Kotlin Design Patterns and Best Practices

Second Edition

Build scalable applications using traditional, reactive, and concurrent design patterns in Kotlin

Alexey Soshin

Foreword by Anton Arhipov, Kotlin Developer Advocate at JetBrains

BIRMINGHAM—MUMBAI

Kotlin Design Patterns and Best Practices Second Edition

Group Product Manager: Aaron Lazar

Publishing Product Manager: Sathyanarayanan Ellapulli

Senior Editor: Rohit Singh

Content Development Editor: Rosal Colaco

Technical Editor: Karan Solanki

Copy Editor: Safis Editing

Project Coordinator: Deeksha Thakkar

Proofreader: Safis Editing

Indexer: Manju Arasan

Production Designer: Jyoti Chauhan

Marketing Coordinator: Sonakshi Bubbar

First published: June 2018

Second edition: December 2021

Production reference: 1131221

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80181-572-7

www.packt.com

To Lula Leus, my constant source of inspiration.

To my mentor, Lior Bar On. Without you, I would have never started writing.

– Alexey Soshin

Foreword

Kotlin just turned 10 recently. It's a relatively young programming language. However, Kotlin stands on the shoulders of giants: many features and best practices have been borrowed from other programming languages. Thanks to this, we can reuse the knowledge that we have learned elsewhere when developing Kotlin programs.

Design patterns are part of the best practices that will help developers to use Kotlin efficiently. It is great to see the effort in describing the patterns that comes from the passionate people in the Kotlin community. Alexey Soshin has put a lot of effort into sharing his knowledge, not only in this book but also in interactive courses and other media.

Kotlin Design Patterns and Best Practices provides a gentle introduction to the Kotlin programming language. It guides you through the vocabulary of design patterns chapter by chapter. If you are a Java developer and have used design patterns previously, the book will show you how to do the same with Kotlin.

Design patterns appeared in the era of object-oriented languages such as C++ and Java. Kotlin, however, provides language features that allow developers to program in a functional style. This book captures the benefits of the functional approach and explains how to apply it with Kotlin.

A sizable part of this book covers asynchronous programming with Kotlin coroutines. Today, it's essential to understand how to write concurrent programs and use the asynchronous programming approach to implement concurrency in your programs. This understanding is also critical for designing these programs correctly. Concurrency design patterns will help you understand how to develop concurrent programs better.

This book will give you a solid grounding for becoming familiar with Kotlin and best practices for building programs with this modern programming language. I salute the author for his effort and enthusiasm!

Anton Arhipov

Kotlin Developer Advocate at JetBrains

Contributors

About the author

Alexey Soshin is a software architect with 15 years of experience in the industry. He started exploring Kotlin when Kotlin was still in beta, and since then has been a big enthusiast of the language. He's a conference speaker, published writer, and the author of a video course titled *Pragmatic System Design*.

About the reviewers

Aditya Kumar is an Android developer with around 4 years of experience in this domain. In his journey so far, he has worked with companies such as Microsoft and Uber. He has helped in developing a few key components of many projects and is always known for his contribution to the Kotlin community. Besides Android development, he is also interested in other technologies such as engineering systems and backend engineering and is very keen on exploring those aspects in the future.

Nicola Corti is a Google Developer Expert for Kotlin. He has been working with the language since before version 1.0 and he is the maintainer of several open source libraries and tools for mobile developers (Detekt, Chucker, AppIntro, and so on). He's currently working in the React Native team at Meta in London, UK, helping to build and ship one of the most popular cross-platform frameworks for mobile. Furthermore, he is an active member of the developer community. His involvement ranges from speaking at international conferences to being a member of CFP committees and supporting developer communities across Europe.

Joost Heijkoop is an independent consultant, a seasoned JVM and frontend developer working to make things better and tackle hard problems, and an organizer at Kotlin.amsterdam and Amsterdam.scala, and is always happy to help.

Table of Contents

Section 1: Classical Patterns

Chapter 1: Getting Started with Kotlin

Chapter 2: Working with Creational Patterns

Chapter 3: Understanding Structural Patterns

Chapter 4: Getting Familiar with Behavioral Patterns

Section 2: Reactive and Concurrent Patterns

Chapter 6: Threads and Coroutines

Chapter 7: Controlling the Data Flow

Chapter 8: Designing for Concurrency

Section 3: Practical Application of Design Patterns

Chapter 9: Idioms and Anti-Patterns

Questions

Chapter 10: Concurrent Microservices with Ktor

Preface

Design patterns enable you as a developer to speed up the development process by providing you with proven development paradigms. Reusing design patterns helps prevent complex issues that can cause major problems, improves your code base, promotes code reuse, and makes an architecture more robust.

The mission of this book is to ease the adoption of design patterns in Kotlin and provide good practices for programmers.

The book begins by showing you the practical aspects of smarter coding in Kotlin, explaining the basic Kotlin syntax and the impact of design patterns. From there, the book provides an in-depth explanation of the classical creational, structural, and behavioral design pattern families, before heading into functional programming. It then takes you through reactive and concurrent patterns, teaching you about using streams, threads, and coroutines to write better code along the way.

By the end of the book, you will be able to efficiently address common problems faced while developing applications and be comfortable working on scalable and maintainable projects of any size.

Who this book is for

This book is for developers who would like to master design patterns with Kotlin in order to build reliable, scalable, and maintainable applications. Prior programming knowledge is highly advised in order to get started with this book. Prior design pattern knowledge would be helpful, but is not mandatory.

What this book covers

Chapter 1, *Getting Started with Kotlin*, covers basic Kotlin syntax and discusses what design patterns are good for and why they should be used in Kotlin. The goal of this chapter is not to cover the entire Kotlin vocabulary but to get you familiar with some basic concepts and idioms. The following chapters will slowly expose you to more language features as they become relevant to the design patterns we'll discuss.

Chapter 2, *Working with Creational Patterns*, explains all the classical creational patterns. These patterns deal with how and when to create your objects. Mastering these patterns will allow you to manage the life cycle of your objects better and write code that is easy to maintain.

Chapter 3, *Understanding Structural Patterns*, focuses on how to create hierarchies of objects that are flexible and simple to extend. It covers the Decorator and Adapter patterns, among others.

Chapter 4, *Getting Familiar with Behavioral Patterns*, covers behavioral patterns with Kotlin. Behavioral patterns deal with how objects interact with one another and how objects can change behavior dynamically. We'll see how objects can communicate efficiently and in a decoupled manner.

Chapter 5, *Introducing Functional Programming*, covers the basic principles of functional programming and how they fit into the Kotlin programming language. It will cover topics such as immutability, higher-order functions, and functions as values.

Chapter 6, *Threads and Coroutines*, dives deeper into how to launch new threads in Kotlin and covers the reasons why coroutines can scale much better than threads. We will discuss how the Kotlin compiler treats coroutines and the relationship with coroutine scopes and dispatchers.

Chapter 7, *Controlling the Data Flow*, covers higher-order functions for collections. We'll see how sequences, channels, and flows apply those functions in a concurrent and reactive manner.

Chapter 8, *Designing for Concurrency*, explains how concurrent design patterns help us manage many tasks at once and structure their life cycle. By using these patterns efficiently, we can avoiding problems such as resource leaks and deadlocks.

Chapter 9, *Idioms and Anti-Patterns*, discusses the best and worst practices in Kotlin. You'll learn what idiomatic Kotlin code should look like and also which patterns to avoid. After completing this chapter, you should be able to write more readable and maintainable Kotlin code, as well as avoiding some common pitfalls.

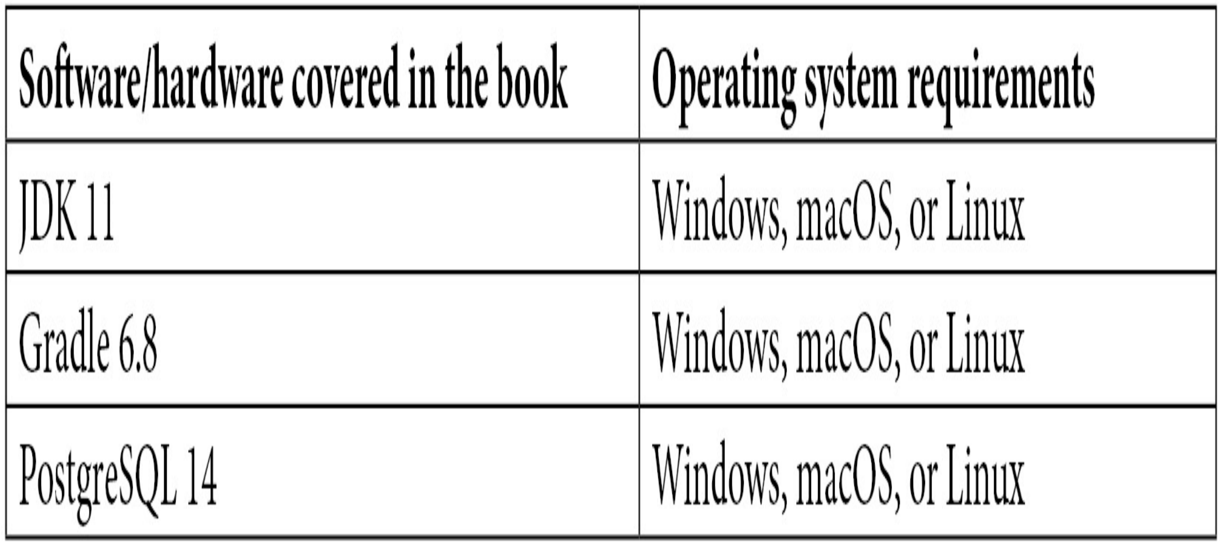
Chapter 10, *Concurrent Microservices with Ktor*, puts the skills we've learned so far to use by building a microservice using the Kotlin programming language. For that, we'll use the Ktor framework, which was developed by JetBrains.

Chapter 11, *Reactive Microservices with Vert.x*, demonstrates an alternative approach to building microservices with Kotlin by using the Vert.x framework, which is based on reactive design patterns. We'll discuss the tradeoffs between the approaches, looking at some real code examples, and figure out when to use them.

Assessments contains all the answers to the questions from all the chapters in this book.

To get the most out of this book

You should have basic knowledge of Java and know what the JVM is. It is also assumed that you are comfortable working with the command line. A few command-line examples we use in this book are based on OS X but could be easily adapted for Windows or Linux.



Software/hardware covered in the book	Operating system requirements
JDK 11	Windows, macOS, or Linux
Gradle 6.8	Windows, macOS, or Linux
PostgreSQL 14	Windows, macOS, or Linux

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices. If there's an update to the code, it will be updated in the GitHub repository.

We have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781801815727_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "From the listener side, handling exceptions is as simple as wrapping the **`collect()`** function in a **`try`**/**`catch`** block."

A block of code is set as follows:

```
val chan = produce(capacity = 10) {
    (1..10).forEach {
        send(it)
    }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
flow {
    (1..10).forEach {
    ...
        if (it == 9) {
            throw RuntimeException()
        }
    }
}
```

Any command-line input or output is written as follows:

```
...
4 seconds -> received 30
5 seconds -> received 40
6 seconds -> received 49
...
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "On the next screen, choose **JUnit 5** as your **Test framework** and set **Target JVM version** to **1.8**, then click **Finish**."

TIPS OR IMPORTANT NOTES

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *Kotlin Design Patterns and Best Practices*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Section 1: Classical Patterns

In this section, we will cover the basic syntax of the Kotlin programming language and the implementation of all the classical design patterns in Kotlin.

The classical design patterns deal with three major problems in system design: how to create objects efficiently, how to encapsulate object hierarchies, and how to make object behavior more dynamic.

We'll discuss which design patterns come as part of the language, and how to implement those that don't.

This section comprises the following chapters:

- *Chapter 1*, *Getting Started with Kotlin*
- *Chapter 2*, *Working with Creational Patterns*
- *Chapter 3*, *Understanding Structural Patterns*
- *Chapter 4*, *Getting Familiar with Behavioral Patterns*

Chapter 1: Getting Started with Kotlin

The bulk of this chapter will be dedicated to basic Kotlin syntax. It is important to be comfortable with a language before we start implementing any design patterns in it.

We'll also briefly discuss what problems design patterns solve and why you should use them in Kotlin. This will be helpful to those who are less familiar with the concept of design patterns. But even for experienced engineers, it may provide an interesting perspective.

This chapter doesn't aim to cover the entire language vocabulary but to get you familiar with some basic concepts and idioms. The following chapters will expose you to even more language features as they become relevant to the design patterns that we'll discuss.

In this chapter, we will cover the following main topics:

- Basic language syntax and features
- Understanding Kotlin code structure
- Type system and `null` safety
- Reviewing Kotlin data structures
- Control flow
- Working with text and loops
- Classes and inheritance
- Extension functions

- Introduction to design patterns

By the end of this chapter, you'll have a knowledge of Kotlin's basics, which will be the foundation for the following chapters.

Technical requirements

To follow the instructions in this chapter, you'll need the following:

- IntelliJ IDEA Community Edition (https://www.jetbrains.com/idea/download/)
- OpenJDK 11 or higher (https://openjdk.java.net/install/)

The code files for this chapter are available at https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter01.

Basic language syntax and features

Whether you come from **Java**, **C#**, **Scala**, or any other statically typed programming language, you'll find Kotlin syntax quite familiar. This is not by coincidence but to make the transition to this new language as smooth as possible for those with previous experience in other languages. Besides that familiarity, Kotlin brings a vast amount of features, such as better type safety. As we move ahead, you'll notice that all of them are attempting to solve real-world problems. That pragmatic approach is remarkably consistent across the language. For example, one of the strongest benefits of Kotlin is complete Java interoperability. You can have Java and Kotlin classes alongside each other and freely use any library that is available in Java for a Kotlin project.

To summarize, the goals of the language are as follows:

- **Pragmatic**: Makes things we do often easy to achieve
- **Readable**: Keeps a balance between conciseness and clarity on what the code does
- **Easy to reuse**: Supports adapting code to different situations
- **Safe**: Makes it hard to write code that crashes
- **Interoperable**: Allows the use of existing libraries and frameworks

This chapter will discuss how these goals are achieved.

Multi-paradigm language

Some of the major paradigms in programming languages are procedural, object-oriented, and functional paradigms.

Being pragmatic, Kotlin allows for any of these paradigms. It has classes and inheritance, coming from the object-oriented approach. It has higher-order functions from functional programming. You don't have to wrap everything in classes if you don't want to, though. Kotlin allows you to structure your entire code as just a set of procedures and structs if you need to. You will see how all these approaches come together, as different examples will combine different paradigms to solve the problems discussed.

Instead of covering all aspects of a topic from start to finish, we will be building the knowledge as we go.

Understanding Kotlin code structure

The first thing you'll need to do when you start programming in Kotlin is to create a new file. Kotlin's file extension is usually **`.kt`**.

Unlike Java, there's no strong relationship between the filename and class name. You can put as many public classes in your file as you want, as long as the classes are related to one another and your file doesn't grow too long to read.

Naming conventions

As a convention, if your file contains a single class, name your file the same as your class.

If your file contains more than one class, then the filename should describe the common purpose of those classes. Use Camel case when naming your files, as per the Kotlin coding conventions: https://kotlinlang.org/docs/coding-conventions.html.

The main file in your Kotlin project should usually be named **`Main.kt`**.

Packages

A **package** is a collection of files and classes that all share a similar purpose or domain. Packages are a convenient way to have all your classes and functions under the same namespace, and often in the same folder. That's the reason Kotlin, similar to many other languages, uses the notion of a package.

The package that the file belongs to is declared using a `package` keyword:

```
package me.soshin
```

Similar to placing classes in files, you can put any package in any directory or file, but if you're mixing Java and Kotlin, Kotlin files should follow Java package rules, as given at https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html.

In purely Kotlin projects, common package prefixes can be omitted from the folder structure. For example, if all your projects are under the `me.soshin` package, and part of your application deals with mortgages, you can place your files directly in the `/mortgages` folder and not in the `/me/soshin/mortgages` folder like Java requires.

There is no need to declare a package for your `Main.kt` file.

Comments

Going forward, we will be documenting parts of the code using **Kotlin comments**. Similar to many other programming languages, Kotlin uses `//` for a single-line comment and `/* */` for multiline comments.

Comments are a useful way to provide more context both to other developers and to your future self. Now, let's write our first Kotlin program and discuss how Kotlin's guiding principles are applied to it.

Hello Kotlin

There's no book dedicated to a programming language that can avoid the ubiquitous *Hello World* example. We're certainly not going to challenge that honored tradition.

To begin learning how Kotlin works, let's put the following code in our **`Main.kt`** file and run it:

```
fun main() {

    println("Hello Kotlin")

}
```

When your run this example, for example by pressing the **Run** button in your IntelliJ IDEA, it simply outputs the following:

```
> Hello Kotlin
```

There are some interesting attributes in that piece of code in comparison to the following Java code that does exactly the same:

```
class Main {

    public static void main(String[] args) {

        System.out.println("Hello Java");

    }

}
```

Let's focus on those attributes in the next sections.

No wrapping class

In Java, C#, Scala, and many other languages, it's necessary to wrap every function in a class for it to become executable.

Kotlin, though, has the concept of **package-level functions**. If your function doesn't need to access properties of a class, you don't need to wrap it in a class. It's as simple as that.

We'll discuss package-level functions in more detail in the following chapters.

IMPORTANT NOTE:

From here on, we'll use ellipsis notation (three dots) to indicate that some parts of the code were omitted to focus on the important bits. You can always find the full code examples at the GitHub link for this chapter.

No arguments

Arguments, supplied as an array of strings, are a way to configure your command-line application. In Java, you cannot have a runnable **`main()`** function that doesn't take this array of arguments:

```
public static void main(String[] args) { ... }
```

But in Kotlin, those are entirely optional.

No static modifier

Some languages use the **`static`** keyword to indicate that a function in a class can be executed without the need to instantiate the class. The **`main()`** function is one such example.

In Kotlin, there's no such limitation. If your function doesn't have any state, you can place it outside of a class, and there is no **`static`** keyword in Kotlin.

A less verbose print function

Instead of the verbose **`System.out.println`** method that outputs a string to the standard output, Kotlin provides us with an alias called **`println()`** that does exactly the same.

No semicolons

In Java, and many other languages, every statement or expression must be terminated with a semicolon, as shown in the following example:

```
System.out.println("Semicolon =>");
```

Kotlin is a pragmatic language. So, instead, it infers during compilation where it should put the semicolons:

```
println("No semicolons! =>")
```

Most of the time, you won't need to put semicolons in your code. They're considered optional.

This is an excellent example of how pragmatic and concise Kotlin is. It sheds lots of fluff and lets you focus on what's important.

IMPORTANT NOTE:

You don't have to write your code in a file for simple snippets. You can also play with the language online: try https://play.kotlinlang.org/ or use a REPL and an interactive shell after installing Kotlin and running **`kotlinc`**.

Understanding types

Previously, we said that Kotlin is a type-safe language. Let's examine the Kotlin type system and compare it to what Java provides.

IMPORTANT NOTE:

The Java examples are for familiarity and not to prove that Kotlin is superior to Java in any way.

Basic types

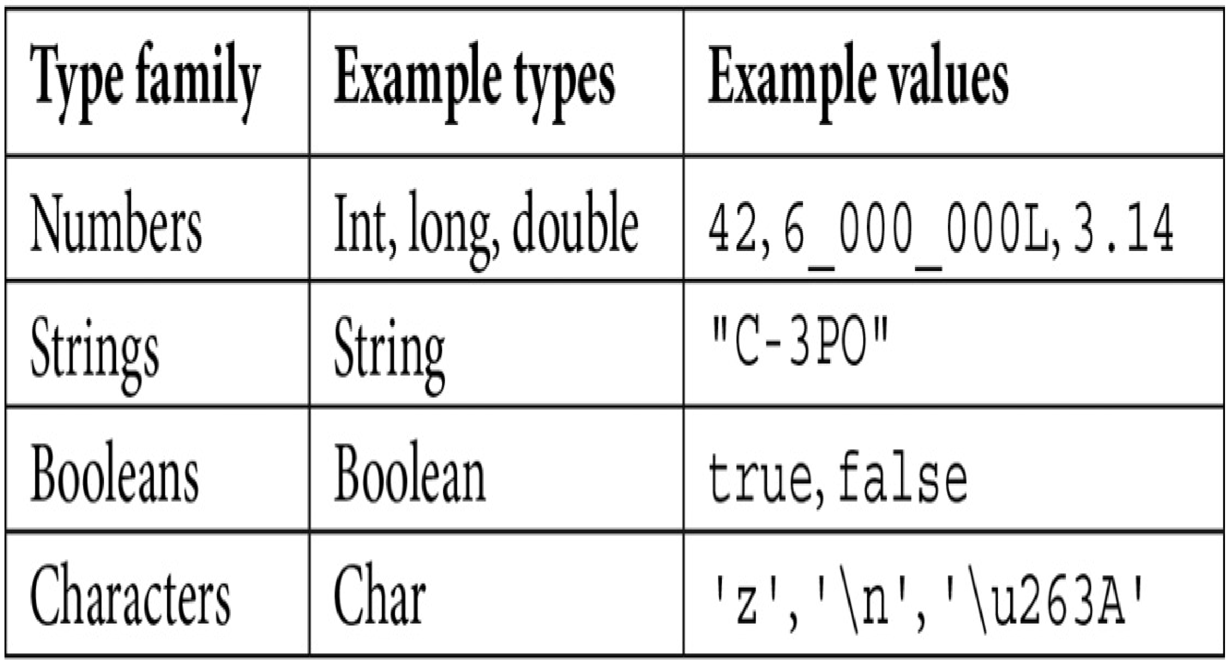
Some languages make a distinction between primitive types and objects. Taking Java as an example, there is the `int` type and `Integer` – the former being more memory-efficient and the latter more expressive by supporting a lack of value and having methods.

There is no such distinction in Kotlin. From a developer's perspective, all the types are the same.

But it doesn't mean that Kotlin is less efficient than Java in that aspect. The Kotlin compiler optimizes types. So, you don't need to worry about it much.

Most of the Kotlin types are named similarly to Java, the exceptions being Java's `Integer` being called `Int` and Java's void being called `Unit`.

It doesn't make much sense to list all the types, but here are some examples:



Type family	Example types	Example values
Numbers	Int, long, double	42, 6_000_000L, 3.14
Strings	String	"C-3PO"
Booleans	Boolean	true, false
Characters	Char	'z', '\n', '\u263A'

Table 1.1 - Kotlin types

Type inference

Let's declare our first Kotlin variable by extracting the string from our **`Hello Kotlin`** example:

```
var greeting = "Hello Kotlin"

println(greeting)
```

Note that nowhere in our code is it stated that **`greeting`** is of the **`String`** type. Instead, the compiler decides what type of variable should be used. Unlike interpreted languages, such as JavaScript, Python, or Ruby, the type of variable is defined only once.

In Kotlin, this will produce an error:

```
var greeting = "Hello Kotlin"

greeting = 1 // <- Greeting is a String
```

If you'd like to define the type of variable explicitly, you may use the following notation:

```
var greeting: String = "Hello Kotlin"
```

Values

In Java, variables can be declared **`final`**. Final variables can be assigned only once and their reference is effectively immutable:

```
final String s = "Hi";

s = "Bye"; // Doesn't work
```

Kotlin urges us to use immutable data as much as possible. Immutable variables in Kotlin are called **values** and use the **`val`** keyword:

```
val greeting = "Hi"

greeting = "Bye"// Doesn't work, "Val cannot be reassigned"
```

Values are preferable over variables. Immutable data is easier to reason about, especially when writing concurrent code. We'll touch more on that in Chapter 5, *Introducing Functional Programming*.

Comparison and equality

We were taught very early in Java that comparing objects using **==** won't produce the expected results, since it tests for reference equality – whether two pointers are the same, and not whether two objects are equal.

Instead, in Java, we use **`equals()`** for objects and **==** to compare only primitives, which may cause some confusion.

JVM does integer caching and string interning to prevent that in some basic cases, so for the sake of the example, we'll use a large integer:

```
Integer a = 1000;

Integer b = 1000;

System.out.println(a == b);      // false

System.out.println(a.equals(b)); // true
```

This behavior is far from intuitive. Instead, Kotlin translates **==** to **`equals()`**:

```
val a = 1000

val b = 1000

println(a == b)      // true

println(a.equals(b)) // true
```

If you do want to check for reference equality, use **===**. This won't work for some of the basic types, though:

```
println(a === b) // Still true
```

We'll discuss referential equality more when we learn how to instantiate classes.

Declaring functions

In Java, every method must be wrapped by a class or interface, even if it doesn't rely on any information from it. You're probably familiar with many **Util** classes in Java that only have static methods, and their only purpose is to satisfy the language requirements and bundle those methods together.

We already mentioned earlier that in Kotlin, a function can be declared outside of a class. We've seen it with the **main()** function. The keyword to declare a function is **fun**. The argument type comes after the argument name, and not before:

```
fun greet(greeting: String) {

    println(greeting)

}
```

If you need to return a result, its type will come after the function declaration:

```
fun getGreeting(): String {

    return "Hello, Kotlin!"

