)
```

```

class ContactListFilters {
    var prefix: String = ""
    var onlyWithPhoneNumber: Boolean = false

    fun getPredicate(): (Person) -> Boolean { #1
        val startsWithPrefix = { p: Person ->
            p.firstName.startsWith(prefix) || p.lastName.startsWith(prefix)
        }
        if (!onlyWithPhoneNumber) {
            return startsWithPrefix #2
        }
        return { startsWithPrefix(it)
            && it.phoneNumber != null } #3
    }
}

fun main() {
    val contacts = listOf(
        Person("Dmitry", "Jemerov", "123-4567"),
        Person("Svetlana", "Isakova", null)
    )
    val contactListFilters = ContactListFilters()
    with (contactListFilters) {
        prefix = "Dm"
        onlyWithPhoneNumber = true
    }
    println(
        contacts.filter(contactListFilters.getPredicate()) #4
    )
    // [Person(firstName=Dmitry, lastName=Jemerov, phoneNumber=123-4567)]
}

```

The `getPredicate` method returns a function value that you pass to the `filter` function as an argument. Kotlin function types allow you to do this just as easily as for values of other types, such as strings.

Higher-order functions give you an extremely powerful tool for improving the structure of your code and removing duplication. Let's see how lambdas can help extract repeated logic from your code.

## 10.1.6 Making code more reusable by reducing duplication with lambdas

Function types and lambda expressions together constitute a great tool to create reusable code. Many kinds of code duplication that previously could be avoided only through cumbersome constructions can now be eliminated by using succinct lambda expressions.

Let's look at an example that analyzes visits to a website. The class `SiteVisit` stores the path of each visit, its duration, and the user's OS. Various OSs are represented with an enum.

**Listing 10.9. Defining the site visit data**

```
data class SiteVisit(  
    val path: String,  
    val duration: Double,  
    val os: OS  
)  
  
enum class OS { WINDOWS, LINUX, MAC, IOS, ANDROID }  
  
val log = listOf(  
    SiteVisit("/", 34.0, OS.WINDOWS),  
    SiteVisit("/", 22.0, OS.MAC),  
    SiteVisit("/login", 12.0, OS.WINDOWS),  
    SiteVisit("/signup", 8.0, OS.IOS),  
    SiteVisit("/", 16.3, OS.ANDROID)  
)
```

Imagine that you need to display the average duration of visits from Windows machines. You can perform the task using the `average` function.

**Listing 10.10. Analyzing site visit data with hard-coded filters**

```
val averageWindowsDuration = log  
    .filter { it.os == OS.WINDOWS }  
    .map(SiteVisit::duration)  
    .average()  
  
fun main() {  
    println(averageWindowsDuration)  
    // 23.0  
}
```

Now, suppose you need to calculate the same statistics for Mac users. To

avoid duplication, you can extract the platform as a parameter.

**Listing 10.11. Removing duplication with a regular function**

```
fun List<SiteVisit>.averageDurationFor(os: OS) = #1
    filter { it.os == os }.map(SiteVisit::duration).average()

fun main() {
    println(log.averageDurationFor(OS.WINDOWS))
    // 23.0
    println(log.averageDurationFor(OS.MAC))
    // 22.0
}
```

Note how making this function an extension improves readability. You can even declare this function as a local extension function if it makes sense only in the local context.

But it's not powerful enough. Imagine that you're interested in the average duration of visits from the mobile platforms (currently you recognize two of them: iOS and Android).

**Listing 10.12. Analyzing site visit data with a complex hard-coded filter**

```
fun main() {
    val averageMobileDuration = log
        .filter { it.os in setOf(OS.IOS, OS.ANDROID) }
        .map(SiteVisit::duration)
        .average()
    println(averageMobileDuration)
    // 12.15
}
```

Now a simple parameter representing the platform doesn't do the job. It's also likely that you'll want to query the log with more complex conditions, such as "What's the average duration of visits to the signup page from iOS?" Lambdas can help. You can use function types to extract the required condition into a parameter.

**Listing 10.13. Removing duplication with a higher-order function**

```
fun List<SiteVisit>.averageDurationFor(predicate: (SiteVisit) ->
```

```
    filter(predicate).map(SiteVisit::duration).average()