}
```

You can try this out yourself:

```
fun main() {
    greet(getGreeting())
}
```

If the function doesn't return anything, the return type can be omitted completely. There's no need to declare it as **`void`**, or its Kotlin counterpart, **`Unit`**.

When a function is very short and consists of just a single expression, such as our **`getGreeting()`** function, we can remove the return type and the curly brackets, and use a shorter notation:

```
fun getGreeting() = "Hello, Kotlin!"
```

Here, the Kotlin compiler will infer that we're returning a **`String`** type.

Unlike some scripting languages, the order in which functions are declared is not important. Your **`main`** function will have access to all the other functions in its scope, even if those are declared after it in the code file.

There are many other topics regarding function declarations, such as named arguments, default parameters, and variable numbers of arguments. We'll introduce them in the following chapters with relevant examples.

IMPORTANT NOTE:

Many examples in this book assume that the code we provide is wrapped in the ***`main`*** *function. If you don't see a signature of the function, it probably should be part of the* ***`main`*** *function. As an alternative, you can also run the examples in an IntelliJ scratch file.*

Null safety

Probably the most notorious exception in the Java world is **`NullPointerException`**. The reason behind this exception is that every object in Java can be **`null`**. The code here shows us why this is a problem:

```
final String s = null;

System.out.println(s.length());

// Causes NullPointerException
```

It's not like Java didn't attempt to solve that problem, though. Since **Java 8**, there has been an **`Optional`** construct that represents a value that may not be there:

```
var optional = Optional.of("I'm not null");

if (optional.isPresent()) {

    System.out.println(optional.get().length());

}
```

But it doesn't solve our problem. If our function receives **`Optional`** as an argument, we can still pass it a **`null`** value and crash the program at runtime:

```
void printLength(Optional<String> optional) {

    if (optional.isPresent()) { // <- Missing null check

      here

        System.out.println(optional.get().length());

    }

}

printLength (null); // Crashes!
```

Kotlin checks for nulls during compile time:

```
val s: String = null // Won't compile
```

Let's take a look at the **`printLength()`** function written in Kotlin:

```
fun printLength(s: String) {
    println(s.length)
}
```

Calling this function with **`null`** won't compile at all:

```
printLength(null)
// Null cannot be a value of a non-null type String
```

If you specifically want your type to be able to receive nulls, you'll need to mark it as nullable using the question mark:

```
fun printLength(stringOrNull: String?) { ... }
```

There are multiple techniques in Kotlin for dealing with nulls, such as smart casts, the Elvis operator, and so on. We'll discuss alternatives to nulls in *Chapter 4*, *Getting Familiar with Behavioral Patterns*. Let's now move on to data structures in Kotlin.

Reviewing Kotlin data structures

There are three important groups of data structures we should get familiar with in Kotlin: lists, sets, and maps. We'll cover each briefly, then discuss some other topics related to data structures, such as mutability and tuples.

Lists

A **list** represents an ordered collection of elements of the same type. To declare a list in Kotlin, we use the **`listOf()`** function:

```
val hobbits = listOf("Frodo", "Sam", "Pippin", "Merry")
```

Note that we didn't specify the type of the list. The reason is that the type inference can also be used when constructing collections in Kotlin, the same as when initializing variables.

If you want to provide the type of the list, you similarly do that for defining arguments for a function:

```
val hobbits: List<String> = listOf("Frodo", "Sam", "Pippin",
  "Merry")
```

To access an element in the list at a particular index, we use square brackets:

```
println(hobbits[1])
```

The preceding code will output this:

```
> Sam
```

Sets

A **set** represents a collection of unique elements. Looking for the presence of an element in a set is much faster than looking it up in a list. But, unlike lists, sets don't provide indexes access.

Let's create a set of football World Cup champions until after 1994:

```
val footballChampions = setOf("France", "Germany", "Spain",
  "Italy", "Brazil", "France", "Brazil", "Germany")

println(footballChampions) // [France, Germany, Spain,   Italy,
Brazil]
```

You can see that each country exists in a set exactly once. To check whether an element is in a `Set` collection, you can use the `in` function:

```
println("Israel" in footballChampions)
println("Italy" in footballChampions)
```

This gives us the following:

```
> false
> true
```

Note that although sets, in general, do not guarantee the order of elements, the current implementation of a **`setOf()`** function returns **`LinkedHashSet`**, which preserves insertion order – **`France`** appears first in the output, since it was the first country in the input.

Maps

A **map** is a collection of key-value pairs, in which keys are unique. The keyword that creates a pair of two elements is **`to`**. In fact, this is not a real keyword but a special function. We'll learn about it more in *Chapter 5*, *Introducing Functional Programming*.

In the meantime, let's create a map of some of the Batman movies and the actors that played Bruce Wayne in them:

```
val movieBatmans = mapOf(
    "Batman Returns" to "Michael Keaton",
    "Batman Forever" to "Val Kilmer",
    "Batman & Robin" to "George Clooney"
)
println(movieBatmans)
```

This prints the following:

```
> {Batman Returns=Michael Keaton,

> Batman Forever=Val Kilmer,

> Batman & Robin=George Clooney}
```

To access a value by its key, we use square brackets and provide the key:

```
println(movieBatmans["Batman Returns"])
```

The preceding code will output this:

```
> Michael Keaton
```

Those data structures also support checking that an element doesn't exist:

```
println(" Batman Begins " !in movieBatmans)
```

We get the following output:

```
> true
```

Mutability

All of the data structures we have discussed so far are immutable or, more correctly, read-only.

There are no methods to add new elements to a list we create with the **`listOf()`** function, and we also cannot replace any element:

```
hobbits[0] = "Bilbo " // Unresolved reference!
```

Immutable data structures are great for writing concurrent code. But, sometimes, we still need a collection we can modify. In order to do that, we can use the mutable counterparts of the collection functions:

```
val editableHobbits = mutableListOf("Frodo", "Sam",   "Pippin",
"Merry")

editableHobbits.add("Bilbo")
```

Editable collection types have functions such as **`add()`** that allow us to modify or, in other words, mutate them.

Alternative implementations for collections

If you have worked with JVM before, you may know that there are other implementations of sets and maps. For example, **`TreeMap`** stores the keys in a sorted order.

Here's how you can instantiate them in Kotlin:

```
import java.util.*

// Mutable map that is sorted by its keys

val treeMap = java.util.TreeMap(

    mapOf(

        "Practical Pig" to "bricks",

        "Fifer" to "straw",

        "Fiddler" to "sticks"

    )

)

println(treeMap.keys)
```

We will get the following output:

```
> [Fiddler, Fifer, Practical Pig]
```

Note that the names of the *Three Little Pigs* are ordered alphabetically.

Arrays

There is one other data structure we should cover in this section – **arrays**. In Java, arrays have a special syntax that uses square brackets. For example, an array of strings is declared **`String[]`**, while a list of strings is declared as **`List<String>`**. An element in a Java array is accessed using square brackets, while an element in a list is accessed using the **`get()`** method.

To get the number of elements in an array in Java, we use the **`length()`** method, and to do the same with a collection, we use the **`size()`** method. This is part of Java's legacy and its attempts to resemble C++.

In Kotlin, array syntax is consistent with other types of collections. An array of strings is declared as **`Array<String>`**:

```
val musketeers: Array<String> = arrayOf("Athos", "Porthos",
  "Aramis")
```

This is the first time we see angle brackets in Kotlin code. Similar to Java or TypeScript, the type between them is called **type argument**. It indicates that this array contains strings. We'll discuss this topic in detail in *Chapter 4*, *Getting Familiar with Behavioral Patterns*, while covering generics.

If you already have a collection and would like to convert it into an array, use the **`toTypedArray`** function:

```
listOf(1, 2, 3, 5).toTypedArray()
```

In terms of its abilities, a Kotlin array is very similar to a list. For example, to get the number of elements in a Kotlin array, we use the same **`size`** property as other collections.

When would you need to use arrays then? One example is accepting arguments in the **`main`** function. Previously, we've seen only main functions without arguments, but sometimes you want to pass them from a command line.

Here's an example of a **main** function that accepts arguments from a command line and prints all of them, separated by commas:

```
fun main(args: Array<String>) {

    println(args.joinToString(", "))

}
```

Other cases include invoking Java functions that expect arrays or using **varargs** syntax, which we will discuss in *Chapter 3*, *Understanding Structural Patterns*.

As we are now familiar with some basic data structures, it's time to discuss how we can apply logic to them using **if** and **when** expressions.

Control flow

You could say that the control flow is the bread and butter of writing programs. We'll start with two conditional expressions, **if** and **when**.

The if expression

In Java, **if** is a statement. Statements do not return any value. Let's look at the following function, which returns one of two possible values:

```
public String getUnixSocketPolling(boolean isBsd) {

    if (isBsd) {

        return "kqueue";

    }
    else {
        return "epoll";
    }
}
```

While this example is easy to follow, in general, having multiple **`return`** statements is considered bad practice because they often make the code harder to comprehend.

We could rewrite this method using Java's **`var`** keyword:

```
public String getUnixSocketPolling(boolean isBsd) {
    var pollingType = "epoll";
    if (isBsd) {
        pollingType = "kqueue";
    }
    return pollingType;
}
```

Now, we have a single **`return`** statement, but we had to introduce a mutable variable. Again, with such a simple example, this is not an issue. But, in general, you should try to avoid mutable shared state as much as possible, since such code is not thread-safe.

Why are we having problems writing that in the first place, though?

Contrary to Java, in Kotlin, **`if`** is an expression, meaning it returns a value. We could rewrite the previous function in Kotlin as follows:

```
fun getUnixSocketPolling(isBsd: Boolean): String {
    return if (isBsd) {
        "kqueue"
    } else {
        "epoll"
    }
}
```

Or we could use a shorter form:

```
fun getUnixSocketPolling(isBsd: Boolean): String     = if (isBsd)
"kqueue" else "epoll"
```

Due to the fact that **`if`** is an expression, we didn't need to introduce any local variables.

Here, we're again making use of single-expression functions and type inference. The important part is that **`if`** returns a value of the **`String`** type. There's no need for multiple return statements or mutable variables whatsoever.

IMPORTANT NOTE:

Single-line functions in Kotlin are very cool and pragmatic, but you should make sure that somebody else other than you understands what they do. Use with care.

The when expression

What if (no pun intended) we want to have more conditions in our **`if`** *statement?*

In Java, we use the **`switch`** statement. In Kotlin, there's a **`when`** expression, which is a lot more powerful, since it can embed some other Kotlin features. Let's create a method that's given a superhero and tells us who their archenemy is:

```
fun archenemy(heroName: String) = when (heroName) {
    "Batman" -> "Joker"
    "Superman" -> "Lex Luthor"
    "Spider-Man" -> "Green Goblin"
    else -> "Sorry, no idea"
}
```

The **`when`** expression is very powerful. In the next chapters, we will elaborate on how we can combine it with ranges, **`enums`**, and **`sealed`** classes as well.

As a general rule, use **`when`** if you have more than two conditions. Use **`if`** for simple cases.

Working with text

We've already seen many examples of working with text in the previous section. After all, it's not possible to print **`Hello Kotlin`** without using a string, or at least it would be very awkward and inconvenient.

In this section, we'll discuss some of the more advanced features that allow you to manipulate text efficiently.

String interpolation

Let's assume now we would like to actually print the results from the previous section.

First, as you may have already noticed, in one of the previous examples, Kotlin provides a nifty `println()` standard function that wraps the bulkier `System.out.println` command from Java.

But, more importantly, as in many other modern languages, Kotlin supports string interpolation using the `${}` syntax. Let's take the example from before:

```
val hero = "Batman"

println("Archenemy of $hero is ${archenemy(hero)}")
```

The preceding code would print as follows:

```
> Archenemy of Batman is Joker
```

Note that if you're interpolating a value of a function, you need to wrap it in curly braces. If it's a variable, curly braces could be omitted.

Multiline strings

Kotlin supports multiline strings, also known as **raw strings**. This feature exists in many modern languages, and was brought to **Java 15** as **text blocks**.

The idea is quite simple. If we want to print a piece of text that spans multiple lines, let's say something from *Alice's Adventures in Wonderland* by Lewis Carroll, one way is to concatenate it:

```
println("Twinkle, Twinkle Little Bat\n" +

    "How I wonder what you're at!\n" +

    "Up above the world you fly,\n" +

    "Like a tea tray in the sky.\n" +
    "Twinkle, twinkle, little bat!\n" +
    "How I wonder what you're at!")
```

While this approach certainly works, it's quite cumbersome.

Instead, we could define the same string literal using triple quotes:

```
println("""Twinkle, Twinkle Little Bat
           How I wonder what you're at!
           Up above the world you fly,
           Like a tea tray in the sky.
           Twinkle, twinkle, little bat!
           How I wonder what you're at!""")
```

This is a much cleaner way to achieve the same goal. If you execute this example, you may be surprised that the poem is not indented correctly. The reason is that multiline strings preserve whitespace characters, such as tabs.

To print the results correctly, we need to add a **`trimIndent()`** invocation:

```
println("""
    Twinkle, Twinkle Little Bat
    How I wonder what you're at!
    """.trimIndent())
```

Multiline strings also have another benefit – there's no need to escape quotes in them. Let's look at the following example:

```
println("From \" Alice's Adventures in Wonderland\" ")
```

Notice how the quote characters that are part of the text had to be escaped using the backslash character.

Now, let's look at the same text using multiline syntax:

```
println(""" From " Alice's Adventures in Wonderland" """)
```

Note that there's no need for escape characters anymore.

Loops

Now, let's discuss another typical control structure – a **loop**. Loops are a very natural construct for most developers. Without loops, it would be tough to repeat the same code block more than once (although we will discuss how to do that without loops in later chapters).

for-each loop

Probably the most helpful type of a loop in Kotlin is a **`for-each`** loop. This loop can iterate over strings, data structures, and basically everything that has an iterator. We'll learn more about iterators in Chapter 4, *Getting Familiar with Behavioral Patterns*, so for now, let's demonstrate their use on a simple string:

```
for (c in "Word") {
    println(c)
}
```

This will print the following:

```
>W
>o
>r
>d
```

The **for-each** loop works on all the types of data structures we already discussed as well, that is, lists, sets, and maps. Let's take a list as an example:

```
val jokers = listOf("Heath Ledger", "Joaquin Phoenix",   "Jack Nicholson")

for (j in jokers) {

    println(j)

}
```

We'll get the following output:

```
> Heath Ledger

> Joaquin Phoenix

> Jack Nicholson
```

You'll see this loop many more times in this book, as it's very useful.

The for loop

While in some languages **for-each** and **for** loops are two completely different constructs, in Kotlin a **for** loop is simply a **for-each** loop over a range.

To understand it better, let's look at a **for** loop that prints all the single-digit numbers:

```
for (i in 0..9) {

    println(i)

}
```

This doesn't look anything like a Java **`for`** loop and may remind you more of Python. The two dots are called a **range operator**.

If you run this code, you will notice that this loop is inclusive. It prints all the numbers, including **`9`**. This is similar to the following Java code:

```
for (int i = 0; i <= 9; i++)
```

If you want your range to be exclusive and not to include the last element, you can use the **`until`** function:

```
for (i in 0 until 10) {

    println("for until $i")

// Same output as the previous

        loop

}
```

If you'd like to print the numbers in reverse order, you can use the **`downTo`** function:

```
for (i in 9 downTo 0) {

    println("for downTo $i") // 9, 8, 7...

}
```

It may seem confusing that **`until`** and **`downTo`** are called functions, although they look more like operators. This is another interesting Kotlin feature called **infix call**, which will be discussed later.

The while loop

There are no changes to the **`while`** loop functionality compared to some other languages, so we'll cover them very briefly:

```
var x = 0

while (x < 10) {

    x++

    println("while $x")

}
```

This will print numbers from **`1`** to **`10`**. Note that we are forced to define **`x`** as **`var`**. The lesser-used **`do while`** loop is also present in the language:

```
var x = 5

do {

    println("do while $x")

    x--

} while (x > 0)
```

Most probably, you won't be using the **`while`** loop and especially the **`do while`** loop much in Kotlin. In the following chapters, we'll discuss much more idiomatic ways to do this.

Classes and inheritance

Although Kotlin is a multi-paradigm language, it has a strong affinity to the Java programming language, which is based on classes. Keeping Java and JVM interoperability in mind, it's no wonder that Kotlin also has the notion of classes and classical inheritance.

In this section, we'll cover the syntax for declaring classes, interfaces, abstract classes, and data classes.

Classes

A **class** is a collection of data, called properties, and methods. To declare a class, we use the **`class`** keyword, exactly like Java.

Let's imagine we're building a video game. We can define a class to represent the player as follows:

```
class Player {

}
```

The instantiation of a class simply looks like this:

```
val player = Player()
```

Note that there's no **`new`** keyword in Kotlin. The Kotlin compiler knows that we want to create a new instance of that class by the **round brackets** after the class name.

If the class has no body, as in this simple example, we can omit the curly braces:

```
class Player // Totally fine
```

Classes without any functions or properties aren't particularly useful, but we'll explore in Chapter 4, *Getting Familiar with Behavioral Patterns*, why this syntax exists and how it is consistent with other language features.

Primary constructor

It would be useful for the player to be able to specify their name during creation. In order to do that, let's add a primary constructor to our class:

```
class Player(name: String)
```

Now, this declaration won't work anymore:

```
val player = Player()
```

Also, we'll have to provide a name for every new player we instantiate:

```
val player = Player("Roland")
```

We'll return to constructors soon enough. But for now, let's discuss properties.

Properties

In Java, we are used to the concept of getters and setters. If we were to write a class representing a player in a game in Kotlin using Java idioms, it may have looked like this:

```
class Player(name: String) {

    private var name: String = name

    fun getName(): String {

        return name

    }

    fun setName(name: String) {

        this.name = name;

    }

}
```

If we want to get a player's name, we invoke the **`getName()`** method. If we want to change a player's name, we invoke the **`setName()`** method. That's quite simple to follow but very verbose.

It is the first time we see the **`this`** keyword in Kotlin, so let's quickly explain what it means. Similar to many other languages, **`this`** holds the reference to the current object of that class. In our case, it points to the instance of a **`Player`** class.

Why don't we write our classes like that, though?

```
class Player {

    var name: String = ""

}
```

Seems like this approach has lots of benefits. It is much less verbose for sure. Reading a person's name is now much shorter – **`player.name`**.

Also, changing the name is much more intuitive – **`player.name = "Alex";`**.

But by doing so, we lost a lot of control over our object. We cannot make **`Player`** immutable, for example. If we want everybody to be able to read the player's name, they'll also be able to change it at any point in time. This is a significant problem if we want to change that code later. With a setter, we can control that, but not with a public field.

Kotlin properties provide a solution for all those problems. Let's look at the following class definition:

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

Note that this is almost the same as the example from the *Primary constructor* section, but now **`name`** has a **`val`** modifier.

This may look the same as the **`PublicPerson`** Java example, with all its problems. But actually, this implementation is similar to **`ImmutablePerson`**, with all its benefits.

How is that possible? Behind the scenes, Kotlin will generate a member and a getter with the same name for our convenience. We can set the property value in the constructor and then access it using its name:

```
val player = Player("Alex")
```

```
println(player.name)
```

Trying to change the name of our **`Player`** will result in an error, though:

```
player.name = "Alexey" // value cannot be reassigned
```

Since we defined this property as a value, it is read-only. To be able to change a property, we need to define it as mutable. Prefixing a constructor parameter with **`var`** will automatically generate both a getter and a setter:

```
class Player(val name: String, var score: Int)
```

If we don't want the ability to provide the value at construction time, we can move the property inside the class body:

```
class Player(val name: String) {

    var score: Int = 0

}
```

Note that now we must also provide a default value for that property, since it cannot be simply **`null`**.

Custom setters and getters

Although we can set a score now easily, its value may be invalid. Take the following example:

```
player.score = -10
```

If we want to have a mutable property with some validations, we need to define an explicit setter for it, using **`set`** syntax:

```
class Player(val name: String) {

    var score: Int = 0

       set(value) {

             field = if (value >= 0) {
                 value
             } else {
                 0
             }
        }
}
```

Here, **`value`** is the new value of the property and **`field`** is its current value. If our new value is negative, we decide to use a default value.

Coming from Java, you may be tempted to write the following code in your setter instead:

```
set(value) {
    this.score = if (value >= 0) value else 0
}
```

But, in Kotlin, this will create an infinite recursion. You must remember that Kotlin generates a setter for mutable properties. So, the previous code will be translated to something like this:

```
// This is a pseudocode, not real Kotlin code!
...
fun setValue(value: Int) {
    setValue(value) // Infinite recursion!
}
...
```

For that reason, we use the **`field`** identifier, which is provided automatically.

In a similar manner, we can declare a custom getter:

```
class Player(name: String) {

    val name = name

        get() = field.toUpperCase()

}
```

First, we save a value received as a constructor argument into a field with the same name. Then, we define a custom getter that will convert all characters in this property to uppercase:

```
println(player.name)
```

We'll get this as our output:

```
> ALEX
```

Interfaces

You are probably already familiar with the concept of **interfaces** from other languages. But let's quickly recap.

In typed languages, interfaces provide a way to define behavior that some class will have to implement. The keyword to define an interface is simply **`interface`**.

Let's now define an interface for rolling a die:

```
interface DiceRoller {

    fun rollDice(): Int

}
```

To implement the interface, a class specifies its name after a colon. There's no **`implement`** keyword in Kotlin.

```
import kotlin.random.*

class Player(...) : DiceRoller

{

    ...

    fun rollDice() = Random.nextInt(0, 6)

}
```

This is also the first time we see the **`import`** keyword. As the name implies, it allows us to import another package, such as **`kotlin.random`**, from the Kotlin standard library.

Interfaces in Kotlin also support default functions. If a function doesn't rely on any state, such as this function that simply rolls a random number between **`0`** and **`5`**, we can move it into the interface:

```
interface DiceRoller {

    fun rollDice() = Random.nextInt(0, 6)

}
```

Abstract classes

Abstract classes, another concept familiar to many, are similar to interfaces in that they cannot be instantiated directly. Another class must extend them first. The difference is that unlike **`interface`**, an abstract class can contain state.

Let's create an abstract class that is able to move our player on the board or, for the sake of simplicity, just store the new coordinates:

```
abstract class Moveable() {

    private var x: Int = 0

    private var y: Int = 0

    fun move(x: Int, y: Int) {

        this.x = x

        this.y = y

    }

}
```

Any class that implements **`Moveable`** will inherit a **`move()`** function as well.

Now, let's discuss in some more detail the **`private`** keyword you see here for the first time.

Visibility modifiers

We mentioned the **`private`** keyword earlier in this chapter but didn't have a chance to explain it. The **`private`** properties or functions are only accessible to the class that declared them – **`Moveable`**, in this case.

The default visibility of classes and properties is public, so there is no need to use the **`public`** keyword all the time.

In order to extend an abstract class, we simply put its name after a colon. There's also no **`extends`** keyword in Kotlin.

```
class ActivePlayer(name: String) : Moveable(), DiceRoller {

...
}
```

How would you be able to differentiate between an abstract class and an interface, then?

An abstract class has round brackets after its name to indicate that it has a constructor. In the upcoming chapters, we'll see some uses of that syntax.

Inheritance

Apart from extending abstract classes, we can also extend regular classes as well.

Let's try to extend our **`Player`** class using the same syntax we used for an abstract class. We will attempt to create a **`ConfusedPlayer`** class, that is, a player that when given (*x* and *y*) moves to (*y* and *x*) instead.

First, let's just create a class that inherits from **`Player`**:

```
class ConfusedPlayer(name: String ): ActivePlayer(name)
```

Here, you can see the reason for round brackets even in abstract classes. This allows passing arguments to the parent class constructor. This is similar to using the **`super`** keyword in Java.

Surprisingly, this doesn't compile. The reason for this is that all classes in Kotlin are final by default and cannot be inherited from.

To allow other classes to inherit from them, we need to declare them **`open`**:

```
open class ActivePlayer (...) : Moveable(), DiceRoller {

...

}
```

Let's now try and override the **`move`** method now:

```
class ConfusedPlayer(name : String): Player(name) {

    // move() must be declared open

    override fun move(x: Int, y: Int) {

        this.x = y // must be declared protected

        this.y = x // must be declared protected

    }

}
```

Overriding allows us to redefine the behavior of a function from a parent class. Whereas in Java, **`@Override`** is an optional annotation, in Kotlin **`override`** is a mandatory keyword. You cannot hide supertype methods, and code that doesn't use **`override`** explicitly won't compile.

There are two other problems that we introduced in that piece of code. First, we cannot override a method that is not declared **`open`** as well. Second, we cannot modify the coordinates of our player from a child class since both coordinates are **`private`**.

Let's use the **`protected`** visibility modifier the makes the properties accessible to child classes and mark the function as **`open`** to be able to override it:

```
abstract class Moveable() {

    protected var x: Int = 0

    protected var y: Int = 0

    open fun move(x: Int, y: Int) {

        this.x = x

        this.y = y
    }
}
```

Now, both of the problems are fixed. You also see the **`protected`** keyword here for the first time. Similar to Java, this visibility modifier makes a property or a method visible only to the class itself and to its subclasses.

Data classes

Remember that Kotlin is all about productiveness. One of the most common tasks for Java developers is to create yet another **Plain Old Java Object** (**POJO**). If you're not familiar with POJO, it is basically an object that only has getters, setters, and implementation of **`equals`** or **`hashCode`** methods. This task is so common that Kotlin has it built into the language. It's called a **data class**.

Let's take a look at the following example:

```
data class User(val username: String, private val
  password: String)
```

This will generate us a class with two getters and no setters (note the **`val`** part), which will also implement **`equals`**, **`hashCode`**, and **`clone`** functions in the correct way.

The introduction of **`data`** classes is one of the most significant improvements in reducing the amount of boilerplate in the Kotlin language. Just like the regular classes, **`data`** classes can have their own functions:

```
data class User(val username: String, private val
  password: String) {
    fun hidePassword() = "*".repeat(password.length)
}
val user = User("Alexey", "abcd1234")
println(user.hidePassword()) // ********
```

Compared to regular classes, the main limitation of `data` classes is that they are always `final`, meaning that no other class can inherit from them. But it's a small price to pay to have `equals` and `hashCode` functions generate automatically.

Kotlin data classes versus Java records

Learning from Kotlin, Java 15 introduced the notion of **records**. Here is how we can represent the same data as a Java `record`:

```
public record User(String username, String password) {}
```

Both syntaxes are pretty concise. *Are there any differences, though?*

- Kotlin `data` classes a have `copy()` function that records lack. We'll cover it in *Chapter 2*, *Working with Creational Patterns*, while discussing the **prototype** design pattern.
- In a record, all properties must be `final`, or, in Kotlin terms, records support only values and not variables.
- The `data` classes can inherit from other classes, while records don't allow that.

To summarize, `data` classes are superior to records in many ways. But both are great features of the respective languages. And since Kotlin is built with interoperability in mind, you can also easily mark a `data` class as a record to be accessible from Java:

```
@JvmRecord
data class User(val username: String, val password: String)
```

Extension functions

The last feature we'll cover in this chapter before moving on is **extension functions**. Sometimes, you may want to extend the functionality of a class that is declared `final`. For example, you would like to have a string that has the `hidePassword()` function from the previous section.

One way to achieve that is to declare a class that wraps the string for us:

```
data class Password(val password: String) {
    fun hidePassword() = "*".repeat(password.length)
}
```

This solution is quite wasteful, though. It adds another level of indirection.

In Kotlin, there's a better way to implement this.

To extend a class without inheriting from it, we can prefix the function name with the name of the class we'd like to extend:

```
fun String.hidePassword() = "*".repeat(this.length)
```

This looks almost like a regular top-level function declaration, but with one crucial change – before the function name comes a class name. That class is called a **method receiver**.

Inside the function body, `this` will refer to any `String` class that the function was invoked on.

Now, let's declare a regular string and try to invoke this new function on it:

```
val password: String = "secretpassword"
```

```
println("Password: ${password.hidePassword()}")
```

This prints the following:

```
> Password: **************
```

What black magic is this? We managed to add a function to a **`final`** class, something that technically should be impossible.

This is another feature of the Kotlin compiler, one among many. This extension function will be compiled to the following code:

```
// This is not real Kotlin
fun hidePassword(this: String) {
    "*".repeat(this.length)
}
```

You can see that, in fact, this is a regular top-level function. Its first argument is an instance of the class that we extend. This also might remind you of how methods on structs in **Go** work.

The code that prints the masked password will be adapted accordingly:

```
val password: String = "secretpassword"
println("Password: ${hidePassword(password)}")
```

For that reason, the extension functions cannot override the member function of the class, or access its **`private`** or **`protected`** properties.

Introduction to design patterns

Now that we are a bit more familiar with basic Kotlin syntax, we can move on to discuss what design patterns are all about.

What are design patterns?

There are different misconceptions surrounding design patterns. In general, they are as follows:

- Design patterns are just missing language features.
- Design patterns are not necessary in a dynamic language.
- Design patterns are only relevant to object-oriented languages.
- Design patterns are only used in enterprises.

Actually, design patterns are just a proven way to solve a common problem. As a concept, they are not limited to a specific programming language (Java), nor to a family of languages (the C family, for example), nor are they limited to programming in general. You may have even heard of design patterns in software architecture, which discuss how different systems can efficiently communicate with each other. There are service-oriented architectural patterns, which you may know as **Service-Oriented Architecture** (**SOA**), and microservice design patterns that evolved from SOA and emerged over the past few years. The future will, for sure, bring us even more design pattern families.

Even in the physical world, outside software development, we're surrounded by design patterns and commonly accepted solutions to a particular problem. Let's look at an example.

Design patterns in real life

Did you ride an elevator lately? Was there a mirror on the wall of the elevator? Why is that? How did you feel when you last rode an elevator that *had no mirror and no glass walls?*

The main reason we commonly have mirrors in our elevators is to solve a frequent problem – riding in an elevator is boring. We could put in a picture. But a picture would also get boring after a while, if you rode the same elevator at least twice a day. Cheap, but not much of an improvement.

We could put in a TV screen, as some do. But it makes the elevator more expensive. And it also requires a lot of maintenance. We need to put some content on the screen to make it not too repetitive. So, either there's a person whose responsibility is to renew the content once in a while or a third-party company that does it for us. We'll also have to handle different problems that may occur with screen hardware and the software behind it. Seeing the *blue screen of death* is amusing, of course, but only mildly.

Some architects even go for putting elevator shafts on the building exterior and making part of the walls transparent. This may provide some exciting views. But this solution also requires maintenance (dirty windows don't make for the best view) and a lot of architectural planning.

So, we put in a mirror. You get to watch an attractive person even if you ride alone. Some studies indicate that we find ourselves more attractive than we are, anyway. Maybe you get a chance to review your appearances one last time before that important meeting. Mirrors visually expand the visual space and make the entire trip less claustrophobic or less awkward if it's the start of a day and the elevator is really crowded.

Design process

Let's try and understand what we did just now.

We didn't invent mirrors in elevators. We've seen them thousands of times. But we formalized the problem (riding in an elevator is boring) and discussed alternative solutions (TV screens and glass walls) and the benefits of the commonly used solution (solves the problem and is easy to implement). That's what design patterns are all about.

The basic steps of the design process are as follows:

1. Define exactly what the current problem is.
2. Consider different alternatives, based on the pros and cons.
3. Choose the solution that solves the problem while best fitting your specific constraints.

Why use design patterns in Kotlin?

Kotlin comes to solve the real-world problems of today. In the following chapters, we will discuss both the *design patterns* first introduced by the *Gang of Four* back in 1994, as well as design patterns that emerged from the functional programming paradigm and the design patterns that we use to handle concurrency in our applications.

You'll find that some of the design patterns are so common or useful that they're already built into the language as reserved keywords or standard functions. Some of them will need to combine a set of language features. And some are not so useful anymore, since the world has moved forward, and other patterns are replacing them.

But in any case, familiarity with design patterns and best practices expands your *developer toolbox* and creates a shared vocabulary between you and your colleagues.

Summary

In this chapter, we covered the main goals of the Kotlin programming language. We learned how variables are declared, the basic types, `null` safety, and type inference. We observed how program flow is controlled by commands such as `if`, `when`, `for`, and `while`, and we also took a look at the different keywords used to define classes and interfaces: class, interface, `data` class, and `abstract` class. We learned how to construct new classes and how to implement interfaces and inherit from other classes. Finally, we covered what design patterns are suitable for and why we need them in Kotlin.

Now, you should be able to write simple programs in Kotlin that are pragmatic and type-safe. There are many more aspects of the language we need to discuss. We'll cover them in later chapters once we need to apply them.

In the next chapter, we'll discuss the first of the three design pattern families – creation patterns.

Questions

1. What's the difference between `var` and `val` in Kotlin?
2. How do you extend a class in Kotlin?
3. How do you add functionality to a `final` class?

Chapter 2: Working with Creational Patterns

In this chapter, we'll cover how classic **creational patterns** are implemented using **Kotlin**. These patterns deal with *how* and *when* you *create* your objects. For each design pattern, we will discuss what it aims to achieve and how Kotlin accommodates those needs.

We will cover the following topics in this chapter:

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Mastering these design patterns will allow you to manage your objects better, adapt well to changes, and write code that is easy to maintain.

Technical requirements

For this chapter, you will need to install the following:

- **IntelliJ IDEA Community Edition** (https://www.jetbrains.com/idea/download/)
- **OpenJDK 11** (or higher) (https://openjdk.java.net/install/)

You can find the code files for this chapter on **GitHub** at https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter02.

Singleton

Singleton – the most popular bachelor in town. Everybody knows him, everybody talks about him, and everybody knows where to look for him.

Even people who don't like using design patterns will know Singleton by name. At one point, it was even proclaimed an **anti-pattern**, but only because of its wide popularity.

So, for those who are encountering it for the first time, what is this design pattern all about?

Usually, if you have a class, you can create as many instances of it as you want. For example, let's say that we both are asked to list all of our favorite movies:

```
val myFavoriteMovies = listOf("Black Hawk Down", "Blade   Runner")

val yourFavoriteMovies = listOf(...)
```

Note that we can create as many instances of `List` as we want, and there's no problem with that. Most classes can have multiple instances.

Next, what if we both want to list the best movies in the Quick and Angry series?

```
val myFavoriteQuickAndAngryMovies = listOf()

val yourFavoriteQuickAndAngryMovies = listOf()
```

Note that these two lists are exactly the same because they are empty. And they will stay empty because they are immutable and because the *Quick and Angry* series is simply horrendous. I hope you would agree with that.

Since these two instances of a class are exactly the same, according to the **equals method**, it doesn't make much sense to keep them in memory multiple times. It would be great if all references to an empty list pointed to the same instance of an object. And in fact, that's what happens with null, if you think about it. All nulls are the same.

That's the main idea behind the Singleton design pattern.

There are a couple of requirements for the Singleton design pattern:

- We should have exactly one instance in our system.
- This instance should be accessible from any part of our system.

In **Java** and some other languages, this task is quite complex. First, you need to forbid new instances of an object being created by making a constructor for the `private` class. Then, you also need to make sure that instantiation is preferably lazy, thread-safe, and performant, with the following requirements:

- **Lazy**: We might not want to instantiate a singleton object when our program starts, as this may be an expensive operation. We would like to instantiate it only when it's needed for the first time.
- **Thread-safe**: If two threads are trying to instantiate a singleton object at the same time, they both should receive the same instance and not two different instances. If you're not familiar with this concept, we'll cover it in *Chapter 5*, *Introducing Functional Programming*.

- **Performant**: If many threads are trying to instantiate a singleton object at the same time, we shouldn't block them for a long period of time, as this will be halting their execution.

Meeting all of these requirements in Java or **C++** is quite difficult, or at least very verbose.

Kotlin makes creating singletons easy by introducing a keyword called **`object`**. You may recognize this keyword from **Scala**. By using this keyword, we'll get an implementation of a singleton object, which accommodates all of our requirements.

IMPORTANT NOTE:

*The **`object`** keyword is used for more than just creating singletons. We'll discuss this in depth later in this chapter.*

We declare objects just like a regular class but with no constructor, as a singleton object cannot be instantiated by us:

```
object NoMoviesList
```

From now on, we can access **`NoMoviesList`** from anywhere in our code, and there will be exactly one instance of it:

```
val myFavoriteQuickAndAngryMovies = NoMoviesList

val yourFavoriteQuickAndAngryMovies = NoMoviesList

println(myFavoriteQuickAndAngryMovies ===

    yourFavoriteQuickAndAngryMovies) // true
```

Take note of the referential equality sign that checks that two variables point to the same object in memory. *Is this really a list though?*

Let's create a function that prints the list of our movies:

```
fun printMovies(movies: List<String>) {

    for (m in movies) {

        println(m)

    }

}
```

When we pass an initial list of movies, the code compiles just fine:

```
// Prints each movie on a newline

printMovies(myFavoriteMovies)
```

But if we pass it our empty movie list, the code won't compile:

```
printMovies(myFavoriteQuickAndAngryMovies)

// Type mismatch: inferred type is NoMoviesList but // List<String>
was expected
```

The reason for this is that our function only accepts arguments of the *list of strings* type, while there's nothing to tell the function that **`NoMoviesList`** is of this type (even though its name suggests it).

Luckily, in Kotlin, singleton objects can implement interfaces, and a generic **`List`** interface is available:

```
object NoMoviesList : List<String>
```

Now, our compiler will prompt us to implement the required functions. We'll do that by adding a body to **`object`**:

```
object NoMoviesList : List<String> {

    override val size = 0

    override fun contains(element: String) = false

    ... /

}
```

We'll leave it to you to implement the other functions if you wish. This should be a good exercise of everything you've learned about Kotlin until now. However, you don't have to do this. Kotlin already provides a function to create empty lists of any type:

```
printMovies(emptyList())
```

If you're curious, this function returns a singleton object that implements a **`List`**. You can see the complete implementation in the Kotlin source code using your IntelliJ IDEA or on GitHub (https://github.com/JetBrains/kotlin/blob/master/libraries/stdlib/src/kotlin/collections/Collections.kt). This is an excellent example of how design patterns are still actively applied in modern software.

A Kotlin **`object`** has one major difference from a class – it can't have constructors. If you need to implement initialization for your Singleton, such as loading data from a configuration file for the first time, you can use the **`init`** block instead:

```
object Logger {
    init {
        println("I was accessed for the first time")
        // Initialization logic goes here
    }
    // More code goes here
}
```

Note that if a Singleton is never invoked, it won't run its initialization logic at all, thereby saving resources. This is called **lazy initialization**.

Now that we have learned how to limit object creation, let's discuss how to create objects without using a constructor directly.

Factory Method

The **Factory Method** design pattern is all about creating objects.

But why do we need a method to create objects? Isn't that what constructors are for?

Well, constructors have limitations.

As an example, imagine we're building a game of chess. We would like to allow our players to save the state of the game into a text file and then restore the game from that position.

Since the size of the board is predetermined, we only need to record the position and type of each piece. We'll use algebraic notation for this – for example, the Queen piece at C3 will be stored in our file as **`qc3`**, the pawn piece at A8 will be stored as **`pa8`**, and so on.

Let's assume that we already read this file into a list of strings (which, by the way, would be an excellent application of the Singleton design pattern we discussed earlier).

Given the list of notations, we would like to populate our board with them:

```
// More pieces here

val notations = listOf("pa8", "qc3", ...)

val pieces = mutableListOf<ChessPiece>()

for (n in notations) {

    pieces.add(createPiece(n))
}
println(pieces)
```

Before we can implement our **`createPiece`** function, we need to decide what's common to all chess pieces. We'll create an interface for that:

```
interface ChessPiece {
    val file: Char
    val rank: Char
}
```

Note that interfaces in Kotlin can declare properties, which is a very powerful feature.

Each chess piece will be a **`data class`** that implements our interface:

```
data class Pawn(
    override val file: Char,
    override val rank: Char
) : ChessPiece
data class Queen(
    override val file: Char,
    override val rank: Char
) : ChessPiece
```

The implementation of the other chess pieces is left as an exercise for you to do.

Now, what's left is to implement our **`createPiece`** function:

```
fun createPiece(notation: String): ChessPiece {
    val (type, file, rank) = notation.toCharArray()
    return when (type) {
        'q' -> Queen(file, rank)
        'p' -> Pawn(file, rank)
        // ...
        else -> throw RuntimeException("Unknown piece: $type")
    }
}
```

Before we can discuss what this function achieves, let's cover three new syntax elements we haven't seen before.

First, the **`toCharArray`** function splits a string into an array of characters. Since we assume that all of our notations are three characters long, the element at the **`0`** position will represent the *type* of the chess piece, the element at the **`1`** position will represent its vertical column – also known as **`file`** – and the last element will represent its horizontal column – also known as **`rank`**.

Next, we can see three values: **`type`**, **`file`**, and **`rank`**, surrounded by parentheses. This is called a **destructuring declaration**, and you may be familiar with them from JavaScript, for example. Any **`data class`** can be destructured.

The previous code example is similar to the following, much more verbose code:

```
val type = notation.toCharArray()[0]
val file = notation.toCharArray()[1]
val rank = notation.toCharArray()[2]
```

Now, let's focus on the **`when`** expression. Based on the letter representing the type, it instantiates one of the implementations of the **`ChessPiece`** interface. Remember, this is what the Factory Method design pattern is all about.

To make sure you grasp this design pattern well, feel free to implement the classes and logic for other chess pieces as an exercise.

Finally, let's look at the bottom of our function, where we see the first use of a **`throw`** expression.

This expression, as the name suggests, *throws* an exception, which will stop the normal execution of our simple program. We'll discuss how to handle exceptions in Chapter 5, *Introducing Functional Programming*.

In the real world, the Factory Method design pattern is often used by libraries that need to parse configuration files – be they of the XML, JSON, or YAML format – into runtime objects.

Static Factory Method

There is a similarly named design pattern (which has a slightly different implementation) that is often confused with the Factory Method design pattern, and it is described in the *Gang of Four* book – the **Static Factory Method** design pattern.

The Static Factory Method design pattern was popularized by Joshua Bloch in his book, *Effective Java*. To understand this better, let's look at some examples from the Java standard library: the **`valueOf()`** methods. There are at least two ways to construct a **`Long`** (that is, a 64-bit integer) from a string:

```
Long l1 = new Long("1"); // constructor
Long l2 = Long.valueOf("1"); // static factory method
```

Both the constructor and the **`valueOf()`** method receive string as input and produce **`Long`** as output.

So, why should we prefer the Static Factory Method design pattern to a simple constructor?

Here are some of the advantages of using the Static Factory Method compared to constructors:

- It provides an opportunity to explicitly name different object constructors. This is especially useful when your class has multiple constructors.
- We usually don't expect exceptions from a constructor. That doesn't mean that the instantiation of a class can't fail. Exceptions from a regular method, on the other hand, are much more accepted.
- Speaking of expectations, we expect the constructor to be fast. But construction of some objects is inherently slow. Consider using the Static Factory Method instead.

These are mostly style advantages; however, there are also technological advantages to this approach.

Caching

The Static Factory Method design pattern may provide **caching**, as **`Long`** actually does. Instead of always returning a new instance for any value, **`valueOf()`** checks in the cache whether this value was already parsed. If it was, it returns a cached instance. Repeatedly calling the Static Factory

Method with the same values may produce less garbage for collection than using constructors all the time.

Subclassing

When calling the constructor, we always instantiate the class we specify. On the other hand, calling a Static Factory Method is less restrictive and may produce either an instance of the class itself or one of its subclasses. We'll come to this after discussing the implementation of this design pattern in Kotlin.

Static Factory Method in Kotlin

We discussed the `object` keyword earlier in this chapter in the *Singleton* section. Now, we'll see another use of it as a **companion object**.

In Java, Static Factory Methods are declared `static`. But in Kotlin, there's no such keyword. Instead, methods that don't belong to an instance of a class can be declared inside `companion object`:

```
class Server(port: Long) {

    init {

        println("Server started on port $port")

    }

    companion object {

        fun withPort(port: Long) = Server(port)

  }

}
```

IMPORTANT NOTE:

Companion objects may have a name – for example, ***`companion object`*** *parser. But this is only to provide clarity about what the goal of the object is.*

As you can see, this time, we have declared an object that is prefixed by the **`companion`** keyword. Also, it's located inside a class, and not at the package level in the way we saw in the Singleton design pattern.

This object has its own methods, and you may wonder what the benefit of this is. Just like a Java static method, calling a **`companion object`** will lazily instantiate it when the containing class is accessed for the first time:

```
Server.withPort(8080) // Server started on port 8080
```

Moreover, calling it on an instance of a class simply won't work, unlike in Java:

```
Server(8080) // Won't compile, constructor is private
```

IMPORTANT NOTE:

*A class may have only one **`companion object`**.*

Sometimes, we also want the Static Factory Method to be the only way to instantiate our object. In order to do that, we can declare the default constructor of our object as **`private`**:

```
class Server private constructor(port: Long) {

    ...

}
```

This means that now there's only one way of constructing an instance of our class – through our Static Factory Method:

```
val server = Server(8080))  // Doesn't compile

val server = Server.withPort(8080) // Works!
```

Let's now discuss another design pattern that is often confused with the Factory Method – Abstract Factory.

Abstract Factory

Abstract Factory is a greatly misunderstood design pattern. It has a notorious reputation for being very complex and bizarre. Actually, it's quite simple. If you understood the Factory Method design pattern, you'll understand this one in no time. This is because the Abstract Factory design pattern is a factory of factories. That's all there is to it. The *factory* is a function or class that's able to create other classes. In other words, an abstract factory is a class that wraps multiple factory methods.

You may understand this and still wonder what the use of such a design pattern may be. In the real world, the Abstract Factory design pattern is often used in frameworks and libraries that get their configuration from files. The **Spring Framework** is just one example of these.

To better understand how the design pattern works, let's assume we have a configuration for our server written in a YAML file:

```
server:
    port: 8080
environment: production
```

Our task is to construct objects from this configuration.

In the previous section, we discussed how to use Factory Method to construct objects from the same family. But here, we have two families of objects that are related to each other but are not *siblings*.

First, let's describe them as interfaces:

```
interface Property {
    val name: String
    val value: Any
}
```

Instead of a **`data class`**, we'll return an interface. You'll see how this helps us later in this section:

```
interface ServerConfiguration {
    val properties: List<Property>
}
```

Then, we can provide basic implementations to be used later:

```
data class PropertyImpl(
    override val name: String,
    override val value: Any
) : Property
data class ServerConfigurationImpl(
    override val properties: List<Property>
) : ServerConfiguration
```

The server configuration simply contains the list of properties – and a *property* is a pair comprising a **`name`** object and a **`value`** object.

This is the first time we have seen the **`Any`** type being used. The **`Any`** type is Kotlin's version of Java's **`object`**, but with one important distinction: it cannot be null.

Now, let's write our first Factory Method, which will create **`Property`** given as a string:

```
fun property(prop: String): Property {
    val (name, value) = prop.split(":")
    return when (name) {
        "port" -> PropertyImpl(name, value.trim().toInt())
        "environment" -> PropertyImpl(name, value.trim())
        else -> throw RuntimeException("Unknown property:
          $name")
    }
}
```

As in many other languages, **`trim()`** is a function that is declared on strings that removes any spaces in the string. Now, let's create two properties to represent the port (**`port`**) and environment (**`environment`**) of our service:

```
val portProperty = property("port: 8080")
val environment = property("environment: production")
```

There is a slight issue with this code. To understand what it is, let's try to store the value of the **`port`** property into another variable:

```
val port: Int = portProperty.value
// Type mismatch: inferred type is Any but Int was expected
```

We already ensured that **`port`** is parsed to an **`Int`** in our Factory Method. But now, this information is lost because the type of the value is declared as **`Any`**. It can be **`String`**, **`Int`**, or any other type, for that matter. We need a new tool to solve this issue, so let's take a short detour and discuss casts in Kotlin.

Casts

Casts in typed languages are a way to try and force the compiler to use the type we specify, instead of the type it has inferred. If we are sure what type the value is, we can use an *unsafe* cast on it:

```
val port: Int = portProperty.value as Int
```

The reason it is called *unsafe* is that if the value is not of the type we expect, our program will crash without the compiler being able to warn us.

Alternatively, we could use a *safe* cast:

```
val port: Int? = portProperty.value as? Int
```

Safe casts won't crash our program, but if the type of the object is not what we expect, it will return null. Notice that our **`port`** variable now is declared as the nullable **`Int`**, so we have to explicitly deal with the possibility of not getting what we want during compilation time.

Subclassing

Instead of resorting to casts, let's try another approach. Instead of using a single implementation with a value of the **`Any`** type, we'll use two separate implementations:

```
data class IntProperty(

    override val name: String,

    override val value: Int

) : Property

data class StringProperty(

    override val name: String,

    override val value: String

) : Property
```

Our Factory Method will have to change a little to be able to return one of the two classes:

```
fun property(prop: String): Property {

    val (name, value) = prop.split(":")

    return when (name) {

        "port" -> IntProperty(name, value.trim().toInt())

        "environment" -> StringProperty(name, value.trim())

        else -> throw RuntimeException("Unknown property:
          $name")

    }

}
```

This looks fine, but if we try to compile our code again, it still won't work:

```
val portProperty = Parser.property("port: 8080")

val port: Int = portProperty.value
```

Although we now have two concrete classes, the compiler doesn't know if the parsed property is **`IntProperty`** or **`StringProperty`**. All it knows is that it's **`Property`**, and the type of the value is still **`Any`**:

```
> Type mismatch: inferred type is Any but Int was expected
```

We need another trick, and that trick is called **smart casts**.

Smart casts

We can check if an object is of a given type by using the **`is`** keyword:

```
println(portProperty is IntProperty) // true
```

However, the Kotlin compiler is very smart. *If we performed a type check on an* **`if`** *expression, it would mean that* **`portProperty`** *was indeed* **`IntProperty`***, right?* So, it could be safely cast.

The Kotlin compiler will do just that for us:

```
if (portProperty is IntProperty) {

    val port: Int = portProperty.value // works!

}
```

There is no compilation error anymore, and we also do not have to deal with nullable values.

Smart casts also work on nulls. In Kotlin's type hierarchy, the non-nullable **`Int`** type is a subclass of a nullable type, **`Int?`**, and this is true for all types. Previously, we mentioned that a *safe* cast will return **`null`** if it fails:

```
val port: Int? = portProperty.value as? Int
```

We could check if **`port`** is null, and if it isn't, it will be smartly cast to a non-nullable type:

```
if (port != null) {

    val port: Int = port

}
```

Nice! But wait, what's going on in this code?

In the previous chapter, we said that values cannot be reassigned. But here, we defined the **`port`** value twice. *How is this possible?* This is not a bug, but another Kotlin feature, and it is called **variable shadowing**.

Variable shadowing

First, let's consider how our code would look if there was no shadowing. We would have to declare two variables with different names:

```
val portOrNull: Int? = portProperty.value as? Int
```

```
if (portOrNull != null) {

    val port: Int = portOrNull // works

}
```

However, this is a waste, for two reasons. First, the variable names become quite verbose. Second, the **`portOrNull`** variable would most probably never be used past this point because null is not a very useful value to begin with. Instead, we can declare values with the same names in different scopes, denoted by curly brackets (**`{}`**).

Please note that variable shadowing may confuse you, and it is error-prone by nature. However, it is important to be aware that it exists, but the recommendation is to name your variables explicitly whenever possible.

Collection of Factory Methods

Now that we've had our detour into casts and variable shadowing, let's go back to the previous code example and implement a second Factory Method, that will create a **`server`** configuration object:

```
fun server(propertyStrings: List<String>):

  ServerConfiguration {

    val parsedProperties = mutableListOf<Property>()

    for (p in propertyStrings) {

        parsedProperties += property(p)

    }

    return ServerConfigurationImpl(parsedProperties)

}
```

This method takes the lines from our configuration file and converts them into **`Property`** objects using the **`property()`** Factory Method that we've already implemented.

We can test that our second Factory Method works as well:

```
println(server(listOf("port: 8080", "environment:
  production")))
```

```
> ServerConfigurationImpl(properties=[IntProperty(name=port,
value=8080), StringProperty(name=environment, value=production)])
```

Since these two methods are related, it would be good to put them together under the same class. Let's call this class **`Parser`**. Although we didn't parse any actual file and agreed that we get its contents line by line already, by the end of this book, you would probably agree that implementing the actual reading logic is quite trivial.

We can also use Static Factory Method and the **`companion object`** syntax we learned about in the previous section.

The resulting implementation will look like this:

```
class Parser {
    companion object {
        fun property(prop: String): Property {
           ...
        }
        fun server(propertyStrings: List<String>): ...{
           ...
        }
    }
}
```

This pattern allows us to create *families* of objects – in this case, **`ServerConfig`** is the *parent* of a property.

The previous code is just one way to implement an Abstract Factory. You may find some implementations that rely on implementing an interface instead:

```
interface Parser {

    fun property(prop: String): Property

    fun server(propertyStrings: List<String>):
      ServerConfiguration

}

class YamlParser : Parser {

    // Implementation specific to YAML files

}

class JsonParser : Parser {

    // Implementation specific to JSON files

}
```

This approach may be better if your Factory Methods grow to contain lots of code.

One last question you may have is where we can see Abstract Factory used in real code. One example is the **`java.util.Collections`** class. It has methods such as **`emptyMap`**, **`emptyList`**, and **`emptySet`**, which each generate a different class. However, what is common to all of them is that they are all collections.

Builder

Sometimes, our objects are very simple and have only one constructor, be it an empty or non-empty one. But sometimes, their creation is very complex and based on a lot of parameters. We've seen one pattern already that provides *a better constructor* – the Static Factory Method design pattern. Now, we'll discuss the **Builder** design pattern, which will help us create complex objects.

As an example of such an object, imagine we need to design a system that sends emails. We won't implement the actual mechanism of sending them, we will just design a class that represents it.

An email may have the following properties:

- An address (at least one is mandatory)
- CC (optional)
- Title (optional)
- Body (optional)
- Important flag (optional)

We can describe an email in our system as a `data class`:

```
data class Mail_V1(

    val to: List<String>,

    val cc: List<String>?,

    val title: String?,

    val message: String?,

    val important: Boolean,
)
```

IMPORTANT NOTE:

*Look at the definition of the last argument in the preceding code. This comma is not a typo. It is called a **trailing comma**, and these were introduced in **Kotlin 1.4**. This is done so you can easily change the order of the arguments.*

Next, let's attempt to create an email addressed to our manager:

```
val mail = Mail_V1(
    listOf("manager@company.com"),    // To
    null,                             // CC
    "Ping ",                          // Title
    null,                             // Message,
    true))                            // Important
```

Note that we have defined *carbon copy* (that's what `cc` stands for) as nullable so that it can receive either a list of emails or null. Another option would be to define it as **`List<String>`** and force our code to pass **`listOf()`**.

Since our constructor receives a lot of arguments, we had to put in some comments in order not to get confused.

But what happens if we need to change this class now?

First, our code will stop compiling. Second, we need to keep track of the comments. In short, constructors with a long list of arguments quickly become a mess.

This is the problem the Builder design pattern sets out to solve. It decouples the assigning of arguments from the creation of objects and allows the creation of complex objects one step at a time. In this section, we'll see a number of approaches to this problem.

Let's start by creating a new class, **`MailBuilder`**, which will wrap our **`Mail`** class:

```
class MailBuilder {

    private var to: List<String> = listOf()

    private var cc: List<String> = listOf()

    private var title: String = ""

    private var message: String = ""

    private var important: Boolean = false

    class Mail internal constructor(

        val to: List<String>,

        val cc: List<String>?,

        val title: String?,

        val message: String?,

        val important: Boolean

    )

    ... // More code will come here soon

}
```

Our builder has exactly the same properties as our resulting class. But these properties are all mutable.

Note that the constructor is marked using the **`internal`** visibility modifier. This means that our **`Mail`** class will be accessible to any code inside our module.

To finalize the creation of our class, we'll introduce the **`build()`** function:

```
fun build(): Mail {

    if (to.isEmpty()) {
        throw RuntimeException("To property is empty")
    }
    return Mail(to, cc, title, message, important)
}
```

And for each property, we'll have another function to be able to set it:

```
fun message(message: String): MailBuilder {
    this.message = message
    return this
}
// More functions for each of the properties
```

Now, we can use our builder to create an email in the following way:

```
val email = MailBuilder("hello@hello.com").title("What's
  up?").build()
```

After setting a new value, we return a reference to our object by using **`this`**, which provides us with access to the next setter to allow us to perform chaining (please refer to the *Fluent setters* section in this chapter for an explanation of this).

This is a working approach. But it has a couple of downsides:

- The properties of our resulting class must be repeated insider the builder.
- For every property, we need to declare a function to set its value.

Kotlin provides two other ways that you may find even more useful.

Fluent setters

The approach using **fluent setters** is a bit more concise. Here, we won't construct any additional classes. Instead, our `data class` constructor will contain only the mandatory fields. All other fields will become `private`, and we'll provide setters for these fields:

```
data class Mail_V2(
    val to: List<String>,
    private var _message: String? = null,
    private var _cc: List<String>? = null,
    private var _title: String? = null,
    private var _important: Boolean? = null
) {
    fun message(message: String) = apply {
        _message = message
    }
    // Pattern repeats for every other field
    //...
}
```

IMPORTANT NOTE:

Using underscores for `private` variables is a common convention in Kotlin. It allows us to avoid repeating `this.message = message` and mistakes such as `message = message`.

In this code example, we used the `apply` function. This is part of the family of scoping functions that can be invoked on every Kotlin object, and we'll cover them in detail in Chapter 9, *Idioms and Anti-Patterns*. The `apply` function returns the reference to an object after executing the block. So, it's a shorter version of the setter function from the previous example:

```
fun message(message: String): MailBuilder {

    this.message = message

    return this

}
```

This provides us with the same API as the previous example:

```
val mailV2 = Mail_V2(listOf("manager@company.com")).message("Ping")
```

However, we may not need setters at all. Instead, we can use the **`apply()`** function we previously discussed on the object itself. This is one of the extension functions that every object in Kotlin has. This approach will work only if all of the optional fields are *variables* instead of *values*.

Then, we can create our email like this:

```
val mail = Mail_V2("hello@mail.com").apply {

    message = "Something"

    title = "Apply"

}
```

This is a nice approach, and it requires less code to implement. However, there are a few downsides to this approach too:

- We had to make all of the optional arguments mutable. Immutable fields should always be preferred to mutable ones, as they are thread-safe and easier to reason about.
- All of our optional arguments are also nullable. Kotlin is a null-safe language, so every time we access them, we first have to check that

their value was set.

- This syntax is very verbose. For each field, we need to repeat the same pattern over and over again.

Now, let's discuss the last approach to this problem.

Default arguments

In Kotlin, we can specify default values for constructor and function parameters:

```
data class Mail_V3(

    val to: List<String>,

    val cc: List<String> = listOf(),

    val title: String = "",

    val message: String = "",

    val important: Boolean = false

)
```

Default arguments are set using the `=` operator after the type. This means that although our constructor still has all the arguments, we don't need to provide them any.

So, if you would like to create an email without a body, you can do it like this:

```
val mail = Mail_V3(listOf("manager@company.com"), listOf(), "Ping")
```

However, note that we had to specify that we don't want anyone in the CC field by providing an empty list, which is a bit inconvenient.

What if we wanted to send an email that is only flagged as important?

Not having to specify order with fluent setters was very handy. Kotlin has *named arguments* for that:

```
val mail = Mail_V3(title = "Hello", message = "There", to =
listOf("my@dear.cat"))
```

Combining default parameters with named arguments makes creating complex objects in Kotlin rather easy. For that reason, you will rarely need the Builder design pattern at all in Kotlin.

In real applications, you'll often see the Builder design pattern used to construct instances of servers. A server would accept an optional host and an optional port and so on, and then when all of the arguments were set, you'd invoke a listen method to start it.

Prototype

The **Prototype** design pattern is all about customization and creating objects that are similar but slightly different. To understand it better, Let's look at an example.

Imagine we have a system that manages users and their permissions. A **`data class`** representing a user might look like this:

```
data class User(

    val name: String,

    val role: Role,

    val permissions: Set<String>,

) {
    fun hasPermission(permission: String) = permission in
      permissions

}
```

Each user must have a role, and each role has a set of permissions.

We'll describe a role as an **`enum`** class:

```
enum class Role {

    ADMIN,

    SUPER_ADMIN,

    REGULAR_USER

}
```

The **`enum`** classes are a way to represent a collection of constants. This is more convenient than representing a role as a string, for example, as we check at compile time that such an object exists.

When we create a new *user*, we assign them permissions that are similar to another user with the same *role*:

```
// In real application this would be a database of users

val allUsers = mutableListOf<User>()

fun createUser(name: String, role: Role) {

    for (u in allUsers) {

        if (u.role == role) {

            allUsers += User(name, role, u.permissions)

            return

        }

    }

    // Handle case that no other user with such a role exists
}
```

Let's imagine that we now need to add a new field to the **`User`** class, which we will name **`tasks`**:

```
data class User(
    val name: String,
    val role: Role,
    val permissions: Set<String>,
    val tasks: List<String>,
) {
   ...
}
```

Our **`createUser`** function will stop compiling. We'll have to change it by copying the value of this newly added field to the new instance of our class:

```
allUsers += User(name, role, u.permissions, u.tasks)
```

This work will have to be repeated every time the **`User`** class is changed.

However, there's a bigger problem still: *What if a new requirement is introduced, making the* **`permissions`** *property, for example,* **`private`***?*

```
data class User(
    val name: String,
    val role: Role,
    private val permissions: Set<String>,
    val tasks: List<String>,
) {
   ...
}
```

Our code will stop compiling again, and we'll have to change it again. The constant requirement of changes to the code is a clear indication that we need another approach to solve this problem.

Starting from a prototype

The whole idea of a *prototype* is to be able to clone an object easily. There are at least two reasons you may want to do this:

- It helps in instances where creating your object is very expensive – for example, if you need to fetch it from the database.
- It helps if you need to create objects that are similar but vary slightly and you don't want to repeat similar parts over and over again.

  IMPORTANT NOTE:

  There are also more advanced reasons to use the Prototype design pattern. JavaScript, for example, uses prototypes to implement inheritance-like behavior without having classes.

Luckily, Kotlin fixes the somewhat broken Java **`clone()`** method. Data classes have a **`copy()`** method, which takes an existing **`data class`**, and creates a new copy of it, optionally changing some of its attributes in the process:

```
// Name argument is underscored here simply not to confuse
// it with the property of the same name in the User object
fun createUser(_name: String, role: Role) {
    for (u in allUsers) {
        if (u.role == role) {
            allUsers += u.copy(name = _name)
            return
        }
    }
    // Handle case that no other user with such a role exists
}
```

In a similar way to what we saw with the Builder design pattern, named arguments allow us to specify attributes that we can change in any order. And we need to specify only the attributes we want to change. All of the other data will be copied for us, even the **`private`** properties.

The **`data class`** is yet another example of a design pattern that is so common that it became part of a language syntax. They are an extremely useful feature, and we will see them being used many more times in this book.

Summary

In this chapter, we have learned when and how to use creational design patterns. We started by discussing how to use the **`object`** keyword to construct a singleton class, and then we discussed the use of **`companion object`** if you need a Static Factory Method. We also covered how to assign multiple variables at once using destructuring declarations.

Then, we discussed smart casts, and how they can be applied in the Abstract Factory design pattern to create families of objects. We then moved to the Builder design pattern and learned that functions can have default parameter values. We then learned that we can refer to their arguments using not only positions but also names.

Finally, we covered the `copy()` function of the data classes, and how it helps us when implementing the Prototype design pattern to produce similar objects with slight changes. You should now understand how to use creational design patterns to better manage your objects.

In the next chapter, we'll cover the second family of design patterns: **structural patterns**. These design patterns will help us create extensible and maintainable object hierarchies.

Questions

1. Name two uses for the `object` keyword we learned about in this chapter.
2. What is the `apply()` function used for?
3. Provide one example of a Static Factory Method.

Chapter 3: Understanding Structural Patterns

This chapter covers **structural patterns** in **Kotlin**. In general, structural patterns deal with relationships between **objects**.

We'll discuss how to extend the functionality of our objects without producing complex class hierarchies. We'll also discuss how to adapt to changes in the future or fix some of the design decisions taken in the past, as well as how to reduce the memory footprint of our program.

In this chapter, we will cover the following patterns:

- Decorator
- Adapter
- Bridge
- Composite
- Facade
- Flyweight
- Proxy

By the end of this chapter, you'll have a better understanding of how to compose your objects so that they can be simpler to extend and adapt to different types of changes.

Technical requirements

The requirements for this chapter are the same as the previous chapters—you'll need **IntelliJ IDEA** and the **JDK**.

You can find the code files for this chapter on GitHub at https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter03.

Decorator

In the previous chapter, we discussed the **Prototype** design pattern, which allows us to create instances of classes with slightly (or not so slightly) different data. This raises a question:

What if we want to create a set of classes that all have slightly different behavior?

Well, since functions in Kotlin are *first-class citizens* (which we will explain in this chapter), you could use the Prototype design pattern to achieve this aim. After all, creating a set of classes with slightly different behavior is what JavaScript does successfully. But the goal of this chapter is to discuss another approach to the same problem. After all, design patterns are all about *approaches*.

By implementing the **Decorator** design pattern, we allow the users of our code to specify the abilities they want to add.

Enhancing a class

Let's say that we have a rather simple class that registers all of the captains in the Star Trek universe along with their vessels:

```
open class StarTrekRepository {

    private val starshipCaptains = mutableMapOf("USS
        Enterprise" to "Jean-Luc Picard")

    open fun getCaptain(starshipName: String): String {

        return starshipCaptains[starshipName] ?: "Unknown"

    }

    open fun addCaptain(starshipName: String, captainName:
        String) {

        starshipCaptains[starshipName] = captainName

    }

}
```

One day, your captain—sorry, *scrum master*—comes to you with an urgent requirement. From now on, every time someone searches for a captain, we must also log this into a console. However, there's a catch to this simple task: you cannot modify the **`StarTrekRepository`** class directly. There are other consumers for this class, and they don't need this logging behavior.

But before we dive deeper into this problem, let's discuss one peculiarity we can observe in our class – that is, a strange operator you can see in the **`getCaptain`** function.

The Elvis operator

In *Chapter 1*, *Getting Started with Kotlin*, we learned that Kotlin is not only strongly typed, but it is also a null-safe language.

What happens if, as in our example, there could be no value stored in a map for a particular key?

If we're working with a map, one option is to use the `getOrDefault` method that maps provide in Kotlin. This might be a viable option in this particular case, but it won't work in situations where you might have to deal with a null value.

Another option is to use the **Elvis operator** (`?:`). If you're wondering about how this operator got its name, it does resemble Elvis Presley's hairstyle somewhat:

Figure 3.1 – If we turn the Elvis operator 90 degrees clockwise, it looks a bit like a pompadour hairstyle

The goal of the Elvis operator is to provide a default value in case we receive a null value. Take another look at the `getCaptain` function to see how this is done. The *desugared* form of the same function would be as follows:

```
return if (starshipCaptains[starshipName] == null)
    "Unknown" else starshipCaptains[starshipName]
```

So, you can see that this operator saves us a lot of typing.

The inheritance problem

Let's go back to the task at hand. Since our class and its methods are declared open, we can extend the class and override the function we need:

```
class LoggingGetCaptainStarTrekRepository :
  StarTrekRepository() {
    override fun getCaptain(starshipName: String): String {
        println("Getting captain for $starshipName")
        return super.getCaptain(starshipName)
    }
}
```

That was quite easy! Although the name of that class is getting quite long.

Note how we delegate to the implementation in our parent class by using the **`super`** keyword here. However, the next day, your boss (sorry, *scrum-master*) comes again and asks for another feature. When adding a captain, we need to check that their name is no longer than 15 characters. That may be a problem for some Klingons, but you decide to implement it anyway. And, by the way, this feature should not be related to the logging feature we developed previously. Sometimes we just want the logging, and sometimes we just want the validation. So, here's what our new class will look like:

```
class ValidatingAddCaptainStarTrekRepository :
  StarTrekRepository() {
    override fun addCaptain(starshipName: String,
        captainName: String) {
        if (captainName.length > 15) {
            throw RuntimeException("$captainName is longer
                than 20 characters!")
        }
        super.addCaptain(starshipName, captainName)
    }
}
```

Another task done.

However, the next day, another requirement arises: in some cases, we need **`StarTrekRepository`** to have logging enabled and also perform validation at the same time. I guess we'll have to name it **`LoggingGetCaptainValidatingAddCaptainStarTrekRepository`** now.

Problems like this are surprisingly common, and they are a clear indication that a design pattern may help us here.

The purpose of the Decorator design pattern is to add new behaviors to our objects dynamically. In our example, *logging* and *validating* are two behaviors that we sometimes want to be applied to our object and sometimes don't want to be applied.

We'll start by converting our **`StarTrekRepository`** into an interface:

```
interface StarTrekRepository {
    fun getCaptain(starshipName: String): String
    fun addCaptain(starshipName: String, captainName:
        String)
    }
```

Then, we'll implement that interface using the same logic as before:

```
class DefaultStarTrekRepository : StarTrekRepository {
    private val starshipCaptains = mutableMapOf("USS Enter
        prise" to "Jean-Luc Picard")
    override fun getCaptain(starshipName: String): String {
        return starshipCaptains[starshipName] ?: "Unknown"
    }

    override fun addCaptain(starshipName: String, captain
        Name: String) {

        starshipCaptains[starshipName] = captainName

    }

}
```

Next, instead of extending our concrete implementation, we'll implement the interface and use a new keyword called **`by`**:

```
class LoggingGetCaptain(private val repository:

  StarTrekRepository): StarTrekRepository by repository {

    override fun getCaptain(starshipName: String): String {

        println("Getting captain for $starshipName")

        return repository.getCaptain(starshipName)

    }

}
```

The **`by`** keyword delegates the implementation of an interface to another object. That's why the **`LoggingGetCaptain`** class doesn't have to implement any of the functions declared in the interface. They are all implemented by default by another object that the instance wraps.

In this case, the hardest part to understand is the signature. What we need from the Decorator design pattern is as follows:

- We need to be able to receive the object we're decorating.
- We need to be able to keep a reference to the object.

- When our decorator is called, we need to be able to decide if we would like to change the behavior of the object we're holding or to delegate the call.
- We need to be able to extract an interface or have one provided already by the (library) author.

Note that we don't use the **`super`** keyword anymore. If we tried to, it wouldn't work, as there is a class that we're implementing now. Instead, we use the reference to the **`wrapped`** interface.

To make sure we understand this pattern, let's implement our second decorator:

```
class ValidatingAdd(private val repository:

  StarTrekRepository): StarTrekRepository by repository {

    private val maxNameLength = 15

    override fun addCaptain(starshipName: String,

        captainName: String) {

        require (captainName.length < maxNameLength) {

            "$captainName name is longer than
                $maxNameLength characters!"

        }

        repository.addCaptain(starshipName, captainName)

    }

}
```

The only difference between the preceding example and the **`ValidatingAddCaptainStarTrekRepository`** implementation is that we use the **`require`** function instead of an **`if`** expression. This is often more readable, and it will also throw **`IllegalArgumentException`** if the expression is **`false`**.

Let's see how it works now:

```
val starTrekRepository = DefaultStarTrekRepository()

val withValidating = ValidatingAdd(starTrekRepository)

val withLoggingAndValidating =
    LoggingGetCaptain(withValidating)

withLoggingAndValidating.getCaptain("USS Enterprise")

withLoggingAndValidating.addCaptain("USS Voyager",    "Kathryn
Janeway")
```

The last line will throw an exception:

```
> Kathryn Janeway name is longer than 15 characters!
```

As you can see, this pattern allows us to *compose behavior*, just as we wanted. Now, let's take a short detour and discuss *operator overloading* in Kotlin, as this will help us to improve our design pattern even more.

Operator overloading

Let's take another look at the interface that was extracted. Here, we are describing basic operations on a map that are usually associated with array/map access and assignment. In Kotlin, we have some nice syntactic sugar called **operator overloading**. If we look at **`DefaultStarTrekRepository`**, we can see that working with maps is very intuitive in Kotlin:

```
starshipCaptains[starshipName]

starshipCaptains[starshipName] = captainName
```

It would be useful if we could work with our repository as if it was a map:

```
withLoggingAndValidating["USS Enterprise"]

withLoggingAndValidating["USS Voyager"] = "Kathryn Janeway"
```

Using Kotlin, we can actually achieve this behavior quite easily. First, let's change our interface:

```
interface StarTrekRepository {

    operator fun get(starshipName: String): String

    operator fun set(starshipName: String, captainName:
        String)

}
```

Note that we've added the `operator` keyword that prefixes the function definition. Let's understand what this keyword means.

Most programming languages support some form of operator overloading. Let's take **Java** as an example and look at the following two lines:

```
System.out.println(1 + 1); // Prints 2

System.out.println("1" + "1") // Prints 11
```

We can see that the **+** operator acts differently depending on whether the arguments are strings or integers. That is, it can add two numbers, but it can also concatenate two strings. You can imagine that the *plus* operation can be defined on other types. For example, it makes a lot of sense to concatenate two lists using the same operator:

```
List.of("a") + List.of("b")
```

Unfortunately, this code won't compile in Java, and we can't do anything about it. That's because operator overloading is a feature reserved to the language itself, and not for its users.

Let's look at another extreme, the **Scala** programming language. In Scala, any set of characters can be defined as an operator. So, you may encounter code such as the following:

```
Seq("a") ==== Seq("b") // You'll have to guess what   this code
does
```

Kotlin takes a middle ground between these two approaches. It allows you to overload certain *well-known* operations, but it limits what can and cannot be overloaded. Although this list is limited, it is quite long, so we'll not write it in full here. However, you can find it in the official Kotlin documentation: https://kotlinlang.org/docs/operator-overloading.html.

If you use the **`operator`** keyword with a function that is unsupported or with the wrong set of arguments, you'll get a compilation error. The square brackets that we started with in the previous code example are called indexed access operators and correlate with the **`get(x)`** and **`set(x, y)`** methods we have just defined.

Caveats of the Decorator design pattern

The Decorator design pattern is great because it lets us compose objects *on the fly*. And using Kotlin's **`by`** keyword makes it easy to implement. But there are still limitations that you need to be aware of.

First, you cannot see *inside* of the Decorator. This means that there's no way of knowing which specific object it wraps:

```
println(withLoggingAndValidating is LoggingGetCaptain)

// This is our top level decorator, no problem here

println(withLoggingAndValidating is StarTrekRepository)
```

```
// This is the interface we implement, still no problem
println(withLoggingAndValidating is ValidatingAdd)
// We wrap this class, but compiler cannot validate it
println(withLoggingAndValidating is DefaultStarTrekRepository)
// We wrap this class, but compiler cannot validate it
```

Although **`withLoggingAndValidating`** contains **`ValidatingAdd`** (and it may behave like it), it is not an instance of **`ValidatingAdd`**! Keep that in mind when performing casts and type checks.

So, you might wonder where this pattern would be used in the real world. One example is the **`java.io.*`** package, with classes implementing the **`Reader`** and **`Writer`** interfaces.

For example, if you want to read a file efficiently, you can use **`BufferedReader`**, which receives another reader as its constructor argument:

```
val reader = BufferedReader(FileReader("/some/file"))
```

`FileReader` serves this purpose, as it implements the **`Reader`** interface. So does **`BufferedReader`** itself.

Let's move on to our next design pattern.

Adapter

The main goal of the **Adapter** design pattern is to convert one interface to another interface. In the physical world, the best example of this idea would be an electrical plug adapter or a USB adapter.

Imagine yourself in a hotel room late in the evening, with 7% battery left on your phone. Your phone charger was left in the office at the other end of the city. You only have an EU plug charger with a Mini USB cable. But your phone uses USB-C, as you had to upgrade. You're in New York, so all of your outlets are (of course) USB-A. So, what do you do? Oh, it's easy. You look for a Mini USB to USB-C adapter in the middle of the night and hope that you have remembered to bring your EU to US plug adapter as well. Only 5% battery left – time is running out!

So, now that we understand what adapters are for in the physical world, let's see how we can apply the same principle in code.

Let's start with interfaces.

`USPlug` assumes that power is **`Int`**. It has **`1`** as its value if it has power and any other value if it doesn't:

```
interface USPlug {

    val hasPower: Int

}
```

`EUPlug` treats power as **`String`**, which is either **`TRUE`** or **`FALSE`**:

```
interface EUPlug {

    val hasPower: String // "TRUE" or "FALSE"

}
```

For **`UsbMini`**, power is an **`enum`**:

```
interface UsbMini {

    val hasPower: Power

}

enum class Power {

    TRUE, FALSE
}
```

Finally, for **`UsbTypeC`**, power is a **`Boolean`** value:

```
interface UsbTypeC {

    val hasPower: Boolean

}
```

Our goal is to bring the power value from a US power outlet to our cellphone, which will be represented by this function:

```
fun cellPhone(chargeCable: UsbTypeC) {

    if (chargeCable.hasPower) {

        println("I've Got The Power!")

    } else {

        println("No power")

    }

}
```

Let's start by declaring what a US power outlet will look like in our code. It will be a function that returns a **`USPlug`**:

```
// Power outlet exposes USPlug interface

fun usPowerOutlet(): USPlug {

    return object : USPlug {

        override val hasPower = 1

    }

}
```

In the previous chapter, we discussed two different uses of the **`object`** keyword. In the global scope, it creates a Singleton object. When used together with the **`companion`** keyword inside of a class, it creates a place for defining **`static`** functions. The same keyword can also be used to generate anonymous classes. Anonymous classes are classes that are created *on the fly*, usually to implement an interface in an ad-hoc manner.

Our charger will be a function that takes **`EUPlug`** as an input and outputs **`UsbMini`**:

```
// Charger accepts EUPlug interface and exposes UsbMini

// interface

fun charger(plug: EUPlug): UsbMini {

    return object : UsbMini {

        override val hasPower=Power.valueOf(plug.hasPower)

    }

}
```

Next, let's try to combine our **`cellPhone`**, **`charger`**, and **`usPowerOutlet`** functions:

```
cellPhone(

    // Type mismatch: inferred type is UsbMini but     // UsbTypeC
was expected

    charger(

        // Type mismatch: inferred type is USPlug but         //
EUPlug was expected

        usPowerOutlet()

    )

)
```

As you can see, we get two different type errors – the Adapter design pattern should help us solve these.

Adapting existing code

We need two types of adapters: one for our power plugs and another one for our USB ports.

In Java, you would usually create a pair of classes for this purpose. In Kotlin, we can replace these classes with **extension functions**. We already mentioned extension functions briefly in *Chapter 1*, *Getting Started with Kotlin*. Now, it's time to cover them in more detail.

We could adapt the US plug to work with the EU plug by defining the following extension function:

```
fun USPlug.toEUPlug(): EUPlug {
    val hasPower = if (this.hasPower == 1) "TRUE" else
      "FALSE"
    return object : EUPlug {
        // Transfer power
        override val hasPower = hasPower
    }
}
```

The **`this`** keyword in the context of an extension function refers to the object we're extending – just as if we were implementing this method inside of the class definition. Again, we use an anonymous class to implement the required interface on the fly.

We can create a USB adapter between the Mini USB and USB-C instances in a similar way:

```
fun UsbMini.toUsbTypeC(): UsbTypeC {
    val hasPower = this.hasPower == Power.TRUE
    return object : UsbTypeC {
        override val hasPower = hasPower
    }
}
```

Finally, we can get back online again by combining all those adapters together:

```
cellPhone(
    charger(
        usPowerOutlet().toEUPlug()
    ).toUsbTypeC()
)
```

As you can see, we didn't have to create any new classes that implement these interfaces. By using Kotlin's extension functions, our code stays short and to the point.

The Adapter design pattern is more straightforward than the other design patterns, and you'll see it used widely. Now, let's discuss some of its real-world uses in more detail.

Adapters in the real world

You've probably encountered many uses of the Adapter design pattern already. These are normally used to adapt between *concepts* and *implementations*. For example, let's take the concept of a JVM collection versus the concept of a JVM stream.

We already discussed **collections** in Chapter 1, *Getting Started with Kotlin*. A **list** is a collection of elements that can be created using the **`listOf()`** function:

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

A **stream** is a *lazy* collection of elements. You cannot simply pass a collection to a function that receives a stream, even though it may make sense:

```
fun printStream(stream: Stream<String>) {

    stream.forEach(e -> println(e))

}

printStream(list) // Doesn't compile
```

Luckily, collections provide us with the **`.stream()`** adapter method:

```
printStream(list.stream()) // Adapted successfully
```

Many other Kotlin objects have adapter methods that usually start with **`to`** as a prefix. For example, **`toTypedArray()`** converts a list to an array.

Caveats of using adapters

Have you ever plugged a 110 V US appliance into a 220 V EU socket through an adapter, and fried it totally?

If you're not careful, that's something that could also happen to your code. The following example uses another adapter, and it also compiles well:

```
val stream = Stream.generate { 42 }

stream.toList()
```

But it never completes because **`Stream.generate()`** produces an infinite list of integers. So, be careful and adopt this design pattern wisely.

Bridge

While the Adapter design pattern helps you to work with legacy code, the **Bridge** design pattern helps you to avoid abusing inheritance. The way it works is actually very simple.

Let's imagine we want to build a system to manage different kinds of troopers for the Galactic Empire.

We'll start with an interface:

```
interface Trooper {

    fun move(x: Long, y: Long)

    fun attackRebel(x: Long, y: Long)

}
```

And we'll create multiple implementations for different types of troopers:

```
class StormTrooper : Trooper {

    override fun move(x: Long, y: Long) {

        // Move at normal speed

    }

    override fun attackRebel(x: Long, y: Long) {

        // Missed most of the time

    }

}

class ShockTrooper : Trooper {
    override fun move(x: Long, y: Long) {

        // Moves slower than regular StormTrooper

    }

    override fun attackRebel(x: Long, y: Long) {

        // Sometimes hits

    }

}
```

There are also stronger versions of them:

```
class RiotControlTrooper : StormTrooper() {

    override fun attackRebel(x: Long, y: Long) {

        // Has an electric baton, stay away!

    }

}

class FlameTrooper : ShockTrooper() {

    override fun attackRebel(x: Long, y: Long) {

        // Uses flametrower, dangerous!

    }

}
```

And there are also scout troopers that can run faster than the others:

```
class ScoutTrooper : ShockTrooper() {

    override fun move(x: Long, y: Long) {

        // Runs faster

    }

}
```

That's a lot of classes!

One day, our dear designer comes and asks that all stormtroopers should be able to shout, and each will have a different phrase. Without thinking twice, we add a new function to our interface:

```
interface Infantry {

    fun move(x: Long, y: Long)

    fun attackRebel(x: Long, y: Long)

    fun shout(): String

}
```

By doing that, all the classes that implement this interface stop compiling. And we have a lot of them. That's a lot of changes that we'll have to make. So, we'll just have to suck it up and get to work.

Or will we?

We go and change the implementations of five different classes, feeling lucky that there are only five and not fifty.

Bridging changes

The idea behind the Bridge design pattern is to flatten the class hierarchy and have fewer specialized classes in our system. It also helps us to avoid the *fragile base class* problem when modifying the superclass introduces subtle bugs to classes that inherit from it.

First, let's try to understand why we have this complex hierarchy and many classes. It's because we have two orthogonal, unrelated properties: *weapon type* and *movement speed*.

Let's say that instead, we wanted to pass those properties to the constructor of a class that implements the same interface we have been using all along:

```
data class StormTrooper(
    private val weapon: Weapon,
    private val legs: Legs
) : Trooper {
    override fun move(x: Long, y: Long) {
        legs.move(x, y)
    }
    override fun attackRebel(x: Long, y: Long) {
        weapon.attack(x, y)
    }
}
```

The properties that **`StormTrooper`** receives should be interfaces, so we can choose their implementation later:

```
typealias PointsOfDamage = Long
typealias Meters = Int
interface Weapon {
    fun attack(): PointsOfDamage
}
interface Legs {
    fun move(): Meters
}
```

Notice that these methods return **`Meters`** and **`PointsOfDamage`** instead of simply returning **`Long`** and **`Int`**. This feature is called **type aliasing**. To understand how this works, let's take a short detour.

Type aliasing

Kotlin allows us to provide alternative names for existing types. These are called **aliases**.

To declare an alias, we use a new keyword: **`typealias`**. From now on, we can use **`Meters`** instead of plain old **`Int`** to return from our **`move()`** method. These aren't new types. The Kotlin compiler will always translate **`PointsOfDamage`** to **`Long`** during compilation. Using them provides two advantages:

- The first advantage is *better semantics* (as in our case). We can tell exactly what the *meaning* of the value we're returning is.
- The second advantage is being *concise*. Type aliases allow us to hide complex generic expressions. We'll expand on this in the following sections.

Constants

Let's go back to our **`StormTrooper`** class. It's time to provide some implementations for the **`Weapon`** and **`Legs`** interfaces.

First, let's define the regular damage and speed of **`StormTrooper`**, using imperial units:

```
const val RIFLE_DAMAGE = 3L

const val REGULAR_SPEED: Meters = 1
```

These values are very effective since they are known during compilation.

Unlike **`static final`** variables in Java, they cannot be placed inside a class. You should place them either at the top level of your package or nest them inside of an object.

IMPORTANT NOTE:

Although Kotlin has type inference, we can specify the types of our constants explicitly and even use type aliases. How about having ***`DEFAULT_TIMEOUT : Seconds = 60`*** *instead of* ***`DEFAULT_TIMEOUT_SECONDS = 60`*** *in your code?*

Now, we can provide some implementations for our interfaces:

```
class Rifle : Weapon {

    override fun attack(x: Long, y: Long) = RIFLE_DAMAGE

}

class Flamethrower : Weapon {

    override fun attack(x: Long, y: Long)= RIFLE_DAMAGE * 2

}

class Batton : Weapon {

    override fun attack(x: Long, y: Long)= RIFLE_DAMAGE * 3

}
```

Next, let's look at how we can move the following:

```
class RegularLegs : Legs {

    override fun move() = REGULAR_SPEED

}

class AthleticLegs : Legs {

    override fun move() = REGULAR_SPEED * 2

}
```

Finally, we need to make sure that we can implement the same functionality without the complex class hierarchy we had before:

```
val stormTrooper = StormTrooper(Rifle(), RegularLegs())

val flameTrooper = StormTrooper(Flamethrower(),     RegularLegs())

val scoutTrooper = StormTrooper(Rifle(), AthleticLegs())
```

Now we have a flat class hierarchy, which is much simpler to extend and also to understand. If we need more functionality, such as the shouting ability we mentioned earlier, we would add a new interface and a new constructor argument for our class.

In the real world, this pattern is often used in conjunction with dependency injection frameworks. For example, this would allow us to replace an implementation that used a real database with a mocked interface. This would make our code easier to set up and faster to test.

Composite

This chapter is dedicated to composing objects within one another, so it may look strange to have a separate section for the **Composite** design pattern. As a result, this raises a question:

Shouldn't this design pattern encompass all of the others?

As in the case of the Bridge design pattern, the name may not reflect its true uses and benefits.

Let's continue with our `StormTrooper` example from before. Lieutenants of the Empire quickly discover that no matter how well equipped, stormtroopers cannot hold their ground against the rebels because they are uncoordinated.

To provide better coordination, the Empire decides to introduce the concept of a *squad* for the stormtroopers. A squad should contain one or more stormtrooper of any kind, and when given commands, it should behave exactly as if it was a single unit.

`Squad`, clearly, consists of a collection of stormtroopers:

```
class Squad(val units: List<Trooper>)
```

Let's add a couple of them to begin with:

```
val bobaFett = StormTrooper(Rifle(), RegularLegs())

val squad = Squad(listOf(bobaFett.copy(), bobaFett.copy(),
bobaFett.copy()))
```

To make our squad act as if it was a single unit, we'll add two methods to it called **`move`** and **`attack`**:

```
class Squad(private val units: List<Trooper>) {

    fun move(x: Long, y: Long) {

        for (u in units) {

            u.move(x, y)

        }

    }

    fun attack(x: Long, y: Long) {

        for (u in units) {

            u.attackRebel(x, y)

        }

    }
}
```

Both functions will repeat any received orders to all of the units they contain. At first, the approach seems to be working. However, what happens if we change our **`Trooper`** interface by adding a new function? Consider the following code:

```
interface Trooper {

    fun move(x: Long, y: Long)

    fun attackRebel(x: Long, y: Long)

    fun retreat()

}
```

Nothing seems to break, but our **`Squad`** class stops doing what it was supposed to do – that is, act as if it was a single unit. A single unit now has a method that our composite class does not.

In order to prevent this from happening in the future, let's see what happens if our **`Squad`** class implements the same interface as the units it contains:

```
class Squad(private val units: List<StormTrooper>):  Trooper { ...
}
```

That change will force us to implement the **`retreat`** function and mark the other two functions with the **`override`** keyword:

```
class Squad(private val units: List<StormTrooper>): Trooper {

    override fun move(x: Long, y: Long) {

        ...

    }

    override fun attackRebel(x: Long, y: Long) {

        ...
    }
    override fun retreat() {
        ...
    }
}
```

Now, we'll take a short detour to discuss an alternative and more convenient approach to this example – one that would allow us to construct the same object but result in a composite that is more pleasant to use.

Secondary constructors

Our code did achieve its goals. However, it would be good if instead of passing a list of stormtroopers to the constructor (as we do now), we could pass our stormtroopers directly, without wrapping them in a list:

```
val squad = Squad(bobaFett.copy(), bobaFett.copy(),
  bobaFett.copy())
```

One way to achieve this is to add **secondary constructors** to our **`Squad`** class.

Up until now, we were always using the *primary constructor* of the class. That's the constructor declared after the class name. But we can define more than one constructor for a class. We can define secondary constructors for a class using the **`constructor`** keyword inside the class body:

```
class Squad(private val units: List<Trooper>): Trooper {
    constructor(): this(listOf())
    constructor(t1: Trooper): this(listOf(t1))
    constructor(t1: Trooper, t2: Trooper): this(listOf(t1,
      t2))
}
```

Unlike Java, there's no need to repeat the class name for each constructor. That also means fewer changes are required if you decide to rename the class.

Note how each secondary constructor must call the primary constructor. This is similar to using the **`super`** keyword in Java.

The varargs keyword

This is clearly not the way to go, since we cannot predict how many more elements someone might want to pass us. If you come from Java, you have probably thought about **variadic functions** already, which can take an arbitrary number of arguments of the same type. In Java, you would declare the parameter using an ellipsis: **`Trooper... units`**.

Kotlin provides us with the **`vararg`** keyword for the same purpose. By combining a secondary constructor with **`varargs`**, we get the following piece of code, which is very nice:

```
class Squad(private val units: List<Trooper>): Trooper {
    constructor(vararg units: Trooper):
        this(units.toList())
    ...
}
```

Now, we are able to create a squad with any number of stormtroopers without the need to wrap them in a list first:

```
val squad = Squad(bobaFett.copy(), bobaFett.copy(),
bobaFett.copy())
```

Let's try to understand how this works under the hood. The Kotlin compiler translates a **`vararg`** argument to an **`Array`** of the same type:

```
constructor(units: Array<Trooper>) : this(units.toList())
```

Arrays in Kotlin have an Adapter method that allows them to be converted to a list of the same type. Interestingly, we can use the Adapter design pattern to help us implement the Composite design pattern.

Nesting composites

The Composite design pattern has another interesting property. Previously, we proved that we can create a squad containing multiple stormtroopers. We can also create a squad of squads:

```
val platoon = Squad(Squad(), Squad())
```

Now, giving an order to the platoon will work in exactly the same way as giving it to a squad. In fact, this pattern allows us to support a tree-like structure of arbitrary complexity and to perform operations on all of its nodes.

The Composite design pattern may seem a bit incomplete until we reach the next chapter, where we will discover its partner: the **Iterator** design pattern. When both design patterns are combined, they really shine. If you are still unsure how this pattern is useful after completing this section, come back to it after you have also learned about the Iterator design pattern.

In the real world, the Composite design pattern is widely used in **user interface** (**UI**) frameworks. For example, the `Group` widget in **Android** is an implementation of the Composite design pattern. It can group multiple other elements and implement the `View` interface in order to be able to act on their behalf.

As long as all the objects in the hierarchy implement the same interface, no matter how deep the nesting is, we can ask the top-level object to invoke an action on everything beneath it.

Facade

The use of *facade* as a term to refer to a design pattern comes directly from building architecture. That is, a facade is the face of a building that is normally made to look more appealing than the rest of it. In programming, *facades* can help to hide the ugly details of an implementation.

The **Facade** design pattern itself aims to provide a nicer, simpler way to work with a family of classes or interfaces. We previously discussed the idea of a family of classes when covering the **Abstract Factory** design pattern. The Abstract Factory design pattern focuses on creating related classes, while the Facade design pattern focuses on working with them once they have been created.

To better understand this, let's go back to the example we used for the Abstract Factory design pattern. In order to be able to start our server from a configuration using our Abstract Factory, we could provide users of our library with a set of instructions:

- Check if the given file is `.json` or `.yaml` by trying to parse it with a **JSON** parser.
- If we received an error, try parsing it using a **YAML** parser.
- If there were no errors, pass the results to the Abstract Factory to create the necessary objects.

While helpful, following this set of instructions may require quite a bit of skill and knowledge. Developers may struggle to find the correct parser, or they might ignore any exceptions thrown from a JSON parser in instances where it's dealing with a `.yaml` file, for example.

What problems are our users facing at the moment?

To load a configuration, they will need to interact with at least three different interfaces:

- A JSON parser (covered in the *Abstract Factory* section in *Chapter 2*, *Working with Creational Patterns*)
- YAML Parser (covered in the *Abstract Factory* section in *Chapter 2*, *Working with Creational Patterns*)
- Server Factory (covered in the *Factory Method* section in *Chapter 2*, *Working with Creational Patterns*)

Instead, it would be great to have a single function (`startFromConfiguration()`) that would take a path to a configuration file, parse it, and then, if there were no errors in the process, start our server.

We'll be providing a *facade* to our users to simplify working with a set of classes. One way to achieve this goal would be to provide a new class to encapsulate all of this logic for us. This is a common tactic in most languages.

However, in Kotlin, we have a better option that uses a technique we already discussed in this chapter when covering the Adapter design pattern. We can make **`startFromConfiguration()`** an *extension function* on the **`Server`** class:

```
@ExperimentalPathApi

fun Server.startFromConfiguration(fileLocation: String) {

    val path = Path(fileLocation)

    val lines = path.toFile().readLines()

    val configuration = try {

        JsonParser().server(lines)

    }

    catch (e: RuntimeException) {

        YamlParser().server(lines)

    }

    Server.withPort(configuration.port)

}
```

You can see that this implementation is exactly the same as in the Adapter design pattern. The only difference is the end goal. In the case of the Adapter design pattern, the goal is to make an otherwise *unusable* class *usable*. Remember, one of the goals of the Kotlin language is to *reuse* as much as possible. For the Façade design pattern, the goal is to make a *complex* group of classes *easy to use*.

IMPORTANT NOTE:

Depending on when you read this book, you may not need the **`ExperimentalPathApi`** *annotation anymore. This feature was introduced in Kotlin 1.4, and once it is stable it will be made an integral part of the language.*

We already discussed that in Kotlin, **`try`** is an *expression* that returns a *value*. Here, you can see that we can also return a value from a **`catch`** block, further reducing the need for mutable variables.

Next, let's understand what happens in the first two lines of this function. **`Path`** is a rather new API that was introduced in **Kotlin 1.4**. It aims to simplify working with files. Notice that **`toFile`** is an example of the Adapter design pattern that converts between a path and an actual file. Finally, the **`readLine()`** function will attempt to read the entire file into memory, split line by line. Consider using the Facade design pattern when working with any code base that would benefit from being simplified.

Flyweight

Flyweight is an object without any state. The name comes from it being *very light*. If you've been reading either one of the two previous chapters, you might already be thinking of a type of object that should be very light: a **`data`** class. But a **`data`** class is all about state.

So, is the data class related to the Flyweight design pattern at all?

To understand this design pattern better, we need to jump back in time some twenty years. Back in 1994, when the original *Design Patterns* book was published, your regular PC had 4 MB of RAM. During this period, one of the main goals of any process was to save that precious RAM, as you could fit only so much into it.

Nowadays, some *cellphones* have 8 GB of RAM. Bear that in mind when we discuss what the Flyweight design pattern is all about in this section.

Having said that, let's see how we can use our resources more efficiently, as this is always important!

Being conservative

Imagine we're building a 2D side-scrolling arcade platform game. That is, you have your game character, which you control with arrow keys or a gamepad. Your character can move left, right, and jump.

Since we're a really small indie company consisting of one developer (who is also a graphic designer, product manager, and sales representative), two cats, and a canary named Michael, we use only 16 colors in our game. And our character is 64 pixels tall and 64 pixels wide.

Our character has a lot of enemies, which consist mostly of carnivorous Tanzanian snails:

```
class TanzanianSnail
```

Since it's a 2D game, each snail has only two directions of movement: **`LEFT`** and **`RIGHT`**. We can represent these directions using an **`enum`** class:

```
enum class Direction {
   LEFT,
   RIGHT
}
```

To be able to draw itself on a screen, each snail will hold a pair of images and a direction:

```
class TansanianSnail {

    val directionFacing = Direction.LEFT

    val sprites = listOf(File("snail-left.jpg"),

                         File("snail-right.jpg"))

    // More information about the state of a snail comes

        here

    // This may include its health, for example

}
```

IMPORTANT NOTE:

*The definition of the **`File`** class comes from **`java.io.File`**. Remember that you can always refer to our GitHub project to see the needed imports.*

Based on the direction, we can get the current sprite that shows us which direction the snail is facing and use this to draw it:

```
fun getCurrentSprite(): File {

    return when (directionFacing) {

        Direction.LEFT -> sprites[0]

        Direction.RIGHT -> sprites[1]

    }

}
```

When any of the enemies move, they basically just slide left or right.

What we would like is to have multiple animated sprites to reproduce the snail's movements in each direction. We can generate a list of such sprites for each snail enemy using a **`List`** generator:

```
class TansanianSnail {

    val directionFacing = Direction.LEFT

    val sprites = List(8) { i ->

        File(when(i) {

            0 -> "snail-left.jpg"

            1 -> "snail-right.jpg"

            in 2..4 -> "snail-move-left-${i-1}.jpg"

            else -> "snail-move-right${(4-i)}.jpg"

        })

    }

}
```

Here, we initialize a list of eight elements, passing a **`block`** function as a constructor. The benefit of this approach is that we can apply complex logic during the creation of a collection while still keeping it effectively immutable.

For each element, we decide what image to get:

- Positions **`0`** and **`1`** are for still images, facing left and right.
- Positions **`2`** through **`4`** are for moving left.
- Positions **`5`** through **`7`** are for moving right.

Let's do some math now. Each snail is represented by a 64 x 64 image. Assuming each color takes up exactly one byte, the single images will take up 4 KB of RAM in the memory. Since we have eight images for a snail, we need 32 KB of RAM for each one, which allows us to fit only 32 snails into 1 MB of memory.

Since we want to have thousands of these dangerous and extremely fast creatures on screen and to be able to run our game on a 10-year-old phone, we clearly need a better solution.

Saving memory

What's the problem we have with all of our snails?

They're actually quite fat, heavyweight snails. We would like to put them on a diet. Each snail stores eight images within its *snaily* body. But these images are actually the same for each snail. This raises a question:

What if we extract those sprites into a Singleton object or a Factory Method and then only reference them from each instance?

For example, consider the following code:

```
object SnailSprites {
    val sprites = List(8) { i ->
        java.io.File(when (i) {
            0 -> "snail-left.jpg"
            1 -> "snail-right.jpg"
            in 2..4 -> "snail-move-left-${i-1}.jpg"
            else -> "snail-move-right${(4-i)}.jpg"
        })
    }
}

class TansanianSnail() {

    val directionFacing = Direction.LEFT

    val sprites = SnailSprites.sprites

}
```

This way, our **`getCurrentSprite`** function could stay the same, and we'll only consume 256 KB of memory, no matter how many snails we generate. We could generate millions of them without affecting the footprint of our program.

And this is exactly the idea behind the Flyweight design pattern. That is, limit the number of heavyweight objects (in our case, the image files) by sharing them between the lightweight objects (in our case, the snails).

Caveats of the Flyweight design pattern

We should take extra care about the immutability of the data we pass. If, for example, we used **`var`** instead of **`val`** in our Singleton, it could be disastrous for our code. The same goes for mutable data structures. We wouldn't want someone removing an image, replacing it, or clearing the list of images altogether.

Luckily, Kotlin makes handling these cases rather easy. Just make sure to always use values instead of variables in your extrinsic state, and remember to use immutable data structures, which cannot be altered after they have been created.

You can debate the usefulness of this pattern in this era of plentiful memory. However, as we have already said, the tools in the toolbox don't take up much space, and having another design pattern under your belt may still prove useful.

Proxy

Much like the Decorator design pattern, the **Proxy** design pattern extends an object's functionality. However, unlike a decorator, which always does what it's told, having a proxy may mean that when asked to do something, the object does something totally different.

When we discussed **Creational Patterns** in *Chapter 2*, *Working with Creational Patterns*, we already touched on the idea of *expensive* objects. For example, an object that accesses network resources or takes a lot of time to create.

We at the *Funny Cat App* provide our users with funny cat images on a daily basis. On our homepage and mobile application, each user sees a lot of pictures of funny cats. When they click or touch any of those images, it expands to its full-screen glory.

Fetching cat images over the network is very expensive, and it consumes a lot of memory, especially if those are images of cats that tend to indulge in a second dessert after dinner. What we want to do is fetch the full-sized image only once at the time it is requested. And if it is requested multiple times, we want to be able to show it to family or friends. In short, we don't want to have to fetch it every time.

There's no way to avoid loading the image once. But when it's being accessed for the second time, we would like to avoid going over the network again and instead return the result that was cached in memory. That's the idea of the **Proxy** design pattern; instead of the expected behavior of going over the network each time, we're being a bit lazy and returning the result that we already prepared.

It's a bit like going into a cheap diner, ordering a hamburger, and getting it after only two minutes, but cold. Well, that's because someone else hated onions and returned it to the kitchen a while ago. True story.

This sounds like it would require a lot of logic. But as you've probably guessed (especially after meeting the Decorator design pattern), Kotlin can perform miracles by reducing the amount of boilerplate code you need to write to achieve your goals:

```
data class CatImage(val thumbnailUrl: String,
        val url: String) {
    val image: ByteArray by lazy {
        // Read image as bytes
        URL(url).readBytes()
    }
}
```

Previously, we've seen the `by` keyword in a different context – that is, when delegating the implementation of an interface to another class (as discussed in *The Decorator design pattern* section of this chapter).

As you may have noticed, in this case, we use the `by` keyword to delegate the initialization of a field to happen later. We use a function called `lazy`, which is one of the **delegator functions** in the Kotlin standard library. At the first call to the `image` property, it will execute our code block and save its results into the `image` property. The following invocations of that property will simply return its value.

Sometimes, the Proxy design pattern is divided into three sub-patterns:

- **Virtual proxy**: Lazily caches the result
- **Remote proxy**: Issues a call to the remote resource
- **Protection or access control proxy**: Denies access to unauthorized parties

You can regard our previous example as either a virtual proxy or a combination of the virtual and remote types of proxies.

Lazy delegation

You may wonder what happens if two threads try to initialize the image at the same time. By default, the `lazy()` function is synchronized. Only one thread will win, and others will wait until the image is ready.

If you don't mind two threads executing the lazy block (for example, if it's not that expensive), you can use `lazy(LazyThreadSafetyMode.PUBLICATION)` instead.

If performance is absolutely critical for you and you're absolutely sure that two threads won't ever execute the same block simultaneously, you can use `LazyThreadSafetyMode.NONE`, which is not thread-safe.

Proxying and delegation is a very useful approach for many complex problems, and we'll explore this in the following chapters.

Summary

In this chapter, we have learned how structural design patterns can help us to create more flexible code that can adapt to changes with ease, sometimes even at runtime. We've covered how we can add functionality to an existing class with the Decorator design pattern, and we've explored how *operator overloading* can allow us to provide more intuitive syntax to common operations.

We then learned how to adapt one interface to another interface using extension methods, and we also learned how to create anonymous objects to implement an interface only once. Next, we discussed how to simplify class hierarchies using the Bridge design pattern. You should now know how to create a shortcut for a type name with **`typealias`** and also how to define efficient constants with **`const`**.

Moving on, we looked at the Composite design pattern, and we considered how it could help you to design a system that needs to treat groups of objects and regular objects in the same way. We also learned about secondary constructors and how a function can receive an *arbitrary number of arguments* when using the **`vararg`** keyword. We learned how the Facade design pattern helps us to simplify working with complex systems by exposing a simple interface, while the Flyweight design pattern allows us to reduce the memory footprint of our application.

Finally, we've covered how delegating to another class works in Kotlin, implementing the same interface and using the `by` keyword in the Proxy design pattern and demonstrating its use with a `lazy` delegate. With these design patterns, you should be able to structure your system in a much more extensible and maintainable manner.

In the next chapter, we'll discuss the third family of classic design patterns: behavioral patterns.

Questions

1. What differences are there between the implementations of the Decorator and Proxy design patterns?
2. What is the main goal of the Flyweight design pattern?
3. What is the difference between the Facade and Adapter design patterns?

Chapter 4: Getting Familiar with Behavioral Patterns

This chapter discusses behavioral patterns in terms of Kotlin. **Behavioral patterns** deal with how objects interact with one another.

We'll learn how an object can alter its behavior based on the situation, how objects can communicate without knowledge of one another, and how to iterate over complex structures easily. We'll also touch on the concept of functional programming in Kotlin, which will help us implement some of these patterns easily.

In this chapter, we will cover the following topics:

- Strategy
- Iterator
- State
- Command
- Chain of Responsibility
- Interpreter
- Mediator
- Memento
- Visitor
- Template method
- Observer

By the end of this chapter, you'll be able to structure your code in a highly decoupled and flexible manner.

Technical requirements

In addition to the requirements from the previous chapters, you will also need a **Gradle**-enabled **Kotlin** project to be able to add the required dependencies.

You can find the source code for this chapter here: https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter04.

Strategy

The goal of the **Strategy** design pattern is to allow an object to alter its behavior at runtime.

Let's recall the platformer game we were designing in *Chapter 3*, *Understanding Structural Patterns*, while discussing the **Facade** design pattern.

Canary Michael, who acts as a game designer in our small indie game development company, came up with a great idea. *What if we were to give our hero an arsenal of weapons to protect us from those horrible carnivorous snails?*

Weapons all shoot projectiles (you don't want to get too close to those dangerous snails) in the direction our hero is facing:

```
enum class Direction {
    LEFT, RIGHT
}
```

All projectiles should have a pair of coordinates (*our game is 2D, remember?*) and a direction:

```
data class Projectile(private var x: Int,
                      private var y: Int,
                      private var direction: Direction)
```

If we were to shoot only one type of projectile, that would be simple, since we covered the Factory pattern in Chapter 2, *Working with Creational Patterns*.

We could do something like that here:

```
class OurHero {
    private var direction = Direction.LEFT
    private var x: Int = 42
    private var y: Int = 173
    fun shoot(): Projectile {
        return Projectile(x, y, direction)
    }
}
```

But Michael wants our hero to have at least three different weapons:

- **Peashooter**: Shoots small peas that fly straight. Our hero starts with it.
- **Pomegranate**: Explodes when hitting an enemy, much like a grenade.
- **Banana**: Returns like a boomerang when it reaches the end of the screen.

Come on, Michael, give us some slack! Can't you just stick with regular guns that all work the same?

Fruit arsenal

First, let's discuss how we could solve this in the Java way.

In Java, we would have created an interface that abstracts these changes. In our case, what changes is our hero's weapon:

```
interface Weapon {

    fun shoot(x: Int,

              y: Int,

              direction: Direction): Projectile

}
```

Then, all the other weapons would implement this interface. Since we don't deal with aspects such as rendering or animating objects, no specific behavior will be implemented here:

```
// Flies straight

class Peashooter : Weapon {

    override fun shoot(

        x: Int,

        y: Int,

        direction: Direction

    ) = Projectile(x, y, direction)

}

// Returns back after reaching end of the screen
```

```
class Banana : Weapon {
    override fun shoot(
        x: Int,
        y: Int,
        direction: Direction
    ) = Projectile(x, y, direction)
}
// Other similar implementations here
```

All of the weapons in our game will implement the same interface, overriding its single method.

Our hero will hold a reference to a weapon, `Peashooter`, at the beginning:

```
private var currentWeapon: Weapon = Peashooter()
```

This reference will delegate the actual shooting process to it:

```
fun shoot(): Projectile = currentWeapon.shoot(x, y,   direction)
```

What's left is the ability to equip another `weapon`:

```
fun equip(weapon: Weapon) {
    currentWeapon = weapon
}
```

And that's what the **Strategy** design pattern is all about. It makes our algorithms – in this case, the weapons in our game – interchangeable.

Citizen functions

With Kotlin, there's a more efficient way to implement the same functionality using fewer classes. That's thanks to the fact that functions in

Kotlin are first-class citizens. *But what does that mean?*

For one, we can assign functions to the variables of our class, just like any other standard value. It makes sense that you can assign a primitive value to your variable:

```
val x = 7
```

You could also assign an object to a variable, as we have done many times already:

```
var myPet = Canary("Michael")
```

So, why shouldn't you be able to assign a function to your variable?

In Kotlin, you can easily do that. Here's an example:

```
val square = fun(x: Int): Long {
    return (x * x).toLong()
}
```

Let's see how that may help us simplify our design.

First, we'll define a namespace for all our weapons. We can use an object for that. This is not mandatory but it helps keep everything in check. Then, instead of classes, each of our weapons will become a function:

```
object Weapons {
    // Flies straight
    fun peashooter(x: Int, y: Int, direction: Direction):
        Projectile {
        return Projectile(x, y, direction)
    }
    // Returns back after reaching end of the screen
    fun banana(x: Int, y: Int, direction: Direction):
        Projectile {
        return Projectile(x, y, direction)
    }
    // Other similar implementations here
}
```

As you can see, instead of implementing an interface, we have multiple functions receiving the same parameters and returning the same object.

The most interesting part is our hero. The **`OurHero`** class now contains two values, both of which are functions:

```
class OurHero {
    var currentWeapon = Weapons::peashooter
    val shoot = fun() {
        currentWeapon(x, y, direction)
    }
}
```

The interchangeable part is **`currentWeapon`**, while **`shoot`** is now an anonymous function that wraps it.

To test that our idea works, we can shoot the default weapon once, then switch to another weapon and shoot with it again:

```
val hero = OurHero()
hero.shoot()
hero.currentWeapon = Weapons::banana
hero.shoot()
```

Notice that this dramatically reduces the number of classes we have to write while keeping the same functionality. If your interchangeable algorithm doesn't have a state, you can replace it with a simple function. Otherwise, introduce an interface, and let each Strategy pattern implement it.

That's also the first time we used the function reference operator, `::`. This operator allows us to refer to a function as if it was a variable instead of invoking it.

Strategy is a valuable pattern whenever your application needs to change its behavior at runtime. One example is a booking system for flights that allows for overbooking; that is, placing more passengers on a flight than there are seats. You may decide that you wish to enable overbooking up until one day before the flight and then disallow it. You can do this by switching strategies instead of adding complex checks to your code.

Now, let's look at another pattern that should help us work with complex data structures.

Iterator

When we were discussing the **Composite** design pattern in the previous chapter, we noted that the design pattern felt a bit incomplete. Now is the time to reunite the twins separated at birth. Much like Arnold Schwarzenegger and Danny DeVito, they're very different but complement each other well.

As you may remember from the previous chapter, a squad consists of troopers or other squads. Let's create one now:

```
val platoon = Squad(
    Trooper(),
    Squad(
        Trooper(),
    ),
    Trooper(),
    Squad(
        Trooper(),
        Trooper(),
    ),
    Trooper()
)
```

Here, we created a platoon that consists of four troopers in total.

It would be useful if we could print all the troopers in this platoon using a **`for-each`** loop, which we learned about back in Chapter 1, *Getting Started with Kotlin*.

Let's just try to write that code and see what happens:

```
for (trooper in platoon) {
    println(trooper)
}
```

Although this code doesn't compile, the Kotlin compiler provides us with a useful hint:

```
>For loop range must have an iterator method
```

Before we follow the compiler's guidance and implement the method, let's briefly discuss what problem we have at the moment.

Our platoon, which implements a Composite design pattern, is not a flat data structure. It can contain objects that contain other objects – squads can contain troopers as well as other squads. In this case, however, we want to abstract that complexity and work with it as if it was just a list of troopers. The Iterator pattern does just that – it *flattens* our complex data structure into a simple sequence of elements. The order of the elements and what elements to ignore is for the iterator to decide.

To use our **`Squad`** object in a **`for-each`** loop, we will need to implement a special function called **`iterator()`**. And since it's a special function, we'll need to use the **`operator`** keyword:

```
operator fun iterator() = ...
```

What our function returns is an anonymous object that implements the **`Iterator<T>`** interface:

```
operator fun iterator() = object: Iterator<Trooper> {

    override fun hasNext(): Boolean {

        // Are there more objects to iterate over?