fun main() {
    println(
        log.averageDurationFor {
            it.os in setOf(OS.ANDROID, OS.IOS)
        }
    ) // 12.15
    println(
        log.averageDurationFor {
            it.os == OS.IOS && it.path == "/signup"
        }
    ) // 8.0
}
```

Function types can help eliminate code duplication. If you’re tempted to copy and paste a piece of the code, it’s likely that the duplication can be avoided. With lambdas, you can extract not only the data that’s repeated, but the behavior as well.



#### Note

Some well-known design patterns can be simplified using function types and lambda expressions. Let’s consider the Strategy pattern, for example. Without lambda expressions, it requires you to declare an interface with several implementations for each possible strategy. With function types in your language, you can use a general function type to describe the strategy, and pass different lambda expressions as different strategies.

We’ve discussed how to create higher-order functions. Next, let’s look at their performance. Won’t your code be slower if you begin using higher-order functions for everything, instead of writing good-old loops and conditions? The next section discusses why this isn’t always the case and how the `inline` keyword helps.

## 10.2 Removing the overhead of lambdas with inline functions

You've probably noticed that the shorthand syntax for passing a lambda as an argument to a function in Kotlin looks similar to the syntax of regular statements such as `if` and `for`. You saw this during our discussion of the `with` function in [5.4.1](#) and the `apply` function in [5.4.2](#). But what about performance? Aren't we creating unpleasant surprises by defining functions that look exactly like Java statements but run much more slowly?

In [5.2.1](#), we explained that lambdas are normally compiled to anonymous classes. But that means every time you use a lambda expression, an extra class is created; and if the lambda captures some variables, then a new object is created on every invocation. This introduces runtime overhead, causing an implementation that uses a lambda to be less efficient than a function that executes the same code directly.

Could it be possible to tell the compiler to generate code that's as efficient as directly executing the code, but still let you extract the repeated logic into a library function? Indeed, the Kotlin compiler allows you to do that. If you mark a function with the `inline` modifier, the compiler won't generate a function call when this function is used and instead will replace every call to the function with the actual code implementing the function. Let's explore how that works in detail and look at specific examples.

### **10.2.1 Inlining means substituting a function body to each call site**

When you declare a function as `inline`, its body is inlined—in other words, it's substituted directly into places where the function is called instead of being invoked normally. Let's look at an example to understand the resulting code.

The function in listing [10.14](#) can be used to ensure that a shared resource isn't accessed concurrently by multiple threads. The function locks a `Lock` object, executes the given block of code, and then releases the lock.

#### **Listing 10.14. Defining an inline function**

```
import java.util.concurrent.locks.Lock  
import java.util.concurrent.locks.ReentrantLock
```

```

inline fun <T> synchronized(lock: Lock, action: () -> T): T { #1
    lock.lock()
    try {
        return action()
    }
    finally {
        lock.unlock()
    }
}

fun main() {
    val l = ReentrantLock()
    synchronized(l) {
        // ...
    }
}

```

The syntax for calling this function looks exactly like using the `synchronized` statement in Java. The difference is that the Java `synchronized` statement can be used with any object, whereas this function requires you to pass a `Lock` instance. The definition shown here is just an example; the Kotlin standard library also defines an implementation of `synchronized`, one that accepts any object as an argument.

But using explicit locks for synchronization provides for more reliable and maintainable code. In [10.2.5](#), we'll introduce the `withLock` function from the Kotlin standard library, which you should prefer for executing the given action under a lock.

Because you've declared the `synchronized` function as `inline`, the code generated for every call to it is the same as for a `synchronized` statement in Java. Consider this example of using `synchronized()`:

```

fun foo(l: Lock) {
    println("Before sync")
    synchronized(l) {
        println("Action")
    }
    println("After sync")
}

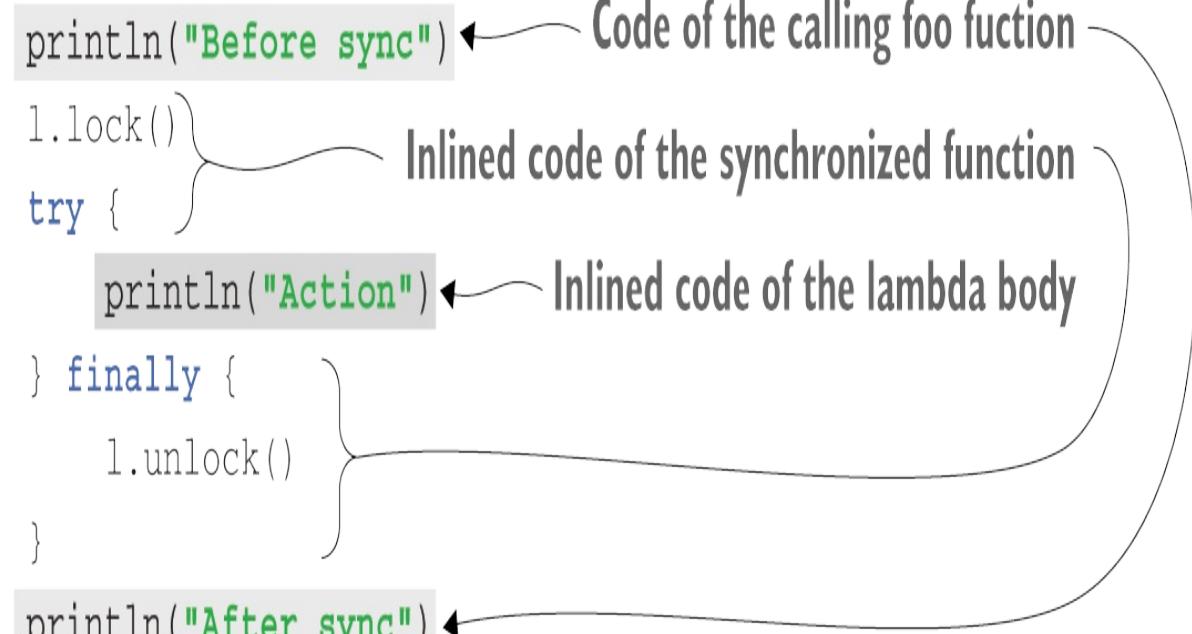
```

[10.5](#) shows the equivalent code, which will be compiled to the same

bytecode:

Figure 10.5. The compiled version of the `foo` function

```
fun __foo__(l: Lock) {  
    println("Before sync") ← Code of the calling foo function  
    l.lock()  
    try {  
        println("Action") ← Inlined code of the synchronized function  
    } finally {  
        l.unlock()  
    }  
    println("After sync") ← Inlined code of the lambda body  
}
```



Note that the inlining is applied to the lambda expression as well as the implementation of the synchronized function. The bytecode generated from the lambda becomes part of the definition of the calling function and isn't wrapped in an anonymous class implementing a function interface.

Note that it's also possible to call an inline function and pass the parameter of a function type from a variable:

```
class LockOwner(val lock: Lock) {  
    fun runUnderLock(body: () -> Unit) {  
        synchronized(lock, body) #1  
    }  
}
```

In this case, the lambda's code isn't available at the site where the inline function is called, and therefore it isn't inlined. Only the body of the

synchronized function is inlined; the lambda is called as usual. The `runUnderLock` function will be compiled to bytecode similar to the following function:

```
class LockOwner(val lock: Lock) {
    fun __runUnderLock__(body: () -> Unit) { #1
        lock.lock()
        try {
            body() #2
        }
        finally {
            lock.unlock()
        }
    }
}
```

If you have two uses of an inline function in different locations with different lambdas, then every call site will be inlined independently. The code of the inline function will be copied to both locations where you use it, with different lambdas substituted into it.

Besides functions, you can also mark your property accessors (`get`, `set`) as `inline`. This becomes useful when making use of Kotlin's *reified generics*. We'll discuss examples and their details in chapter 11.

### 10.2.2 Restrictions on inline functions

Due to the way inlining is performed, not every function that uses lambdas can be inlined. When the function is inlined, the body of the lambda expression that's passed as an argument is substituted directly into the resulting code. That restricts the possible uses of the corresponding parameter in the function body. If the lambda parameter is invoked, such code can be easily inlined. But if the parameter is stored somewhere for further use, the code of the lambda expression can't be inlined, because there must be an object that contains this code:

```
class FunctionStorage {
    var myStoredFunction: ((Int) -> Unit)? = null
    inline fun storeFunction(f: (Int) -> Unit) {
        myStoredFunction = f #1
    }
}
```

```
}
```

Generally, the parameter can be inlined if it's called directly or passed as an argument to another `inline` function. Otherwise, the compiler will prohibit the inlining of the parameter with an error message that says "Illegal usage of `inline-parameter`."

For example, various functions that work on sequences return instances of classes that represent the corresponding sequence operation and receive the lambda as a constructor parameter. Here's how the `Sequence.map` function is defined:

```
fun <T, R> Sequence<T>.map(transform: (T) -> R): Sequence<R> {
    return TransformingSequence(this, transform)
}
```

The `map` function doesn't call the function passed as the `transform` parameter directly. Instead, it passes this function to the constructor of a class that stores it in a property. To support that, the lambda passed as the `transform` argument needs to be compiled into the standard non-inline representation, as an anonymous class implementing a function interface.

If you have a function that expects two or more lambdas as arguments, you may choose to inline only some of them. This makes sense when one of the lambdas is expected to contain a lot of code or is used in a way that doesn't allow inlining. You can mark the parameters that accept such non-inlineable lambdas with the `noinline` modifier:

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit
    // ...
}
```

Note that the compiler fully supports inlining functions across modules, or functions defined in third-party libraries. You can also call most `inline` functions from Java; such calls will not be inlined, but will be compiled as regular function calls.

Later in the book, in chapter 11, you'll see another case where it makes sense to use `noinline` (with some constraints on Java interoperability, however).

### 10.2.3 Inlining collection operations

Let's consider the performance of Kotlin standard library functions that work on collections. Most of the collection functions in the standard library take lambda expressions as arguments. Would it be more efficient to implement these operations directly, instead of using the standard library functions?

For example, let's compare the ways you can filter a list of people, as shown in the next two listings.

**Listing 10.15. Filtering a collection using a lambda**

```
data class Person(val name: String, val age: Int)

val people = listOf(Person("Alice", 29), Person("Bob", 31))

fun main() {
    println(people.filter { it.age < 30 })
    // [Person(name=Alice, age=29)]
}
```

The previous code can be rewritten without lambda expressions, as shown next.