    }

    override fun next(): Trooper {

        // Return next Trooper

    }

}
```

Once again, we can see the use of generics in Kotlin. **`Iterator<Trooper>`** means that the objects that our **`next()`** method returns will always be of the **`Trooper`** type.

To be able to iterate all the elements, we need to implement two methods – one to fetch the next element and one to let the loop know when to stop. Let's do that by executing the following steps:

1. First, we need a state for our iterator. It will remember that the last element is returned:

   ```
   operator fun iterator() = object: Iterator<Trooper> {

       private var i = 0

       // More code here

   }
   ```

2. Next, we need to tell it when to stop. In simple cases, this would be equal to the size of the underlying data structure:

   ```
   override fun hasNext(): Boolean {

       return i < units.size

   }
   ```

   This will be a bit more complex since we need to handle some edge cases. You can find the complete implementation in this book's GitHub repository.

3. Finally, we need to know which unit to return. For simple cases, we could just return the current element and increase the element count by one:

   ```
   override fun next() = units[i++]
   ```

   In our case, this is a bit more complex since squads could contain other squads. Again, you can find the full implementation in this book's GitHub repository.

Sometimes, it also makes sense to receive an iterator as a parameter of a function:

```
fun <T> printAnything(iter: Iterator<T>) {

    while (iter.hasNext()) {

        println(iter.next())

    }

}
```

This function will iterate over anything that supplies an iterator. This is also an example of a generic function in Kotlin. Note **`<T>`**, which comes before the function's name.

As a regular developer that doesn't invent new data structures for a living, you may not implement iterators often. However, it's still important to know how they work behind the scenes.

The following section will show how to design finite-state machines efficiently.

State

You can think of the **State** design pattern as an opinionated Strategy pattern, which we discussed at the beginning of this chapter. But while the Strategy pattern is usually replaced from the outside by the client, the state may change internally based solely on the input it gets.

Look at this dialog a client wrote with the Strategy pattern:

- **Client**: *Here's a new thing to do, start doing it from now on.*

- **Strategy**: *OK, no problem.*
- **Client**: *What I like about you is that you never argue with me.*

Compare it with this one:

- **Client**: *Here's some new input I got from you.*
- **State**: *Oh, I don't know. Maybe I'll start doing something differently. Maybe not.*

The client should also expect that the state may even reject some of its inputs:

- **Client**: *Here's something for you to ponder, State.*
- **State**: *I don't know what it is! Don't you see I'm busy? Go bother some Strategy with this!*

So, why do clients still tolerate that state of ours? Well, the state is good at keeping everything under control.

Fifty shades of State

The carnivorous snails from our platformer game have had enough of this abuse. So, the player throws peas and bananas at them, only to get to another sorry castle. *Now, they shall act!*

Let's see how the State design pattern can help us model a changing behavior of an actor – in our case, of the enemies in our platformer game. By default, the snail should stand still to conserve snail energy. But when the hero gets close, it should dash toward them aggressively.

If the hero manages to injure it, it should retreat to lick its wounds. Then, it will repeat attacking until either of them is dead.

First, we'll declare what can happen during a snail's life:

```
interface WhatCanHappen {

    fun seeHero()

    fun getHit(pointsOfDamage: Int)

    fun calmAgain()

}
```

Our snail implements this interface so that it is notified of anything that may happen to it and act accordingly:

```
class Snail : WhatCanHappen {

    private var healthPoints = 10

    override fun seeHero() {

    }

    override fun getHit(pointsOfDamage: Int) {

    }

    override fun calmAgain() {

    }

}
```

Now, we can declare the **`Mood`** class, which we will mark with the **`sealed`** keyword:

```
sealed class Mood {

   // Some abstract methods here, like draw(), for example

}
```

Sealed classes are abstract and cannot be instantiated. We'll see the benefit of using them in a moment. But before that, let's declare other states:

```
object Still : Mood()

object Aggressive : Mood()

object Retreating : Mood()

object Dead : Mood()
```

These are all the different states – sorry, moods – of our snail.

In terms of the State design pattern, **`Snail`** is the context. It holds the state. So, we declare a member for it:

```
class Snail : WhatCanHappen {

    private var mood: Mood = Still

    // As before

}
```

Now, let's define what **`Snail`** should do when it sees our hero:

```
override fun seeHero() {

    mood = when(mood) {

        is Still -> Aggressive

    }

}
```

Notice that this doesn't compile. This is where the **`sealed`** class comes into play. Much like with an **`enum`**, Kotlin knows that there's a finite number of classes that extend from it. So, it requires that our **`when`** is exhaustive and specifies all the different cases in it.

IMPORTANT NOTE:

If you're using IntelliJ as your IDE, it will even suggest that you ***`Add remaining branches`*** *automatically.*

We can use **`else`** to describe no state change:

```
override fun seeHero() {
    mood = when(mood) {
        is Still -> Aggressive
        else -> mood
    }
}
```

When the snail gets hit, we need to decide whether it's dead or not. For that, we can use **`when`** without an argument:

```
override fun getHit(pointsOfDamage: Int) {
    healthPoints -= pointsOfDamage
    mood = when {
        (healthPoints <= 0) -> Dead
        mood is Aggressive -> Retreating
        else -> mood
    }
}
```

Note that we use the **`is`** keyword here, which is the same as **`instanceof`** in Java, but more concise.

State of the nation

The previous approach contains most of the logic for our context. You may sometimes see a different approach, which is valid as your context becomes bigger.

In this approach, **`Snail`** would become thin:

```
class Snail {

    internal var mood: Mood = Still(this)

    private var healthPoints = 10

    // That's all!

}
```

Note that we marked **`mood`** as **`internal`**. This lets other classes in the same package alter it. Instead of **`Snail`** implementing **`WhatCanHappen`**, our **`Mood`** will implement it instead:

```
sealed class Mood : WhatCanHappen
```

Now, the logic resides within our state objects:

```
class Still(private val snail: Snail) : Mood() {

    override fun seeHero() {

        snail.mood = Aggressive

    }


    override fun getHit(pointsOfDamage: Int) {

        // Same logic from before

    }


    override fun calmAgain() {

       // Return to Still state
    }
}
```

Note that our state objects now receive a reference to their context in the constructor.

Use the first approach if the amount of code in your state is relatively small. Use the second approach for cases if the variants differ a lot. One example from the real world, where this pattern is widely used, is Kotlin's **Coroutines** mechanism. We'll discuss this in detail in *Chapter 5*, *Introducing Functional Programming*.

Now, let's look at another pattern that encapsulates actions.

Command

This design pattern allows you to encapsulate actions inside an object to be executed sometime later. Furthermore, if we can execute one action later, we could also execute many, or even schedule exactly when to execute them.

Let's go back to our `Stormtrooper` management system from *Chapter 3*, *Understanding Structural Patterns*. Here's an example of implementing the `attack` and `move` functions from before:

```
class Stormtrooper(...) {
    fun attack(x: Long, y: Long) {
        println("Attacking ($x, $y)")
        // Actual code here
    }
    fun move(x: Long, y: Long) {
        println("Moving to ($x, $y)")
        // Actual code here
    }
}
```

We could even use the **Bridge** design pattern from the previous chapter to provide the actual implementations.

The problem we need to solve now is that our trooper can remember exactly one command. That's it. If they start at `(0, 0)`, which is the top of the screen, we can tell them to `move(20, 0)`, which is 20 steps to the right, and then to `move(20, 20)`. In this case, they'll move straight to `(20, 20)` and will probably get destroyed because there are rebels that we must try to avoid at all costs:

```
[storm trooper](0, 0) -> good direction  -> (20, 0)
          [rebel] [rebel]                      ⇓
        [rebel] [rebel] [rebel]                ⇓
           [rebel] [rebel]
            (5, 20)                        (20, 20)
```

If you've been following this book from the start or at least joined at Chapter 3, *Understanding Structural Patterns*, you probably have an idea of what we need to do, since we have already discussed the concept of *functions as first-class citizens* in the language.

Let's sketch a draft for this. We know that we want to hold a list of objects, but we don't know what type they should be yet. So, we'll use `Any` for now:

```
class Trooper {
    private val orders = mutableListOf<Any>()
    fun addOrder(order: Any) {
        this.orders.add(order)
    }
    // More code here
}
```

Then, we want to iterate over the list and execute the orders we have:

```
class Trooper {
    ...
    // This will be triggered from the outside once in a while
    fun executeOrders() {
        while (orders.isNotEmpty()) {
            val order = orders.removeFirst()
            order.execute() // Compile error for now
        }
    }
    ...
}
```

Note that Kotlin provides us with the **`isNotEmpty()`** function on collections, as an alternative to the **`!orders.isEmpty()`** check, as well as a **`removeFirst()`** function, which allows us to use our collection as if it was a queue.

Even if you're not familiar with the Command design pattern, you can guess that if we want our code to compile, we can define an interface with a single method, **`execute()`**:

```
interface Command {

    fun execute()

}
```

Then, we can hold a list at the same time in a member property:

```
private val commands = mutableListOf<Command>()
```

Each type of order, be it a move order or an attack order, would implement this interface as needed. That's basically what the Java implementation of this pattern would suggest in most cases. *But isn't there a better way?*

Let's look at **`Command`** again. The **`execute()`** method receives nothing, returns nothing, and does something. It's the same as writing the following code:

```
fun command(): Unit {

    // Some code here

}
```

It's no different from what we've seen previously. We could simplify this further:

```
() -> Unit
```

And instead of having an interface for this called **`Command`**, we'll have **`typealias`**:

```
typealias Command = ()-> Unit
```

This makes our **`Command`** interface redundant and allows us to remove it.

Now, this line stops compiling again:

```
command.execute() // Unresolved reference: execute
```

This is because **`execute()`** is just some name we invented. In Kotlin, functions use **`invoke()`**:

```
command.invoke() // Compiles
```

We can also omit **`invoke()`**, which will leaves us with the following code:

```
fun executeOrders() {

    while (orders.isNotEmpty()) {

        val order = orders.removeFirst()

        order() // Executed the next order

    }

}
```

That's nice, but currently, our function has no parameters at all. *What happens if our function receives arguments?*

One option would be to change the signature of our **`Command`** so that we receive two parameters:

```
(x: Int, y: Int)-> Unit
```

But what if some commands receive no arguments, or only one, or more than two? We also need to remember what to pass to **`invoke()`** at each step.

A much better way is to have a **function generator**. This is a function that returns another function. If you have ever worked with the JavaScript language, then you'll know that it's a common practice to use closures to limit the scope and remember stuff. We'll do the same here:

```
val moveGenerator = fun(trooper: Trooper,

                        x: Int,

                        y: Int): Command {
```

```
    return fun() {
        trooper.move(x, y)
    }
}
```

When called with proper arguments, **`moveGenerator`** will return a new function. This function can be invoked whenever we find it suitable and it will remember three things:

- What method to call
- Which arguments to use
- Which object to use it on

Now, our **`Trooper`** may have a method like this:

```
fun appendMove(x: Int, y: Int) = apply {
    commands.add(moveGenerator(this, x, y))
}
```

This provides us with a nice fluent syntax:

```
val trooper = Trooper()
trooper.appendMove(20, 0)
    .appendMove(20, 20)
    .appendMove(5, 20)
    .execute()
```

Fluent syntax means that we can chain methods on the same object easily without the need to repeat its name many times.

This code will print the following output:

```
> Moving to (20, 0)

> Moving to (20, 20)

> Moving to (5, 20)
```

Now, we may issue any number of commands to our **`Trooper`** without needing to know how they are executed internally.

A function that receives or returns another function is called a **higher-order function**. We'll explore such functions many more times in this book.

Undoing commands

While not directly related, one of the advantages of the Command design pattern is the ability to undo commands. *What if we wanted to support such a functionality?*

Undoing is usually very tricky because it involves one of the following:

- Returning to the previous state (this is impossible if there's more than one client as this requires a lot of memory)
- Computing deltas (tricky to implement)
- Defining opposite operations (not always possible)

In our case, the opposite of the *move from (0,0) to (0, 20)* command would be *move from wherever you're now to (0,0)*. This can be achieved by storing a pair of commands:

```
private val commands =   mutableListOf<Pair<Command, Command>>()
```

We'll need to change our **`appendMove`** function so that it also stores the reverse command every time:

```
fun appendMove(x: Int, y: Int) = apply {
    val oppositeMove = /* If it's the first command,       generate
move to current location. Otherwise, get the       previous command
*/
    commands.add(moveGenerator(this, x, y) to oppositeMove)
}
```

Computing the opposite move is quite complex as we don't save the position of our soldier currently (it was something we should have implemented anyway). We'll also have to deal with some edge cases. But this should provide you with an idea of how such behavior can be achieved.

The **Command** design pattern is yet another example of functionality that is already embedded inside the language. In this case, this functions as a first-class citizen, which reduces the need to implement design patterns yourself. In the real world, this pattern is practical whenever you want to enqueue multiple actions or schedule an action to be executed later.

Chain of Responsibility

I'm a horrible software architect, and I don't particularly appreciate speaking with people. Hence, while sitting in *The Ivory Tower* (*that's the name of the cafe I often visit*), I wrote a small web application. If a developer has a question, they shouldn't approach me directly, oh no! They'll need to send me a proper request through this system and I shall only answer them if I deem their request worthy.

A **filter chain** is a ubiquitous concept in web servers. Usually, when a request reaches you, it's expected that the following is true:

- Its parameters have already been validated.
- The user has already been authenticated, if possible.
- User roles and permissions are known and the user is authorized to perform an action.

So, the code I initially wrote looked something like this:

```
data class Request(val email: String, val question: String)

fun handleRequest(r: Request) {

    // Validate

    if (r.email.isEmpty() || r.question.isEmpty()) {

        return

    }

    // Authenticate

    // Make sure that you know whos is this user

    if (r.isKnownEmail()) {

        return

    }

    // Authorize

    // Requests from juniors are automatically ignored by
       architects

    if (r.isFromJuniorDeveloper()) {

        return

    }

    println("I don't know. Did you check StackOverflow?")

}
```

It's a bit messy, but it works.

Then, I noticed that some developers decided that they can send me two questions at once. We have to add some more logic to this function. But wait – I'm an architect, after all. *So, isn't there a better way to delegate this?*

The goal of the **Chain of Responsibility** design pattern is to break a complex piece of logic into a collection of smaller steps, where each step, or link in the chain, decides whether to proceed to the next one or to return a result.

This time, we won't learn new Kotlin tricks but use those that we already know about. So, for example, we could start by implementing an interface such as this one:

```
interface Handler {

    fun handle(request: Request): Response

}
```

We never discussed what my response to one of the developers looked like. That's because I keep my chain of responsibility so long and complex that usually, they tend to solve the problems by themselves. I've never had to answer one of them, quite frankly. But let's assume the response looks something like this:

```
data class Response(val answer: String)
```

We could do this *the Java way* and start implementing each piece of logic inside its own handler:

```
class BasicValidationHandler(private val next: Handler) :   Handler
{

    override fun handle(request: Request): Response {
        if (request.email.isEmpty() ||
          request.question.isEmpty()) {
            throw IllegalArgumentException()
        }
        return next.handle(request)
    }
}
```

As you can see, here, we are implementing an interface with a single method, which we override with our desired behavior.

Other filters would look very similar to this one. We can compose them in any order we want:

```
val req = Request("developer@company.com",          "Who broke my
build?")
val chain = BasicValidationHandler(
    KnownEmailHandler(
        JuniorDeveloperFilterHandler(
            AnswerHandler()
        )
    )
)
val res = chain.handle(req)
```

But I won't even ask you the rhetorical question this time about better ways to do things. Of course, there's a better way. We're in the Kotlin world now. And we've seen how to use various functions in the previous section. So, let's define a function for this task:

```
typealias Handler = (request: Request) -> Response
```

We don't have a separate class and interface for something that simply receives a request and returns a response. Here's an example of how we can implement authentication in our application by using a simple function as a value:

```
val authentication = fun(next: Handler) =

    fun(request: Request): Response {

        if (!request.isKnownEmail()) {

            throw IllegalArgumentException()

        }

        return next(request)

    }
```

Here, **`authentication`** is a function that receives a function and returns a function. This pattern allows us to easily compose those functions:

```
val req = Request("developer@company.com",     "Why do we need
Software Architects?")

val chain = basicValidation(authentication

  (finalResponse()))

val res = chain(req)

println(res)
```

Which method you choose to use is up to you. For example, using interfaces is more explicit and would suit you better if you're creating a library or framework that others may want to extend.

Using functions is more concise and if you just want to split your code in a more manageable way, it may be the better choice.

You've probably seen this approach many times in the real world. For example, many web server frameworks use it to handle cross-cutting concerns, such as authentication, authorization, logging, and even routing requests. Sometimes, these are called **filters** or **middleware**, but it's the same Chain of Responsibility design pattern in the end. We'll discuss it again in more detail in *Chapter 10*, *Concurrent Microservices with Ktor*, and *Chapter 11*, *Reactive Microservices with Vert.x*, where we'll see how it's implemented by some of the most popular Kotlin frameworks.

The next design pattern will be a bit different from all the others and also somewhat more complex.

Interpreter

This design pattern may seem very simple or very hard, based on how much background you have in computer science. Some books that discuss classical software design patterns even decide to omit it altogether or put it somewhere at the end, for curious readers only.

The reason behind this is that the **Interpreter** design pattern deals with translating specific languages. *But why would we need that? Don't we have compilers to do that anyway?*

We need to go deeper

All developers have to speak many languages or sub-languages. Even as regular developers, we use more than one language. Think of tools that build your projects, such as Maven or Gradle. You can consider their configuration files and build scripts as languages with specific grammar. If you put elements out of order, your project won't be built correctly. This is because such projects have interpreters to analyze configuration files and act upon them.

Other examples are query languages, whether one of the SQL variations or one of the languages specific to NoSQL databases. If you're an Android developer, you may think of XML layouts as such languages too. Even HTML could be considered as a language that defines user interfaces. And there are others, of course.

Maybe you've worked with one of the testing frameworks that defines a custom language for testing, such as **Cucumber** (github.com/cucumber).

Each of these examples can be called a **domain-specific language** (**DSL**). A DSL is a language inside a language, built for a particular domain. We'll discuss how they work in the next section.

A language of your own

In this section, we'll define a simple *DSL-for-SQL* language. We won't define the format or grammar for it; instead, we'll provide an example of what it should look like:

```
val sql = select("name, age") {

    from("users") {

        where("age > 25")

    } // Closes from

} // Closes select
```

```
println(sql)
```

The goal of our language is to improve readability and prevent some common SQL mistakes, such as typos (such as using *FORM* instead of **`FROM`**). We'll cover the compile-time validations and autocompletion along the way.

The preceding code prints the following output:

```
> SELECT name, age FROM users WHERE age > 25
```

We'll start with the easiest part – implementing the **`select`** function:

```
fun select(columns: String, from: SelectClause.()->Unit):
    SelectClause {
    return SelectClause(columns).apply(from)
}
```

We could write this using single expression notation, but we are using the more verbose version for clarity here. This is a function that has two parameters. The first is a **`String`**, which is simple. The second is another function that receives nothing and returns nothing.

The most interesting part is that we specify the receiver for our lambda:

```
SelectClause.()->Unit
```

This is a very smart trick, so be sure to follow along. Remember extension functions, which we discussed in Chapter 1, *Getting Started with Kotlin*, and expanded on in Chapter 2, *Working with Creational Patterns*. The preceding code can be translated into the following code:

```
(SelectClause)->Unit
```

Here, you can see that although it may seem like this lambda receives nothing, it receives one argument: an object of the **`SelectClause`** type. The second trick lies in the usage of the **`apply()`** function, which we've already seen.

Let's look at this line:

```
SelectClause(columns).apply(from)
```

This can be translated into the following piece of code:

```
val selectClause = SelectClause(columns)

from(selectClause)

return selectClause
```

Here are the steps the preceding code will perform:

1. Initialize **`SelectClause`**, which is a simple object that receives one argument in its constructor.
2. Call the **`from()`** function with an instance of **`SelectClause`** as its only argument.
3. Return an instance of **`SelectClause`**.

This code only makes sense if **`from()`** does something useful with **`SelectClause`**.

Let's look at our DSL example again:

```
select("name, age", {

    this@select.from("users", {

        where("age > 25")

    })
})
```

We've made the receiver explicit now, meaning that the **`from()`** function will call the **`from()`** method on the **`SelectClause`** object.

You can start guessing what this method looks like. It receives **`String`** as its first argument and another lambda as its second:

```
class SelectClause(private val columns: String) {

    private lateinit var from: FromClause

    fun from(

        table: String,

        where: FromClause.() -> Unit

    ): FromClause {

        this.from = FromClause(table)

        return this.from.apply(where)

    }

    override fun toString() = "SELECT $columns $from"

}
```

This example could be shortened, but then we'd need to use **`apply()`** within **`apply()`**, which may seem confusing at this point.

This is the first time we've seen the **`lateinit`** keyword. Remember that the Kotlin compiler is very serious about null safety. If we omit **`lateinit`**, it will require us to initialize the variable with a default value. But since we'll only know this at a later time, we are asking the compiler to relax a bit.

IMPORTANT NOTE:

*Note that if we don't make good on our promises and forget to initialize the variable, we'll get **`UninitializedPropertyAccessException`** when we access it for the first time.*

This keyword is quite dangerous, so use it with caution.

Let's go back to our preceding code; all we do is the following:

1. Create an instance of **`FromClause`**.
2. Store **`FromClause`** as a member of **`SelectClause`**.
3. Pass an instance of **`FromClause`** to the **`where`** lambda.
4. Return an instance of **`FromClause`**.

Hopefully, you're starting to get the gist of it:

```
select("name, age", {
    this@select.from("users", {
        this@from.where("age > 25")
    })
})
```

What does this mean? After understanding the **`from()`** method, this should be much simpler. **`FromClause`** must have a method called **`where()`** that receives one argument of the **`String`** type:

```
class FromClause(private val table: String) {
    private lateinit var where: WhereClause
    fun where(conditions: String) = this.apply {
        where = WhereClause(conditions)
    }
    override fun toString() = "FROM $table $where"
}
```

Note that we have made good on our promise and shortened the method this time.

We initialized an instance of **`WhereClause`** with the string we received and returned it – simple as that:

```
class WhereClause(private val conditions: String) {

    override fun toString() = "WHERE $conditions"

}
```

`WhereClause` only prints the word **`WHERE`** and the conditions it received:

```
class FromClause(private val table: String) {

    // More code here...

    override fun toString() = "FROM $table $where"

}
```

`FromClause` prints the word **`FROM`**, as well as the table name it received and everything **`WhereClause`** printed:

```
class SelectClause(private val columns: String) {

    // More code here...

    override fun toString() = "SELECT $columns $from"

}
```

`SelectClause` prints the word **`SELECT`**, the columns it got, and whatever **`FromClause`** printed.

Taking a break

Kotlin provides beautiful capabilities for creating readable and type-safe DSLs. But the Interpreter design pattern is one of the hardest in the toolbox. If you didn't get it from the get-go, take some time to debug the previous code. Understand what the **`this`** expression means at each step, as well as when we call the function of an object and when we call the method of an object.

Call suffix

We left out one last notion of Kotlin's DSL until the end of this section so that we didn't confuse you.

Let's look at our DSL again:

```
val sql = select("name, age") {
              from("users") {
                  where("age > 25")
              } // Closes from
          } // Closes select
```

Note that although the **`select`** function receives two arguments – a string and a lambda – the lambda is written outside of the round brackets, not inside them.

This is called **call suffix** and is a widespread practice in Kotlin. If our function receives another function as its last argument, we can pass it out of parentheses.

This results in a much clearer syntax, especially for DSLs such as this one.

The Interpreter design pattern and Kotlin's abilities to produce DSLs with type-safe builders are compelling. But as they say, *with great power comes great responsibility*. So, consider if your case is complex enough to construct a language within a language, or whether using the Kotlin basic syntax will be enough.

Now, let's go back to the game we were building to see how we can decouple object communication.

Mediator

The development team of our game has some real problems – and they're not related to code directly. As you may recall, our little indie company consists of only me, a canary named *Michael* that acts as a product manager, and two cat designers that sleep most of the day but produce some decent mockups from time to time. We have no **Quality Assurance** (**QA**) whatsoever. Maybe that's one of the reasons our game keeps crashing all the time.

Recently Michael has introduced me to a parrot named Kenny, who happens to be QA:

```
interface QA {
    fun doesMyCodeWork(): Boolean
}
interface Parrot {
    fun isEating(): Boolean
    fun isSleeping(): Boolean
}
object Kenny : QA, Parrot {
    // Implements interface methods based on parrot     // schedule
}
```

`Kenny` is a simple object that implements two interfaces: **`QA`**, to do QA work, and **`Parrot`**, because it's a parrot.

Parrot QAs are very motivated. They're ready to test the latest version of my game at any time. But they don't like to be bothered when they are either sleeping or eating:

```
object Me

object MyCompany {

    val cto = Me

    val qa = Kenny

    fun taskCompleted() {

        if (!qa.isEating() && !qa.isSleeping()) {

            println(qa.doesMyCodeWork())

        }

    }

}
```

In case Kenny has any questions, I gave him my direct number:

```
object Kenny : ... {

    val developer = Me

}
```

Kenny is a hard-working parrot. But we had so many bugs that we also had to hire a second parrot QA, Brad. If Kenny is free, I give the job to him as he's more acquainted with our project. But if he's busy, I check if Brad is free and give the task to him:

```
class MyCompany {

    ...
    val qa2 = Brad

    fun taskCompleted() {

        ...

        else if (!qa2.isEating() && !qa2.isSleeping()) {

            println(qa2.doesMyCodeWork())

        }

    }

}
```

Brad, being more junior, usually checks with Kenny first. And Kenny also gave my number to him:

```
object Brad : QA, Parrot {

    val senior = Kenny

    val developer = Me

    ...