**Listing 10.16. Filtering a collection manually**

```
fun main() {
    val result = mutableListOf<Person>()
    for (person in people) {
        if (person.age < 30) result.add(person)
    }
    println(result)
    // [Person(name=Alice, age=29)]
}
```

In Kotlin, the `filter` function is declared as inline. It means the bytecode of the `filter` function, together with the bytecode of the lambda passed to it, will be inlined where `filter` is called. As a result, the bytecode generated for the first version that uses `filter` is roughly the same as the bytecode generated for the second version. You can safely use idiomatic operations on collections, and Kotlin's support for inline functions ensures that you don't

need to worry about performance.

Imagine now that you apply two operations, `filter` and `map`, in a chain.

```
fun main() {
    println(
        people.filter { it.age > 30 }
            .map(Person::name)
    )
    // [Bob]
}
```

This example uses a lambda expression and a member reference. Once again, both `filter` and `map` are declared as `inline`, so their bodies are inlined, and no extra classes or objects are created. But the code creates an intermediate collection to store the result of filtering the list. The code generated from the `filter` function adds elements to that collection, and the code generated from `map` reads from it.

If the number of elements to process is large, and the overhead of an intermediate collection becomes a concern, you can use a sequence instead, by adding an `asSequence` call to the chain. We previously discussed sequences in [6.2](#). But as you saw in the [10.2.2](#), lambdas used to process a sequence aren't inlined. Each intermediate sequence is represented as an object storing a lambda in its field, and the terminal operation causes a chain of calls through each intermediate sequence to be performed. Therefore, even though operations on sequences are lazy, you shouldn't strive to insert an `asSequence` call into every chain of collection operations in your code. This helps only for large collections; smaller ones can be processed nicely with regular collection operations.

#### 10.2.4 Deciding when to declare functions as inline

Now that you've learned about the benefits of the `inline` keyword, you might want to start using `inline` throughout your codebase, trying to make it run faster. As it turns out, this isn't a good idea. Using the `inline` keyword is likely to improve performance only with functions that take lambdas as arguments; all other cases require additional investigation, measuring, and profiling of your application.

For regular function calls, the JVM already provides powerful inlining support. It analyzes the execution of your code and inlines calls whenever doing so provides the most benefit. This happens automatically while translating bytecode to machine code. In bytecode, the implementation of each function is repeated only once and doesn't need to be copied to every place where the function is called, as with Kotlin's `inline` functions. What's more, the stacktrace is clearer if the function is called directly.

On the other hand, inlining functions with lambda arguments is beneficial. First, the overhead you avoid through inlining is more significant. You save not only on the call, but also on the creation of the extra class for each lambda and an object for the lambda instance. Second, the JVM currently isn't smart enough to always perform inlining through the call and the lambda. Finally, inlining lets you use features that are impossible to make work with regular lambdas, such as non-local returns, which we'll discuss later in this chapter.

But you should still pay attention to the code size when deciding whether to use the `inline` modifier. If the function you want to inline is large, copying its bytecode into every call site could be expensive in terms of bytecode size. In that case, you should try to extract the code not related to the lambda arguments into a separate non-inline function. You can verify for yourself that the `inline` functions in the Kotlin standard library are always small.

Next, let's see how higher-order functions can help you improve your code.

### **10.2.5 Using inlined lambdas for resource management with "withLock", "use", and "useLines"**

One common pattern where lambdas can remove duplicate code is resource management: acquiring a resource before an operation and releasing it afterward. *Resource* here can mean many different things: a file, a lock, a database transaction, and so on. The standard way to implement such a pattern is to use a `try/finally` statement in which the resource is acquired before the `try` block and released in the `finally` block, or to use specialized language constructs like Java's `try-with-resources`.

In [10.2.1](#), you saw an example of how you can encapsulate the logic of the `try/finally` statement in a function and pass the code using the resource as a lambda to that function. The example showed the `synchronized` function, which has the same syntax as the `synchronized` statement in Java: it takes the lock object as an argument. The Kotlin standard library defines another function called `withLock`, which has a more idiomatic API for the same task: it's an extension function on the `Lock` interface. Here's how it can be used:

```
val l: Lock = ReentrantLock()
l.withLock { #1
    // access the resource protected by this lock
}
```

Here's how the `withLock` function is defined in the Kotlin library:

```
inline fun <T> Lock.withLock(action: () -> T): T { #1
    lock()
    try {
        return action()
    } finally {
        unlock()
    }
}
```

Files are another common type of resource where this pattern is used. [10.17](#) shows a Kotlin function that reads the first line from a file. To do so, it uses the `use` function from the Kotlin standard library. The `use` function is an extension function called on a closable resource (an object implementing the `Closable` interface); it receives a lambda as an argument. The function calls the lambda and ensures that the resource is closed, regardless of whether the lambda completes normally or throws an exception. In this example, it makes sure that the `BufferedReader` and `FileReader`, both of which implement `Closeable`, are properly closed down after use.

#### **Listing 10.17. Using the `use` function for resource management**

```
import java.io.BufferedReader
import java.io.FileReader

fun readFirstLineFromFile(fileName: String): String {
    BufferedReader(FileReader(fileName)).use { br -> #1
        return br.readLine() #2
    }
}
```

```
        }  
    }
```

Of course, the `use` function is inlined, so its use doesn't incur any performance overhead.

As in many other cases, the Kotlin standard library also comes with more specialized extension functions. While `use` is designed to work with any type of `Closeable`, the `useLines` function is defined for `File` and `Path` objects, and gives the lambda access to a sequence of strings (as you've gotten to know them in [6.2](#)). This allows you to make the code more concise and idiomatic:

**Listing 10.18. kotlin use lines**

```
import kotlin.io.path.Path  
import kotlin.io.path.useLines  
  
fun readFirstLineFromFile(fileName: String): String {  
    Path(fileName).useLines {  
        return it.first() #1  
    }  
}
```

**No try-with-resources in Kotlin**

Java has a special syntax for working with closable resources such as Files: the *try-with-resources statement*. The equivalent Java code to [10.17](#) to read the first line from a file would look like this:

```
/* Java */  
static String readFirstLineFromFile(String fileName) throws IOException {  
    try (BufferedReader br =  
         new BufferedReader(new FileReader(fileName))) {  
        return br.readLine();  
    }  
}
```

Kotlin doesn't have any equivalent special syntax, because the same task can be accomplished just as seamlessly via `use`. This once again illustrates nicely how versatile higher-order functions (functions expecting lambdas as

arguments) can be.

Note that in the body of the lambdas (both in [10.17](#) and [10.18](#)), you use a non-local return to return a value from the `readFirstLineFromFile` function—you return from `readFirstLineFromFile` whose body contains the invocation of lambda, not just from the lambda itself. Let's discuss the use of return expressions in lambdas in detail.

## 10.3 Returning from lambdas: Control flow in higher-order functions

When you start using lambdas to replace imperative code constructs such as loops, you quickly run into the issue of return expressions. Putting a `return` statement in the middle of a loop is a no-brainer. But what if you convert the loop into the use of a function such as `filter`? How does `return` work in that case? Let's look at some examples.

### 10.3.1 Return statements in lambdas: returning from an enclosing function

We'll compare two different ways of iterating over a collection. In the following listing, it's clear that if the person's name is Alice, you return from the function `lookForAlice`.

**Listing 10.19. Using `return` in a regular loop**

```
data class Person(val name: String, val age: Int)

val people = listOf(Person("Alice", 29), Person("Bob", 31))

fun lookForAlice(people: List<Person>) {
    for (person in people) {
        if (person.name == "Alice") {
            println("Found!")
            return
        }
    }
    println("Alice is not found") #1
}
```

```
fun main() {
    lookForAlice(people)
    // Found!
}
```

Is it safe to rewrite this code using `forEach` iteration? Will the `return` statement mean the same thing? Yes, it's safe to use the `forEach` function instead, as shown next.

**Listing 10.20. Using `return` in a lambda passed to `forEach`**

```
fun lookForAlice(people: List<Person>) {
    people.forEach {
        if (it.name == "Alice") {
            println("Found!")
            return #1
        }
    }
    println("Alice is not found")
}
```

If you use the `return` keyword in a lambda, it *returns from the function in which you called the lambda*, not just from the lambda itself. Such a `return` statement is called a *non-local return*, because it returns from a larger block than the block containing the `return` statement.

To understand the logic behind the rule, think about using a `return` keyword in a `for` loop or a `synchronized` block in a Java method. It's obvious that it returns from the function and not from the loop or block. Kotlin allows you to preserve the same behavior when you switch from language features to functions that take lambdas as arguments.

Note that the return from the outer function is possible *only if the function that takes the lambda as an argument is inlined*. In [10.20](#), the body of the `forEach` function is inlined together with the body of the lambda, so it's easy to compile the `return` expression so that it returns from the enclosing function. Using the `return` expression in lambdas passed to non-inline functions isn't allowed. That is because a non-inline function might store the lambda passed to it in a variable. That means it could execute the lambda later, when the function has already returned, so it's too late for the lambda to

affect when the surrounding function returns.

### 10.3.2 Returning from lambdas: return with a label

You can write a *local* return from a lambda expression as well. A local return stops the execution of the lambda and continues execution of the code from which the lambda was invoked. To distinguish a local return from a non-local one, you use *labels*, which you've briefly seen in [2.4.1](#). You can label a lambda expression from which you want to return, and then refer to this label after the `return` keyword. In this example, you use `forEach` to iterate all elements in the input collection `people`, and use a labeled `return` to skip over elements where the `name` property is not "Alice":

**Listing 10.21. Using a local return with a label**

```
fun lookForAlice(people: List<Person>) {
    people.forEach label@{ #1
        if (it.name != "Alice") return@label #2
        print("Found Alice!") #3
    }
}

fun main() {
    lookForAlice(people)
    // Found Alice!
}
```

To label a lambda expression, put the label name (which can be any identifier), followed by the @ character, before the opening curly brace of the lambda. To return from a lambda, put the @ character followed by the label name after the `return` keyword. This is illustrated in [10.6](#).

**Figure 10.6. Returns from a lambda use the "@" character to mark a label.**

# Lambda label

```
people.forEach label@{  
    if (it.name != "Alice") return@label  
    print("Found Alice!")  
}
```

Return expression label

Alternatively, the name of the function that takes this lambda as an argument can be used as a label.

#### **Listing 10.22. Using the function name as a return label**

```
fun lookForAlice(people: List<Person>) {
    people.forEach {
        if (it.name != "Alice") return@forEach #1
        print("Found Alice!")
    }
}
```

Note that if you specify the label of the lambda expression explicitly, labeling using the function name doesn't work. A lambda expression can't have more than one label.

#### **Labeled "this" expression**

The same rules apply to the labels of `this` expressions. In [5.4](#), we discussed lambdas with receivers—lambdas that contain an implicit *receiver* object that can be accessed via a `this` reference in a lambda (Chapter 13 will explain how to write your own functions that expect lambdas with receivers as arguments). If you specify the label of a lambda with a receiver, you can access its implicit receiver using the corresponding labeled `this` expression:

```
fun main() {
    println(StringBuilder().apply sb@{ #1
        listOf(1, 2, 3).apply { #2
            this@sb.append(this.toString()) #3
        }
    })
    // [1, 2, 3]
}
```

As with labels for `return` expressions, you can specify the label of the lambda expression explicitly or use the function name instead.

The non-local return syntax is fairly verbose and becomes cumbersome if a lambda contains multiple return expressions. As a solution, you can use an alternate syntax to pass around blocks of code: *anonymous functions*.

### 10.3.3 Anonymous functions: local returns by default

An anonymous function is another syntactic form of writing a lambda expression. As such, using anonymous functions is another way to write blocks of code that can be passed to other functions. However, they differ in the way you can use `return` expressions. Let's take a closer look, and start with an example.

**Listing 10.23. Using `return` in an anonymous function**

```
fun lookForAlice(people: List<Person>) {
    people.forEach(fun (person) { #1
        if (person.name == "Alice") return #2
        println("${person.name} is not Alice")
    })
}

fun main() {
    lookForAlice(people)
    // Bob is not Alice
}
```

You can see that an anonymous function looks similar to a regular function, except that its name is omitted, and parameter types can be inferred. Here's another example.

**Listing 10.24. Using an anonymous function with `filter`**

```
people.filter(fun (person): Boolean {
    return person.age < 30
})
```

Anonymous functions follow the same rules as regular functions for specifying the return type. Anonymous functions with a block body, such as the one in [10.24](#), require the return type to be specified explicitly. If you use an expression body, you can omit the return type.

**Listing 10.25. Using an anonymous function with an expression body**

```
people.filter(fun (person) = person.age < 30)
```

Inside an anonymous function, a `return` expression without a label returns from the anonymous function, not from the enclosing one. The rule is simple: `return` *returns from the closest function declared using the `fun` keyword*. Lambda expressions don't use the `fun` keyword, so a `return` in a lambda returns from the outer function. Anonymous functions do use `fun`; therefore, in the previous example, the anonymous function is the closest matching function. Consequently, the `return` expression returns from the anonymous function, not from the enclosing one. The difference is illustrated in figure [10.7](#).

**Figure 10.7. The return expression returns from the function declared using the `fun` keyword.**

```
fun lookForAlice(people: List<Person>) {  
    people.forEach(fun(person) {  
        if (person.name == "Alice") return  
  
    })  
}  
  
fun lookForAlice(people: List<Person>) {  
    people.forEach {  
        if (it.name == "Alice") return —  
    }  
}
```

Note that despite the fact that an anonymous function looks similar to a regular function declaration, it's another syntactic form of a lambda expression. Generally, you will use the lambda syntax you have seen so far

throughout the book. Anonymous functions mainly help shortening code that has a lot of early `return` statements, that would have to be labeled when using the lambda syntax.

The discussion of how lambda expressions are implemented and how they're inlined for inline functions applies to anonymous functions as well.

## 10.4 Summary

- Function types allow you to declare a variable, parameter, or function return value that holds a reference to a function.
- Higher-order functions take other functions as arguments or return them. You can create such functions by using a function type as the type of a function parameter or return value.
- When an inline function is compiled, its bytecode along with the bytecode of a lambda passed to it is inserted directly into the code of the calling function, which ensures that the call happens with no overhead compared to similar code written directly.
- Higher-order functions facilitate code reuse within the parts of a single component and let you build powerful generic and general-purpose libraries.
- Inline functions allow you to use *non-local returns*—return expressions placed in a lambda that return from the enclosing function.
- Anonymous functions provide an alternative syntax to lambda expressions with different rules for resolving the return expressions. You can use them if you need to write a block of code with multiple exit points.

# Appendix A. Building Kotlin projects

This appendix explains how to build Kotlin code with Gradle and Maven, the two most popular build systems used with Kotlin projects.

## A.1 Building Kotlin code with Gradle

The recommended system for building Kotlin projects is Gradle. Gradle has become the de-facto standard build system for Kotlin projects, both on Android and beyond. It has a flexible project model and delivers great build performance thanks to its support for incremental builds, long-lived build processes (the Gradle daemon), and other advanced techniques.

Gradle allows you to write your build scripts either in Kotlin or Groovy. In this book, we use Gradle's Kotlin syntax, meaning both your build configuration and your actual application are written in the same language.

The easiest way to create a Gradle project with Kotlin support is via the builtin project wizard in IntelliJ IDEA, which you can find under "File | New... | Project", or via the "New Project" button on the Welcome Screen.

**Figure A.1. The New Project Wizard in IntelliJ IDEA makes it easy to set up a Kotlin project.**

### New Project

Name: sample

Location: ~/Desktop/kia2a/

Project will be created in: ~/Desktop/kia2a/sample

Create Git repository

Language: Java    Kotlin    Groovy    JavaScript    +

Build system: IntelliJ    Maven    Gradle

JDK: corretto-16 Amazon Corretto versio ▾

Gradle DSL: Groovy    Kotlin

Add sample code

To create a complex project, use the [Kotlin Multiplatform](#) generator.

Advanced Settings

GroupId: io.sebi

ArtifactId: sample

?

Cancel

Create

The standard Gradle build script for building a Kotlin project looks like this:

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

plugins {
    kotlin("jvm") version "1.7.10" #1
}

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

repositories {
    mavenCentral()
}

dependencies { #2
    testImplementation(kotlin("test")) #3
}

tasks.test {
    useJUnitPlatform()
}

tasks.withType<KotlinCompile> {
    kotlinOptions.jvmTarget = "1.8" #4
}
```

The script looks for Kotlin source files in the following locations:

- `src/main/kotlin` and `src/main/java` for the production source files
- `src/test/java` and `src/test/kotlin` for the test source files

Especially when you're introducing Kotlin into an existing project, using a single source directory reduces friction when converting Java files to Kotlin.

If you're using Kotlin reflection, which you've gotten to know in **Chapter 12**, you need to add one more dependency: the Kotlin reflection library. To do so, add the following in the dependencies section of your Gradle build script:

```
implementation(kotlin("reflect"))
```

## A.1.1 Building projects that use annotation processing

Many Java frameworks, especially those used in Android development, rely on annotation processing to generate code at compile time. To use those frameworks with Kotlin, you need to enable Kotlin annotation processing in your build script. You can do this by adding the following line to the `plugins` block of your build file:

```
kotlin("kapt") version "1.7.10"
```

If you have an existing Java project that uses annotation processing and you're introducing Kotlin to it, you need to remove the existing configuration of the `apt` tool. The Kotlin annotation processing tool handles both Java and Kotlin classes, and having two separate annotation processing tools would be redundant. To configure dependencies required for annotation processing, use the `kapt` dependency configuration:

```
dependencies {  
    implementation("com.google.dagger:dagger:2.44.2")  
    kapt("com.google.dagger:dagger-compiler:2.44.2")  
}
```

If you use annotation processors for your `androidTest` or `test` source, the respective `kapt` configurations are named `kaptAndroidTest` and `kaptTest`.

## A.2 Building Kotlin projects with Maven

If you prefer to build your projects with Maven, Kotlin supports that as well. The easiest way to create a Kotlin Maven project is to use the `org.jetbrains.kotlin:kotlin-archetype-jvm` archetype. For existing Maven projects, you can easily add Kotlin support by choosing Tools > Kotlin > Configure Kotlin in Project in the Kotlin IntelliJ IDEA plugin.

To add Maven support to a Kotlin project manually, you need to perform the following steps:

1. Add dependency on the Kotlin standard library (group ID `org.jetbrains.kotlin`, artifact ID `kotlin-stdlib`).

2. Add the Kotlin Maven plugin (group ID `org.jetbrains.kotlin`, artifact ID `kotlin-maven-plugin`), and configure its execution in the `compile` and `test-compile` phases.
3. Configure source directories, if you prefer to keep your Kotlin code in a source root separate from Java source code.

For reasons of space, we're not showing full `pom.xml` examples here, but you can find them in the online documentation at <https://kotlinlang.org/docs/maven.html>.

In a mixed Java/Kotlin project, you need to configure the Kotlin plugin so that it runs before the Java plugin. This is necessary because the Kotlin plugin can parse Java sources, whereas the Java plugin can only read `.class` files; so, the Kotlin files need to be compiled to `.class` before the Java plugin runs. You can find an example showing how this can be configured at <https://kotlinlang.org/docs/maven.html#compile-kotlin-and-java-sources>.

# Appendix B. Documenting Kotlin code

This appendix covers writing documentation comments for Kotlin code and generating API documentation for Kotlin modules.

## B.1 Writing Kotlin documentation comments

The format used to write documentation comments for Kotlin declarations is called *KDoc*. KDoc comments begin with `/**` and use tags starting with `@` to document specific parts of a declaration (just like you might be used to from Javadoc). KDoc uses Markdown

(<https://daringfireball.net/projects/markdown>) as its syntax to write the comments themselves. To make writing documentation comments easier, KDoc supports a number of additional conventions to refer to documentation elements such as function parameters.

Here's a simple example of a KDoc comment for a function.

### **Listing B.1. Using a KDoc comment**

```
/**
 * Calculates the sum of two numbers, [a] and [b]
 */
fun sum(a: Int, b: Int) = a + b
```

To refer to declarations from a KDoc comment, you enclose their names in brackets. The example uses that syntax to refer to the parameters of the function being documented, but you can also use it to refer to other declarations. If the declaration you need to refer to is imported in the code containing the KDoc comment, you can use its name directly. Otherwise, you can use a fully qualified name. If you need to specify a custom label for a link, you use two pairs of brackets and put the label in the first pair and the declaration name in the second: [an example]

[com.mycompany.SomethingTest.simple].

Here's a somewhat more complex example, showing the use of tags in a comment.

**Listing B.2. Using tags in a comment**

```
/**  
 * Performs a complicated operation.  
 *  
 * @param remote If true, executes operation remotely #1  
 * @return The result of executing the operation #2  
 * @throws IOException if remote connection fails #3  
 * @sample com.mycompany.SomethingTest.simple #4  
 */  
fun somethingComplicated(remote: Boolean): ComplicatedResult { /*
```

KDoc supports a number of tags:

- `@param parameterName`, `@param[parameterName]` to document a value parameter of a function, or the type parameter of a generic construct.
- `@return` to document the return value of a function.
- `@constructor` to document the primary constructor of a class.
- `@receiver` to document the receiver of an extension function or property.
- `@property propertyName` to document a property of a class.
- `@throws ClassName`, `@exception ClassName` to document exceptions which may be thrown by a function.
- `@sample` to include the text of the specified function into the documentation text, as an example of using the API being documented. The value of the tag is the fully qualified name of the method to be included.
- `@see otherSymbol` to include a reference to another class or function in the "See also" block of the documentation.
- `@author` to specify the author.
- `@since` to specify the version in which the documented element was introduced.
- `@suppress` to exclude this piece of documentation during export into human-readable formats.

You can find the full list of supported tags at <https://kotlinlang.org/docs/kotlin-doc.html>.

### From Javadoc to KDoc

Besides the difference in syntax—Markdown in KDoc, HTML in Javadoc, there are some other characteristics worth pointing out if you’re used to writing Javadoc to help ease the transition.

Some Javadoc tags aren’t supported in KDoc:

- `@deprecated` is replaced with the `@Deprecated` annotation.
- `@inheritDoc` isn’t supported because in Kotlin, documentation comments are always automatically inherited by overriding declarations.
- `@code`, `@literal`, and `@link` are replaced with the corresponding Markdown formatting.

Note that the documentation style preferred by the Kotlin team is to document the parameters and the return value of a function directly in the text of a documentation comment, as shown in listing [B.1](#). Using tags, as in listing [B.2](#), is recommended only when a parameter or return value has complex semantics and needs to be clearly separated from the main documentation text.



#### Rendered documentation in IntelliJ IDEA and Android Studio

Besides providing syntax highlighting and navigation for symbols in your KDoc comments, IntelliJ IDEA and Android Studio provide a *rendered view* option. You can enable it by hovering your cursor close to the line numbers next to your KDoc comment, and select the "Toggle Rendered View" option. This changes the appearance of comments to a variable-width font, and renders references and hyperlinks in place. This is especially handy when you’re browsing sources of libraries and other read-only code, since it makes the distinction between documentation and implementation even more obvious.

## B.2 Generating API documentation

The documentation-generation tool for Kotlin is called Dokka: <https://github.com/kotlin/dokka>. Just like Kotlin, Dokka fully supports cross-language Java/Kotlin projects. It can read Javadoc comments in Java code and KDoc comments in Kotlin code and generate documentation covering the entire API of a module, regardless of the language used to write each class in it. Dokka supports multiple output formats, including plain HTML, Javadoc-style HTML (using the Java syntax for all declarations and showing how the APIs can be accessed from Java), and Markdown.

You can run Dokka from the command line or as part of your Gradle or Maven build script. The recommended way to run Dokka is to add it to the Gradle build script for your module. Here's the minimum required configuration of Dokka in a Gradle build script:

```
plugins {
    id("org.jetbrains.dokka") version "1.7.10"
}

repositories {
    mavenCentral()
}

dependencies {
    // ...
    dokkaHtmlPlugin("org.jetbrains.dokka:kotlin-as-java-plugin:1.7.10")
}
```

With this configuration, you can run `./gradlew dokkaHtml` to generate documentation for your module in HTML format.

You can find information on specifying additional generation options in the Dokka documentation (<https://github.com/Kotlin/dokka/blob/master/README.md>). The documentation also shows how Dokka can be run as a standalone tool or integrated into Maven build scripts.