}
```

Then, Brad introduces me to George. George is an owl, so he sleeps at different times than Kenny and Brad. This means that he can check my code at night.

George checks everything with Kenny and with me:

```
object George : QA, Owl {

    val developer = Me

    val mate = Kenny

    ...

}
```

The problem is that George is an avid football fan. So, before calling him, we need to check if he's watching a game:

```
class MyCompany {

    ...

    val qa3 = George

    fun taskCompleted() {

        ...

        else if (!qa3.isWatchingFootball()) {

            println(qa3.doesMyCodeWork())

        }

    }

}
```

Kenny, out of habit, checks in with George too, because George is a very knowledgeable owl:

```
object Kenny : QA, Parrot {

    val peer = George

    ...

}
```

Then, there's Sandra. She's a different kind of bird because she's not part of QA but a copywriter:

```
interface Copywriter {

    fun areAllTextsCorrect(): Boolean

}

interface Kiwi

object Sandra : Copywriter, Kiwi {
    override fun areAllTextsCorrect(): Boolean {
        return ...
    }
}
```

I try not to bother her unless there's a major release:

```
class MyMind {
    ...
    val translator = Sandra
    fun taskCompleted(isMajorRelease: Boolean) {
        ...
        if (isMajorRelease) {
            println(translator.areAllTranslationsCorrect())
        }
    }
}
```

I have a few problems here:

- First, my mind almost explodes trying to remember all those names. So might yours.
- Second, I need to remember how to interact with each person. I'm the one doing all the checks before calling them.
- Third, notice how George tries to confirm everything with Kenny, and Kenny with George. Luckily, up until now, George has always been watching a football game when Kenny calls him. And Kenny is asleep

when George needs to confirm something with him. Otherwise, they would get stuck on the phone for eternity.

- Fourth, and what bothers me the most, is that Kenny plans to leave soon to open his own startup, ParrotPi. Imagine all the code we'll have to change now!

All I want to do is check if everything is alright with my code. Someone else should do all this talking!

The middleman

The **Mediator** design pattern is simply a control freak. It doesn't like it when one object speaks to the other directly. It gets mad sometimes when that happens. No – everybody should only speak through him. *What's the explanation for this?* It reduces coupling between objects. Instead of knowing some other objects, everybody should know only them, the mediator.

I decided that Michael should manage all those processes and act as the mediator of them:

```
interface Manager {

    fun isAllGood(majorRelease: Boolean): Boolean

}
```

Only Michael will know all the other birds:

```
object Michael : Canary, ProductManager {

    private val kenny = Kenny(this)

    private val brad = Brad(this)
    override fun isAllGood(majorRelease: Boolean): Boolean {
        if (!kenny.isEating() && !kenny.isSleeping()) {
            println(kenny.doesMyCodeWork())
        } else if (!brad.isEating() && !brad.isSleeping()) {
            println(brad.doesMyCodeWork())
        }
        return true
    }
}
```

Notice how the mediator encapsulates the complex interactions between different objects, exposing a very simple interface.

I'll only remember Michael and he'll do the rest:

```
class MyCompany(private val manager: Manager) {
    fun taskCompleted(isMajorRelease: Boolean) {
        println(manager.isAllGood(isMajorRelease))
    }
}
```

I'll also change my phone number and make sure that everybody gets only Michael's:

```
class Brad(private val manager: Manager) : ... {
   // No reference to Me here
   ...
}
```

Now, if somebody needs somebody else's opinion, they'll need to go through Michael first:

```
class Kenny(private val manager: Manager) : ... {
   // No reference to George, or anyone else
   ...
}
```

As you can see, there's nothing new we can learn about Kotlin through this pattern.

Mediator flavors

There are two *flavors* to the Mediator pattern. We'll call them *strict* and *loose*. We saw the strict version previously. We tell the mediator exactly what to do and expect an answer from it.

The loose version will expect us to notify the mediator of what happened, but not to expect an immediate answer. Instead, if they need to notify us in return, they should call us.

Mediator caveats

Michael suddenly becomes ever so important. Everybody knows only him and only he can manage their interactions. He may even become a *God Object*, all-knowing and almighty, which is an antipattern from Chapter 9, *Idioms and Anti-Patterns*. Even if he's that important, be sure to define what this mediator should, and – even more importantly – shouldn't do.

Let's continue with our example and discuss yet another behavioral pattern.

Memento

Since Michael became a manager, it's been tough to catch him if I have a question. And when I do ask him something, he just throws something and runs to the next meeting.

Yesterday, I asked him what new weapon we should introduce in our game. He told me it should be a *Coconut Cannon*, clear as day. But today, when I presented him with this feature, he chirped at me angrily! Finally, he said he told me to implement a *Pineapple Launcher* instead. I'm lucky he's just a canary.

It would be great if I could record him so that when we have another meeting that goes awry because he's not paying full attention, I can simply replay everything he said.

Let's sum up my problems first – Michael's thoughts are his and his only:

```
class Manager {

    private var thoughts = mutableListOf<String>()

    ...

}
```

The problem is that since Michael is a canary, he can only hold `2` thoughts in his mind:

```
class Manager {

    ...

    fun think(thought: String) {

        thoughts.add(thought)

        if (thoughts.size > 2) {
            thoughts.removeFirst()
        }
    }
}
```

If Michael thinks about more than `2` things at a time, he'll forget the first thing he thought about:

```
michael.think("Need to implement Coconut Cannon")
michael.think("Should get some coffee")
michael.think("Or maybe tea?") // Forgot about Coconut   Cannon
michael.think("No, actually, let's implement Pineapple   Launcher")
// Forgot that he wanted coffee
```

Even in the recording, what he says is quite hard to understand (because he doesn't return anything).

And even if I do record him, Michael can claim it's what he said, not what he meant.

The Memento design pattern solves this problem by saving the internal state of an object, which can't be altered from the outside (so that Michael cannot deny that he said it) and can only be used by the object itself.

In Kotlin, we can use an **`inner`** class to implement this:

```
class Manager {
    ...
    inner class Memory(private val mindState: List<String>) {
        fun restore() {
            thoughts = mindState.toMutableList()
        }
    }
}
```

Here, we can see a new keyword, **`inner`**, for marking our class. If we omit this keyword, the class is called **`Nested`** and is similar to the static nested class from Java. Inner classes have access to the private fields of the outer class. For that reason, our **`Memory`** class can change the internal state of the **`Manager`** class easily.

Now, we can record what Michael says at this moment by creating an imprint of the current state:

```
fun saveThatThought(): Memory {
    return Memory(thoughts.toList())
}
```

At this point, we can capture his thoughts in an object:

```
val michael = Manager()
michael.think("Need to implement Coconut Cannon")
michael.think("Should get some coffee")
val memento = michael.saveThatThought()
michael.think("Or maybe tea?")
michael.think("No, actually, let's implement Pineapple   Launcher")
```

Now, we need to add a means of going back to a previous line of thought:

```
class Manager {
    ...
    fun `what was I thinking back then?`(memory: Memory) {
        memory.restore()
    }
}
```

Here, we can see that if we want to use special characters in function names, such as spaces, we can, but only if a function name is wrapped in *backticks*. Usually, that's not the best idea, but it has its uses, as we'll cover in *Chapter 10*, *Concurrent Microservices with Ktor*.

What's left is using **`memento`** to go back in time:

```
with(michael) {
    think("Or maybe tea?")
    think("No, actually, let's implement Pineapple
        Launcher")
}
michael.`what was I thinking back then?`(memento)
```

The last invocation will return Michael's mind to thinking about Coconut Cannon, of all things.

Note how we use the **`with`** standard function to avoid repeating **`michael.think()`** on each line. This function is helpful if you need to refer to the same object often in the same block of code and would like to avoid repetition.

I don't expect you to see the Memento design pattern implemented very often in the real world. But it still may be useful in some types of applications that need to recover to some previous state.

At the beginning of this chapter, we discussed the Iterator design pattern, which helps us work with complex data structures. Next, we'll look at another design pattern with a somewhat similar goal.

Visitor

This design pattern is usually a close friend of the Composite design pattern, which we discussed in Chapter 3, *Understanding Structural Patterns*. It can either extract data from a complex tree-like structure or add behavior to each node of the tree, much like the Decorator design pattern does for a single object.

My plan, being a lazy software architect, worked out quite well. My request-answering system from the chain of responsibility worked quite well and I don't have plenty of time for coffee. But I'm afraid some developers begin to suspect that I'm a bit of a fraud.

To confuse them, I plan to produce weekly emails with links to all the latest buzzword articles. Of course, I don't plan to read them myself – I just want to collect them from some popular technology sites.

Writing a crawler

Let's look at the following data structure, which is very similar to what we had when we discussed the Iterator design pattern:

```
Page(Container(Image(),
               Link(),
               Image()),
     Table(),
     Link(),
     Container(Table(),
               Link()),
     Container(Image(),

               Container(Image(),

                         Link())))
```

Page is a container for other HTML elements, but not **HtmlElement** by itself. **Container** holds other containers, tables, links, and images. **Image** holds its link in the **src** attribute. **Link** has the **href** attribute instead.

What we would like to do is extract all the URLs from the object.

We will start by creating a function that will receive the root of our object tree – a **Page** container, in this case – and return a list of all the available links:

```
fun collectLinks(page: Page): List<String> {

    // No need for intermediate variable there

    return LinksCrawler().run {

        page.accept(this)

        this.links

    }

}
```

Using **run** allows us to control what we return from the block's body. In this case, we will return the **links** objects we've gathered. Inside the **run** block, this refers to the object it operates on – in our case, **LinksCrawler**.

In Java, the suggested way to implement the Visitor design pattern is to add a method for each class that will accept our new functionality. We'll do the same, but not for all the classes. Instead, we'll only define this method for container elements:

```
private fun Container.accept(feature: LinksCrawler) {
    feature.visit(this)

}

// Or using a shorter syntax:

private fun Page.accept(feature: LinksCrawler) =
  feature.visit(this)
```

Our feature will need to hold a collection internally and expose it for read purposes. In Java, we will only specify the getter for this member; no setter is required. In Kotlin, we can specify the value without a backing field:

```
class LinksCrawler {

    private var _links = mutableListOf<String>()

    val links

        get()= _links.toList()

    ...

}
```

We want our data structure to be immutable. That's the reason we're calling **`toList()`** on it.

IMPORTANT NOTE:

The functions that iterate over branches could be simplified even further if we use the Iterator design pattern.

For containers, we simply pass their elements further:

```
class LinksCrawler {

    ...

    fun visit(page: Page) {

        visit(page.elements)

    }
    fun visit(container: Container) =
        visit(container.elements)

    ...

}
```

Specifying the parent class as **`sealed`** helps the compiler further. We discussed sealed classes earlier in this chapter while covering the State design pattern. Here is the code:

```
sealed class HtmlElement

class Container(...) : HtmlElement(){

    ...

}

class Image(...) : HtmlElement() {

    ...

}

class Link(...) : HtmlElement() {

    ...

}

class Table : HtmlElement()
```

The most interesting logic is in the leaves of our tree-like structure:

```
class LinksCrawler {

    ...

    private fun visit(elements: List<HtmlElement>) {

        for (e in elements) {

            when (e) {

                is Container -> e.accept(this)
                is Link -> _links.add(e.href)
                is Image -> _links.add(e.src)
                else -> {}
            }
        }
    }
}
```

Note that in some cases, we don't want to do anything. This is specified by an empty block in our `else` clause, `else -> {}`. This is yet another example of **smart casts** in Kotlin.

Notice that after we checked that the element is a `Link`, we gained type-safe access to its `href` attribute. That's because the compiler is doing the casts for us. The same is true for the `Image` element.

Although we achieved our goals, the usability of this pattern can be debated. As you can see, it's one of the more verbose elements we have and introduces tight coupling between classes that are receiving additional behavior and the Visitor pattern itself.

Template method

Some lazy people make art out of their laziness. Take me, for example. Here's my daily schedule:

1. 8:00 A.M. – 9:00 A.M.: Arrive at the office
2. 9:00 A.M. – 10:00 A.M.: Drink coffee
3. 10:00 A.M. –1 2:00 P.M.: Attend some meetings or review code

4. 12:00 P.M. – 1:00 P.M.: Go out for lunch
5. 1:00 P.M. – 4:00 P.M.: Attend some meetings or review code
6. 4:00 P.M.: Sneak back home

Some parts of my schedule never change, while some do. Specifically, I have two slots in my calendar that any number of meetings could occupy.

At first, I thought I could decorate my changing schedule with that setup and teardown logic, which happens before and after. But then there's lunch, which is holy for architects and happens in between.

Java is pretty clear on what you should do. First, you create an abstract class. Then, you mark all the methods that you want to implement by yourself as **`private`**:

```
abstract class DayRoutine {
    private fun arriveToWork() {
        println("Hi boss! I appear in the office
            sometimes!")
    }
    private fun drinkCoffee() {
        println("Coffee is delicious today")
    }
    ...
    private fun goToLunch() {
        println("Hamburger and chips, please!")
    }
    ...
    private fun goHome() {

        // Very important no one notices me, so I must keep
        // quiet!

        println()

    }

    ...

}
```

All the methods that are changing from day to day should be defined as **`abstract`**:

```
abstract class DayRoutine {

    ...

    abstract fun doBeforeLunch()

    ...

    abstract fun doAfterLunch()

    ...

}
```

If you want to be able to replace a function but also want to provide a default implementation, you should leave it **`public`**:

```
abstract class DayRoutine {

    ...

    open fun bossHook() {

        // Hope he doesn't hook me there

    }

    ...

}
```

Remember that **`public`** is the default visibility in Kotlin.

Finally, you have a method that executes your algorithm. It's **`final`** by default:

```
abstract class DayRoutine {

    ...

    fun runSchedule() {

        arriveToWork()

        drinkCoffee()

        doAfterLunch()

        goToLunch()

        doAfterLunch()

        goHome()

    }

}
```

Now, if we want to have a schedule for Monday, we can simply implement the missing parts:

```
class MondaySchedule : DayRoutine() {

    override fun doBeforeLunch() {

        println("Some pointless meeting")

        println("Code review. What this does?")

    }

    override fun doAfterLunch() {

        println("Meeting with Ralf")

        println("Telling jokes to other architects")

    }
    override fun bossHook() {

        println("Hey, can I have you for a sec in my
            office?")

    }

}
```

What does Kotlin add on top of that? What it usually does – conciseness. As we saw previously, this can be achieved through functions.

We have three *moving parts* – two mandatory activities (the software architect must do something before and after lunch) and one optional (the boss may stop him before he sneaks off home):

```
fun runSchedule(beforeLunch: () -> Unit,

               afterLunch: () -> Unit,

               bossHook: (() -> Unit)? = fun() { println() }) {

    ...

}
```

We'll have a function that accepts up to three other functions as its arguments. The first two are mandatory, while the third may not be supplied at all or assigned with **`null`** to explicitly state that we don't want that function to occur:

```
fun runSchedule(...) {

    ...

    arriveToWork()

    drinkCoffee()

    beforeLunch()

    goToLunch()
    afterLunch()

    bossHook?.let { it() }

    goHome()

}
```

Inside this function, we'll have our algorithm. The invocations of **`beforeLunch()`** and **`afterLunch()`** should be clear; after all, those are the functions that are passed to us as arguments. The third one, **`bossHook`**, may be **`null`**, so we only execute it if it's not. We can use the following construct for that:

```
?.let { it() }
```

But what about the other functions – the ones we want to always implement by ourselves?

Kotlin has a notion of **local functions**. These are functions that reside in other functions:

```
fun runSchedule(...) {

    fun arriveToWork(){

        println("How are you all?")

    }

    val drinkCoffee = { println("Did someone left the milk
        out?") }

    fun goToLunch() = println("I would like something
        italian")

    val goHome = fun () {

        println("Finally some rest")

    }
    arriveToWork()
    drinkCoffee()
    ...
    goToLunch()
    ...
    goHome()
}
```

These are all valid ways to declare a local function. No matter how you define them, they're invoked in the same way. Local functions can only be accessed by the parent function they were declared in and are a great way to extract common logic without the need to expose it.

With that, we're left with the code structure. Defining the algorithm's structure but letting others decide what to do at some points – that's what the Template method is all about.

We're almost at the end of this chapter. There is just one more design pattern to discuss, but it's one of the most important ones.

Observer

Probably one of the highlights of this chapter, this design pattern provides us with a bridge to the following chapters, which are dedicated to functional programming.

So, what is the **Observer** *pattern all about?* You have one *publisher*, which may also be called a *subject*, that may have many *subscribers*, also known as *observers*. Each time something interesting happens with the publisher, all of its subscribers should be updated.

This may look a lot like the **Mediator** design pattern, but there's a twist. Subscribers should be able to register or unregister themselves at runtime.

In the classical implementation, all subscribers/observers need to implement a particular interface for the publisher to update them. But since Kotlin has higher-order functions, we can omit this part. The publisher will still have to provide a means for observers to be able to subscribe and unsubscribe.

This may have sounded a bit complex, so let's take a look at the following example.

Animal choir example

So, some animals have decided to have a choir of their own. The cat was elected as the conductor of the choir (it doesn't like to sing anyway).

The problem is that these animals escaped from the Java world, so they don't have a common interface. Instead, each has a different way of making a sound:

```
class Bat {

    fun screech() {

        println("Eeeeeee")

    }

}

class Turkey {

    fun gobble() {

        println("Gob-gob")

    }
}
class Dog {
    fun bark() {
        println("Woof")
    }
    fun howl() {
        println("Auuuu")
    }
}
```

Luckily, the cat was elected not only because it was vocally challenged, but also because it was smart enough to follow this chapter until now. So, it knows that in the Kotlin world, it can accept functions:

```
class Cat {
    fun joinChoir(whatToCall: ()->Unit) {
        ...
    }
    fun leaveChoir(whatNotToCall: ()->Unit) {
        ...
    }
}
```

Previously, we learned how to pass a new function as an argument, as well as a literal function. *But how do we pass a reference to a member function?*

We can do this in the same way that we did in the Strategy design pattern; that is, by using the member reference operator (`::`):

```
val catTheConductor = Cat()
val bat = Bat()

val dog = Dog()

val turkey = Turkey()

catTheConductor.joinChoir(bat::screech)

catTheConductor.joinChoir(dog::howl)

catTheConductor.joinChoir(dog::bark)

catTheConductor.joinChoir(turkey::gobble)
```

Now, the cat needs to save all those subscribers somehow. Luckily, we can put them on a map. *What would be the key?* This should be the function itself:

```
class Cat {

    private val participants = mutableMapOf<()->Unit,
()-        >Unit>()

    fun joinChoir(whatToCall: ()->Unit) {

        participants[whatToCall] = whatToCall

    }

    ...

}
```

If all those **`()->Unit`** instances are making you dizzy, be sure to use **`typealias`** to give them more semantic meaning, such as *subscriber*.

Now, the bat decides to leave the choir. After all, no one can hear its beautiful singing:

```
class Cat {

    ...

    fun leaveChoir(whatNotToCall: ()->Unit) {
        participants.remove(whatNotToCall)
    }
    ...
}
```

All **`bat`** needs to do is pass its subscriber function again:

```
catTheConductor.leaveChoir(bat::screech)
```

That's the reason we used the map in the first place. Now, the cat can call all its choir members and tell them to sing – well, produce sounds:

```
typealias Times = Int
class Cat {
    ...
    fun conduct(n: Times) {
        for (p in participants.values) {
            for (i in 1..n) {
                p()
            }
        }
    }
}
```

So, the rehearsal went well. But the cat is very tired after doing all those loops. It would rather delegate the job to choir members. That's not a problem:

```
class Cat {
    private val participants = mutableMapOf<(Int)->Unit,
         (Int)->Unit>()
    fun joinChoir(whatToCall: (Int)->Unit) {
        ...
    }
    fun leaveChoir(whatNotToCall: (Int)->Unit) {
        ...
    }
    fun conduct(n: Times) {
        for (p in participants.values) {
            p(n)
        }
    }
}
```

Our subscribers will have to change slightly to receive a new argument. Here's an example for the **`Turkey`** class:

```
class Turkey {
    fun gobble(repeat: Times) {
        for (i in 1..repeat) {
            println("Gob-gob")
        }
    }
}
```

This is a bit of a problem. *What if the cat was to tell each animal what sound to make: high or low?* We'd have to change all the subscribers again, as well as the cat.

While designing your publisher, pass the single data classes with many properties, instead of sets of data classes or other types. That way, you won't have to refactor your subscribers as much if new properties are added:

```
enum class SoundPitch {HIGH, LOW}

data class Message(val repeat: Times, val pitch:     SoundPitch)

class Bat {

    fun screech(message: Message) {

        for (i in 1..message.repeat) {

            println("${message.pitch} Eeeeeee")

        }

    }

}
```

Here, we used **enum** to describe the different types of pitches and a data class to encapsulate the pitch to be used, as well as how many times the message should be repeated.

Make sure that your messages are immutable. *Otherwise, you may experience strange behavior! What if you have sets of different messages you're sending from the same publisher?* We could use smart casts to solve this:

```
interface Message {

    val repeat: Times

    val pitch: SoundPitch

}

data class LowMessage(override val repeat: Times) : Message {

    override val pitch = SoundPitch.LOW
```

```
}
data class HighMessage(override val repeat: Times) :
  Message {
    override val pitch = SoundPitch.HIGH
}
class Bat {
    fun screech(message: Message) {
        when (message) {
            is HighMessage -> {
                for (i in 1..message.repeat) {
                    println("${message.pitch} Eeeeeee")
                }
            }
            else -> println("Can't :(")
        }
    }
}
```

The Observer design pattern is enormously useful. Its power lies in its flexibility. The publisher doesn't need to know anything about the subscribers, except the signature of the function it invokes. In the real world, it is widely used both in reactive frameworks, which we'll discuss in Chapter 6, *Threads and Coroutines*, and Chapter 11, *Reactive Microservices with Vert.x*, and in Android, where all the UI events are implemented as subscriptions.

Summary

This was a long chapter, but we've also learned a lot. We finished covering all the classical design patterns, including 11 behavioral ones. In Kotlin, functions can be passed to other functions, returned from functions, and assigned to variables. That's what the higher-order functions and functions as first-class citizens concepts are all about. If your class is all about behavior, it often makes sense to replace it with a function. This concept helped us implement the Strategy and Command design patterns.

We learned that the Iterator design pattern is yet another `operator` in the language. Sealed classes make the `when` statements exhaustive and we used them to implement the State design pattern.

We also looked at the Interpreter design pattern and learned that lambda with a receiver allows clearer syntax in your DSLs. Another keyword, `lateinit`, tells the compiler to relax a bit when it's performing its null safety checks. *Use it with care!*

Finally, we covered how to reference an existing method with function references while talking about the Observer design pattern.

In the next chapter, we'll move on from the object-oriented programming paradigm, with its well-known design patterns, to another paradigm – functional programming.

Questions

1. What's the difference between the Mediator and Observer design patterns?

2. What is a **Domain-Specific Language** (**DSL**)?

3. What are the benefits of using a sealed class or interface?

Section 2: Reactive and Concurrent Patterns

This section focuses on modern approaches to design patterns, such as Reactive and concurrent design patterns, and functional programming in general.

We'll start this section with an introduction to the basic principles of functional programming and how its concepts are embedded in Kotlin. Then, we'll examine concurrency primitives in Kotlin, the most important being coroutines. Once we have a good grasp of both functional programming and coroutines, we'll then see how, by combining them, we can create concurrent data structures that allow us to finely control the flow of our data and the design patterns that allow us to better structure concurrent code.

This section comprises the following chapters:

Chapter 5: Introducing Functional Programming

This chapter will discuss the fundamental principles of **functional programming** and how they fit into the **Kotlin** programming language.

As you'll discover, we've already touched on some of the concepts in this chapter, as it would have been hard to discuss the benefits of the language up until now without touching on functional programming concepts such as **data immutability** and **functions as values**. But as we did before, we'll look at those features from a different angle.

In this chapter, we will cover the following topics:

- Reasoning behind the functional approach
- Immutability
- Functions as values
- Expressions, not statements
- Recursion

After completing this chapter, you'll understand how the concepts of functional programming are embedded in the Kotlin language and when to use them.

Technical requirements

For this chapter, you will need to install the following:

- **IntelliJ IDEA Community Edition** (https://www.jetbrains.com/idea/download/)
- **OpenJDK 11** (or higher) (https://openjdk.java.net/install/)

You can find the code files for this chapter on **GitHub** at https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter05.

Reasoning behind the functional approach

Functional programming has been around for as long as other programming paradigms, for example, procedural and object-oriented programming. But in the past 15 years, it has gained significant momentum. The reason for this is that something else stalled: **CPU** speeds. We cannot speed up our CPUs as much as we did in the past, so we must **parallelize** our programs. And it turns out that the functional programming paradigm is exceptionally good at running parallel tasks.

The evolution of multicore processors is a fascinating topic in itself, but we'll cover it only briefly here. Workstations have had multiple processors since at least the 1980s to support the running of tasks from different users in parallel. Since workstations were massive during this era, they didn't need to worry about cramming everything into one chip. But when multiprocessors came to the consumer market around 2005, it became necessary to have one physical unit that could work in parallel. This is why we have multiple cores in one chip in our PC or laptop.

But that's not the only reason we use functional programming. Here are a few more:

- Functional programming favors pure functions, and pure functions are usually easier to reason about and test.
- Code written in a functional way is often more declarative than imperative, dealing with the *what* instead of the *how*, and this can be a benefit.

In the following sections, we'll explore the different aspects of functional programming, starting with *immutability*.

Immutability

One of the fundamental concepts of functional programming is **immutability**. This means that from the moment the function receives input to the moment the function returns output, the object doesn't change. *But how could it change?* Well, let's look at a simple example:

```
fun <T> printAndClear(list: MutableList<T>) {
    for (e in list) {
        println(e)
        list.remove(e)
    }
}
printAndClear(mutableListOf("a", "b", "c"))
```

This code would output `a` first, and then we would receive **`ConcurrentModificationException`**.

The reason for this is that the **`for-each`** loop uses an iterator (which we already discussed in the previous chapter), and by mutating the list inside the loop, we interfere with its operation. However, this raises a question:

Wouldn't it be great if we could protect ourselves from these runtime exceptions in the first place?

Let's see how *immutable collections* can help us with this.

Immutable collections

In Chapter 1, *Getting Started with Kotlin*, we already mentioned that collections in Kotlin are immutable by default, which is unlike many other languages.

The previous problem is caused by us not following the *single-responsibility principle*, which states that a function should do only one thing and do it well. Our function tries both to remove elements from an array and to print them at the same time.

If we change the argument from **`MutableList`** to **`List`**, we won't be able to invoke the **`remove()`** function on it, resolving our current problem. But this raises another question:

What if we need an empty list?

In this case, our function should return a new object:

```
private fun <T> printAndClear(list: MutableList<T>):
  MutableList<T> {
    for (e in list) {
        println(e)
    }
    return mutableListOf()
}
```

In general, functions that don't return any values should be avoided in functional programming, as it usually means that they have a side effect.

However, it's not enough that the collection *type* is immutable. The *content* of the collection should be immutable as well. To understand this better, let's look at the following simple class:

```
data class Player(var score: Int)
```

You can see that this class has only one variable: **`score`**.

Next, we'll create a single instance of the **`data class`** and put it in an immutable collection:

```
val scores = listOf(Player(0))
```

We could put multiple instances of this class inside the collection, but to illustrate our point, only one is needed.

Next, let's introduce the concept of *threads*.

The problem with shared mutable state

If you aren't familiar with **threads**, don't worry, we'll discuss them in detail in *Chapter 6*, *Threads and Coroutines*. All you need to know for now is that threads allow the code to run *concurrently*. When using concurrent code and code that utilizes multiple CPUs, functional programming really helps.

You may find that any other example that doesn't involve concurrency at all may seem rather convoluted or artificial.

For now, let's create a list that contains two threads:

```
val threads = List(2) {
        thread {
            for (i in 1..1000) {
                scores[0].score++
            }
        }
    }
```

As you can see, each thread increments **`score`** by **`1000`** in total, using a regular **`for`** loop.

We wait for the threads to complete by using **`join()`**, and then we check the counter value:

```
for (t in threads) {
    t.join()
}
println(scores[0].score) // Less than 2000 for sure
```

If you run the code yourself, the value will be anything under **`2000`**.

This is a classic case of a *race condition* for mutable variables. The number you'll get will be different every time you run this code. The reason for this may be familiar to you if you have encountered concurrency previously. And, it has nothing to do with threads not completing their work, by the way. You can make sure of this by adding a print message after the loop:

```
thread {
    for (i in 1..1000) {
        scores[0].score = scores[0].score + 1
    }
    println("Done")
}
```

This also isn't the fault of using the increment (++) operator. As you can see, we used the long notation to increment the value, but if you run it again as many times as possible, you would still get the wrong results.

The reason for this behavior is that the addition operation and the assignment operation are not *atomic*. Two threads may override the addition operations of each other, resulting in the number not being incremented enough times.

Here, we used an extreme example of a collection that contains exactly one element. In the real world, the collections you will be working with will usually contain multiple elements. For example, you would track scores for multiple players, or even maintain a ranking system for thousands of players simultaneously. This would complicate the example even further.

What you need to remember is the following: even if a collection is immutable, it may still *contain* mutable objects inside. Mutable objects are not thread-safe.

Next, let's look at *tuples*, which are an example of immutable objects.

Tuples

In functional programming, a **tuple** is a piece of data that cannot be changed after it is created. One of the most basic tuples in Kotlin is `pair`:

```
val pair = "a" to 1
```

`pair` contains two properties – called `first` and `second` – and is immutable:

```
pair.first = "b" // Doesn't work

pair.second = 2  // Still doesn't
```

We can *destructure* `pair` into two separate values using a *destructure declaration*:

```
val (key, value) = pair

println("$key => $value")
```

When iterating over a map, we also work with another type of tuple: `Map.Entry`:

```
for (p in mapOf(1 to "Sunday", 2 to "Monday")) {

   println("${p.key} ${p.value}")

}
```

This tuple already has `key` and `value` members, instead of `first` and `second`.

In addition to `pair`, there is a `Triple` tuple that also contains a `third` value:

```
val firstEdition = Triple("Design Patterns with Kotlin",   310,
2018)
```

In general, `data` classes are usually a good implementation for tuples because they provide clear naming. If you look at the preceding example, it's not immediately obvious that the `310` value represents the number of pages.

However, as we saw in the previous section, not every `data` class is a proper tuple. You need to make sure that all of its members are *values* and not *variables*. You also need to check whether any nested collections or classes it has are immutable as well.

Now, let's discuss another important topic in functional programming: functions as a first-class citizen of the language.

Functions as values

We already covered some of the functional capabilities of Kotlin in the chapters dedicated to design patterns. The **Strategy** and **Command** design patterns are only two examples that rely heavily on the ability to accept functions as arguments, return functions, store functions as values, or put functions inside of collections. In this section, we'll cover some other aspects of functional programming in Kotlin, such as *function purity* and *currying*.

Learning about higher-order functions

As we discussed previously, in Kotlin, it's possible for a function to return another function. Let's look at the following simple function to understand this syntax in depth:

```
fun generateMultiply(): (Int) -> Int {
    return fun(x: Int): Int {
        return x * 2
    }
}
```

Here, our **`generateMultiply`** function returns another function that doesn't have a name. Functions without a name are called **anonymous functions**.

We could also rewrite the preceding code using shorter syntax:

```
fun generateMultiply(): (Int) -> Int {
    return { x: Int ->
        x * 2
    }
}
```

If a function without a name uses short syntax, it's called a **lambda function**.

Next, let's look at the signature of the return type:

```
(Int) -> Int
```

From that signature, we know that the function that we return will accept a single integer as input and produce an integer as output.

If a function doesn't accept any arguments, we denote that using empty round brackets:

```
() -> Int
```

If a function doesn't return anything, we use the **`Unit`** type to specify that:

```
(Int) -> Unit
```

Functions in Kotlin can be assigned to a variable or value to be invoked later on:

```
val multiplyFunction = generateMultiply()
...
```

```
println(multiplyFunction(3, 4))
```

The function assigned to a variable is usually called a **literal function**.

We applied this in *Chapter 4*, *Getting Familiar with Behavioral Patterns*, when discussing the Strategy design pattern.

It's also possible to specify a function as a parameter:

```
fun mathInvoker(x: Int, y: Int, mathFunction: (Int, Int) ->   Int)
{
    println(mathFunction(x, y))
}
mathInvoker(5, 6, multiplyFunction)
```

If a function is the last parameter, it can also be supplied in an ad hoc fashion, outside of the brackets:

```
mathInvoker(7, 8) { x, y ->
   x * y
}
```

This syntax is also called **trailing lambda** or **call suffix**. We saw an example of this in *Chapter 4*, *Getting Familiar with Behavioral Patterns*, when discussing the Interpreter design pattern.

Now that we've covered the basic syntax of functions, let's see how they can be used.

Higher-order functions in a standard library

When working with Kotlin, something you will be doing on a daily basis is working with *collections*. As we mentioned briefly in *Chapter 1*, *Getting Started with Kotlin*, collections have support for higher-order functions.

For example, in the previous chapters, to print elements of a collection one by one, we used a boring **`for-each`** loop:

```
val dwarfs = listOf("Dwalin", "Balin", "Kili", "Fili",   "Dori",
"Nori", "Ori", "Oin", "Gloin", "Bifur", "Bofur",   "Bombur",
"Thorin")

for (d in dwarfs) {

    println(d)

}
```

Many of you probably groaned at seeing this. But I hope you didn't stop reading the book altogether. Of course, there is also another way to achieve the same goal that is common in many programming languages: a **`forEach`** function:

```
dwarfs.forEach { d ->

    println(d)

}
```

This function is one of the most basic examples of a higher-order function. Let's see how it's declared:

```
fun <T> Iterable<T>.forEach(action: (T) -> Unit)
```

Here, **`action`** is a function that receives an element of a collection and doesn't return anything. This function presents an opportunity to discuss another aspect of Kotlin: the **`it`** notation.

The it notation

It is very common in functional programming to keep your functions small and simple. The simpler the function, the easier it is to understand, and the more chances it has to be reused in other places. And the aim *of reusing* code is one of the basic Kotlin principles.

Notice that in the preceding example, we didn't specify the type of the **d** variable. We could do this using the same colon notation we have used elsewhere:

```
dwarfs.forEach { d: String ->

    println(d)

}
```

However, usually, we don't need to do this because the compiler can figure this out from the generic types that we use. After all, **dwarfs** is of the **List<String>** type, so **d** is of the **String** type as well.

The type of the argument is not the only part that we can omit when writing short lambdas like this one. If a lambda takes a single argument, we can use the implicit name for it, which in this case, is **it**:

```
dwarfs.forEach {

    println(it)

}
```

In cases where we need to invoke a single function to a single parameter, we could also use a *function reference*. We saw an example of this in Chapter 4, *Getting Familiar with Behavioral Patterns*, when discussing the Strategy design pattern:

```
dwarfs.forEach(::println)
```

We'll use the shortest notation in most of the following examples. It is advised to use the longer syntax for cases such as *one lambda nested in another*. In those cases, giving proper names for the parameters is more important than conciseness.

Closures

In the object-oriented paradigm, state is always stored within objects. But in functional programming, this isn't necessarily the case. Let's look at the following function as an example:

```
fun counter(): () -> Int {
    var i = 0
    return { i++ }
}
```

The preceding example is clearly a higher-order function, as you can see by its **`return`** type. It returns a function with zero arguments that produces an integer.

Let's store it in a variable, in the way we've already learned, and invoke it multiple times:

```
val next = counter()
println(next())
println(next())
println(next())
```

As you can see, the function is able to keep a state, in this case, the value of a counter, even though it is not part of an object.

This is called a **closure**. The lambda has access to all of the local variables of the function that wraps it, and those local variables persist, as long as the reference to the lambda is kept.

The use of closures is another tool in the functional programming toolbox that reduces the need to define lots of classes that simply wrap a single function with some state.

Pure functions

A **pure function** is a function without any side effects. A **side effect** can be considered anything that accesses or changes the external state. The external state can be a non-local variable (where a variable from a closure is still considered to be non-local) or any kind of IO (that is, reading or writing to a file or using any kind of network capabilities).

IMPORTANT NOTE:

For those not familiar with the term, ***IO*** *stands for* ***Input/Output****, and this covers any kind of interaction that is external to our program, such as writing to files or reading from a network.*

For example, the lambda we just discussed in the *Closures* section is not considered *pure* because it can return different output for the same input when it is invoked multiple times.

Impure functions are hard to test and to reason about in general, as the result they return may depend on the order of execution or on factors that we can't control (such as network issues).

One thing to remember is that logging or even printing to a console still involves IO and is subject to the same set of problems.

Let's look at the following simple function:

```
fun sayHello() = println("Hello")
```

So, in this case, how do you ensure that Hello is printed? The task is not as simple as it seems, as we'll need some way to capture the standard output – that is, the same console where we usually see stuff printed.

We'll compare it to the following function:

```
fun hello() = "Hello"
```

The following function doesn't have any side effects. That makes it a lot easier to test:

```
fun testHello(): Boolean {

    return "Hello" == hello()

}
```

The **`hello()`** function may look a bit meaningless, but that's actually one of the properties of pure functions. Their invocation could be replaced by their result if we knew it ahead of time. This is often called **referential transparency**.

As we mentioned earlier, not every function written in Kotlin is a pure function:

```
fun <T> removeFirst(list: MutableList<T>): T {

    return list.removeAt(0)

}
```

If we call the function twice on the same list, it will return different results:

```
val list = mutableListOf(1, 2, 3)

println(removeFirst(list)) // Prints 1

println(removeFirst(list)) // Prints 2
```

Compare the preceding function to this one:

```
fun <T> withoutFirst(list: List<T>): T {

    return ArrayList(list).removeAt(0)

}
```

Now, our function is totally predictable, no matter how many times we invoke it:

```
val list = mutableListOf(1, 2, 3)

println(withoutFirst(list)) // It's 1

println(withoutFirst(list)) // Still 1
```

As you can see, in this instance, we used an immutable interface, **`List<T>`**, which helps us by preventing the possibility of mutating our input. When combined with the immutable values we discussed in the previous section, pure functions allow easier testing by providing predictable results and the parallelization of our algorithms.

A system that utilizes pure functions is easier to reason about because it doesn't rely on any external factors – what you see is what you get.

Currying

Currying is a way to translate a function that takes a number of arguments into a chain of functions, where each function takes a single argument. This may sound confusing, so let's look at a simple example:

```
fun subtract(x: Int, y: Int): Int {

    return x - y

}

println(subtract(50, 8))
```

This is a function that takes two arguments as an input and returns the difference between them. However, some languages allow us to invoke this function with the following syntax:

```
subtract(50)(8)
```

This is what currying looks like. Currying allows us to take a function with multiple arguments (in our case, two) and convert this function into a set of functions, where each one takes only a single argument.

Let's examine how this can be achieved in Kotlin. We've already seen how we can return a function from another function:

```
fun subtract(x: Int): (Int) -> Int {

    return fun(y: Int): Int {

        return x - y

    }

}
```

Here is the shorter form of the preceding code:

```
fun subtract(x: Int) = fun(y: Int): Int {

    return x - y

}
```

In the preceding example, we use single-expression syntax to return an anonymous function without the need to declare the **`return`** type or use the

`return` keyword.

And here it is in an even shorter form:

```
fun subtract(x: Int) = {y: Int -> x - y}
```

Now, an anonymous function is translated to a lambda, with the **`return`** type of the lambda inferred as well.

Although not very useful by itself, it's still an interesting concept to grasp. And if you're a **JavaScript** developer looking for a new job, make sure you understand it fully, since it's asked about in nearly every interview.

One real-world scenario where you might want to use currying is *logging*. A **`log`** function usually looks something like this:

```
enum class LogLevel {

    ERROR, WARNING, INFO

}

fun log(level: LogLevel, message: String) =     println("$level:
$message")
```

We could fix the log level by storing the function in a variable:

```
val errorLog = fun(message: String) {

    log(LogLevel.ERROR, message)

}
```

Notice that the **`errorLog`** function is easier to use than the regular **`log`** function because it accepts one argument instead of two. However, this raises a question:

What if we don't want to create all of the possible loggers ahead of time?

In this case, we can use currying. The *curried* version of this code would look like this:

```
fun createLogger(level: LogLevel): (String) -> Unit {
    return { message: String ->
        log(level, message)
    }
}
```

Now, it's up to whoever uses our code to create the logger they want:

```
val infoLogger = createLogger(LogLevel.INFO)
infoLogger("Log something")
```

This, in fact, is very similar to the Factory design pattern we covered in Chapter 2, *Working with Creational Patterns*. Again, the power of a modern language decreases the number of custom classes we need to implement to achieve the same behavior.

Next, let's talk about another powerful technique that can save us from having to do the same computation over and over again.

Memoization

If our function always returns the same output for the same input, we can easily map its input to the output, caching the results in the process. This technique is called **memoization**.

A common task when developing different types of systems or solving problems is finding a way to avoid repeating the same computation multiple times. Let's assume we receive multiple lists of integers, and for each list, we would like to print its sum:

```
val input = listOf(

    setOf(1, 2, 3),

    setOf(3, 1, 2),

    setOf(2, 3, 1),

    setOf(4, 5, 6)

)
```

Looking at the input, you can see that the first three sets are in fact equal – the difference is only in the order of the elements, so calculating the sum three times would be wasteful.

The sum calculation can be easily described as a pure function:

```
fun sum(numbers: Set<Int>): Double {

    return numbers.sumByDouble { it.toDouble() }

}
```

This function does not depend on any external state and doesn't change the external state in any way. So, it is safe for the same input to replace the call to this function with the value it had returned previously.

We could store the results of a previous computation for the same set in a mutable map:

```
val resultsCache = mutableMapOf<Set<Int>, Double>()
```

To avoid creating too many classes, we could use a higher-order function that would wrap the result in the cache that we created earlier:

```
fun summarizer(): (Set<Int>) -> Double {
    val resultsCache = mutableMapOf<Set<Int>, Double>()

    return { numbers: Set<Int> ->

        resultsCache.computeIfAbsent(numbers, ::sum)

    }

}
```

Here, we use a method reference operator (`::`) to tell **`computeIfAbsent`** to use the **`sum()`** method in the event where the input hasn't been cached yet.

Note that **`sum()`** is a pure function, while **`summarize()`** is not. The latter will behave differently for the same input. But that's exactly what we want in this case.

Running the following code on the preceding input will invoke the sum function only twice:

```
val summarizer = summarizer()

input.forEach {

    println(summarizer(it))

}
```

The combination of immutable objects, pure functions, and closures provides us with a powerful tool for performance optimization. Just remember: nothing is free. We trade one resource, CPU time, for another resource, which is memory. And it's up to you to decide which resource is more expensive in each case.

Using expressions instead of statements

A **statement** is a block of code that *doesn't return* anything. An **expression**, on the other hand, *returns a new value*. Since statements produce no results, the only way for them to be useful is to mutate the state, whether that's changing a variable, changing a data structure, or performing some kind of IO.

Functional programming tries to avoid mutating the state as much as possible. Theoretically, the more we rely on expressions, the more our functions will be pure, with all the benefits of functional purity.

We've used the `if` expression many times already, so one of its benefits should be clear: it's less verbose and, for that reason, less error-prone than the `if` statement from other languages.

Pattern matching

The concept of **pattern matching** will seem like `switch`/`case` on steroids. We've already seen how the `when` expression can be used, which we explored in Chapter 1, *Getting Started with Kotlin*, so let's briefly discuss why this concept is important for the functional paradigm.

You may know that in Java, `switch` accepts only some primitive types, strings, or enums.

Consider the following code, which is usually used to demonstrate how polymorphism is implemented in the language:

```
class Cat : Animal {

    fun purr(): String {

        return "Purr-purr";
    }
}
class Dog : Animal {
    fun bark(): String {
        return "Bark-bark";
    }
}
interface Animal
```

If we were to decide which of the functions to call, we would need to write code akin to the following:

```
fun getSound(animal: Animal): String {
    var sound: String? = null;
    if (animal is Cat) {
        sound = (animal as Cat).purr();
    }
    else if (animal is Dog) {
        sound = (animal as Dog).bark();
    }
    if (sound == null) {
        throw RuntimeException();
    }
    return sound;
}
```

This code attempts to figure out at runtime what methods the **`getSound`** class implements.

This method could be shortened by introducing multiple returns, but in real projects, multiple returns are usually a bad practice.

Since we don't have a **`switch`** statement for classes, we need to use an **`if`** statement instead.

Now, let's compare the preceding code with the following Kotlin code:

```
fun getSound(animal: Animal) = when(animal) {
    is Cat -> animal.purr()
    is Dog -> animal.bark()
    else -> throw RuntimeException("Unknown animal")
}
```

Since **`when`** is an expression, we avoided declaring the intermediate variable we previously had altogether. In addition, the code that uses pattern matching doesn't need any type checks and casts.

Now we've learned how to replace imperative **`if`** statements with much more functional **`when`** expressions, let's see how we can replace imperative loops in our code by using *recursion*.

Recursion

Recursion is a function invoking itself with new arguments. Many well-known algorithms, such as **Depth First Search**, rely on recursion.

Here is an example of a very inefficient function that uses recursion to calculate the sum of all the numbers in a given list:

```
fun sumRec(i: Int, sum: Long, numbers: List<Int>): Long {
    return if (i == numbers.size) {
        return sum
    } else {
        sumRec(i+1, numbers[i] + sum, numbers)
    }
}
```

We often try to avoid recursion due to the stack overflow errors that we may receive if our call stack is too deep. You can call this function with a list that contains a million numbers to demonstrate this:

```
val numbers = List(1_000_000) {it}
println(sumRec(0,  numbers))
// Crashed pretty soon, around 7K
```

However, Kotlin supports an optimization called **tail recursion**. One of the great benefits of tail recursion is that it avoids the dreaded stack overflow exception. If there is only a single recursive call in our function, we can use that optimization.

Let's rewrite our recursive function using a new keyword, **`tailrec`**, to avoid this problem:

```
tailrec fun sumRec(i: Int, sum: Long, numbers: List<Int>):
  Long {
    return if (i == numbers.size) {
        return sum
    } else {
        sumRec(i+1, numbers[i] + sum, numbers)
    }
}
```

Now, the compiler will optimize our call and avoid the exception completely.

However, this optimization doesn't work if you have multiple recursive calls, such as in the **Merge Sort** algorithm.

Let's examine the following function, which is the *sort* part of the Merge Sort algorithm:

```
tailrec fun mergeSort(numbers: List<Int>): List<Int> {

    return when {

        numbers.size <= 1 -> numbers

        numbers.size == 2 -> {

            return if (numbers[0] < numbers[1]) {

                numbers

            } else {

                listOf(numbers[1], numbers[0])

            }

        }

        else -> {

            val left = mergeSort(numbers.slice
              (0..numbers.size / 2))

            val right = mergeSort(numbers.slice
              (numbers.size / 2 + 1 until numbers.size))

            return merge(left, right)

        }

    }

}
```

Notice that there are two recursive calls instead of one. The Kotlin compiler will then issue the following warning:

```
> "A function is marked as tail-recursive but no tail calls are
found"
```

Summary

You should now have a better understanding of functional programming and its benefits, as well as how Kotlin approaches this topic. We've discussed the concepts of *immutability* and *pure functions*, and how combining these results in more testable code that is easier to maintain.

We discussed how Kotlin supports *closures*, which allow a function to access the variables of the function that wraps it and effectively store the state between executions. This enables techniques such as *currying* and *memoization* that allow us to fix some of the function arguments (by acting as defaults) and remember the value returned from a function in order to avoid recalculating it.

We learned that Kotlin uses the `tailrec` keyword to allow the compiler to optimize *tail recursion*. We also looked at *higher-order functions*, *expressions versus statements*, and *pattern matching*. All of these concepts allow us to write code that is easier to test and has less risk of concurrency bugs.

In the next chapter, we'll put this knowledge to practical use and discover how **reactive programming** builds upon functional programming to create scalable and resilient systems.

Questions

1. What are higher-order functions?
2. What is the `tailrec` keyword in Kotlin?
3. What are pure functions?

Chapter 6: Threads and Coroutines

In the previous chapter, we had a glance at how our application can efficiently serve thousands of requests per second—to discuss why immutability is important, we introduced a race condition problem using two threads.

In this chapter, we'll dive deeper into how to launch new threads in Kotlin and the reasons why coroutines can scale much better than threads. We will discuss how the Kotlin compiler treats coroutines and the relationship between coroutine scopes and dispatchers. We'll discuss the concept of **structured concurrency**, and how it helps us prevent resource leaks in our programs.

We'll cover the following topics in this chapter:

- Looking deeper into threads
- Introducing coroutines and suspend functions
- Starting coroutines
- Jobs
- Coroutines under the hood
- Dispatchers
- Structured concurrency

After reading this chapter, you'll be familiar with Kotlin's concurrency primitives and how to best utilize them.

Technical requirements

In addition to the requirements from the previous chapters, you will also need a **Gradle**-enabled **Kotlin** project to be able to add the required dependencies.

You can find the source code for this chapter here: https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter06.

Looking deeper into threads

Before we dive into the nuances, let's discuss what kinds of problems threads can solve.

In your laptop, you have a CPU with multiple cores – probably four of them, or even eight. This means that it can do four different computations *in parallel*, which is pretty amazing considering that 15 years ago, a single-core CPU was the default and even two cores were only for enthusiasts.

But even back then, you were not limited to doing only a single task at a time, right? You could listen to music and browse the internet at the same time, even on a single-core CPU. *How does your CPU manage to pull that off?* Well, the same way your brain does. It juggles tasks. When you're reading a book while listening to your friend talking, part of the time, you're not reading, and part of the time, you're not listening – that is, until we get at least two cores in our brains.

The servers you run your code on have pretty much the same CPU. This means that they can serve four requests simultaneously. *But what if you* *have 10,000 requests per second?* You can't serve them in parallel because you don't have 10,000 CPU cores. But you can try and serve them concurrently.

The most basic concurrency model provided by JVM is known as a **thread**. Threads allow us to run code concurrently (but not necessarily in parallel) so that we can make better use of multiple CPU cores, for example. They are more lightweight than processes. One process may spawn hundreds of threads. Unlike processes, sharing data between threads is easy. But that also introduces a lot of problems, as we'll see later.

Let's learn how to create two threads in Java first. Each thread will output numbers between `0` and `100`:

```
for (int t = 0; t < 2; t++) {

    int finalT = t;

    new Thread(() -> {

        for (int i = 0; i < 100; i++) {

            System.out.println("Thread " + finalT + ":
              " + i);

        }

    }).start();

}
```

The output will look something like this:

```
> ...

> T0: 12

> T0: 13

> T1: 60
> T0: 14

> T1: 61

> T0: 15

> T1: 16

> ...
```

Note that the output will vary between executions and that at no point is it guaranteed to be interleaved.

The same code in Kotlin would look as follows:

```
repeat(2) { t ->

    thread {

        for (i in 1..100) {

            println("T$t: $i")

        }

    }

}
```

In Kotlin, there's less boilerplate because there's a function that helps us create a new thread. Notice that, unlike Java, we don't need to call **`start()`** to launch the thread. It starts by default. If we would like to postpone it for later, we can set the **`start`** parameter to **`false`**:

```
val t = thread(start = false)

...

// Later

t.start()
```

Another useful concept from Java is **daemon threads**. These threads don't prevent JVM from exiting and are very good for non-critical background tasks.

In Java, the API is not fluent, so we'll have to assign our thread to a variable, set it to be a daemon thread, and then start it. In Kotlin, this is much simpler:

```
thread(isDaemon = true) {
    for (i in 1..1_000_000) {
        println("daemon thread says: $i")
    }
}
```

Notice that although we asked this thread to print numbers up to one million, it prints only a few hundred. That's because it's a daemon thread. When the parent thread stops, all the daemon threads stop as well.

Thread safety

There are entire books written about **thread safety** and there are good reasons for this. Concurrency bugs that are caused by a lack of thread safety are the hardest ones to track. They're hard to reproduce because you'll usually need a lot of threads competing for the same resource in order for an actual race to happen. Because this book is about Kotlin and not thread safety in general, we'll only scratch the surface of this topic. If you're interested in the topic of thread safety in the JVM language, you should check out the book *Java Concurrency in Practice*, by Brian Goetz.

We'll start with the following example, which creates 100,000 threads to increment a `counter`. To make sure that all the threads complete their work before we check the value, we'll use **`CountDownLatch`**:

```
var counter = 0

val latch = CountDownLatch(100_000)

repeat(100) {

    thread {

        repeat(1000) {

            counter++

            latch.countDown()

        }

    }

}

latch.await()

println("Counter $counter")
```

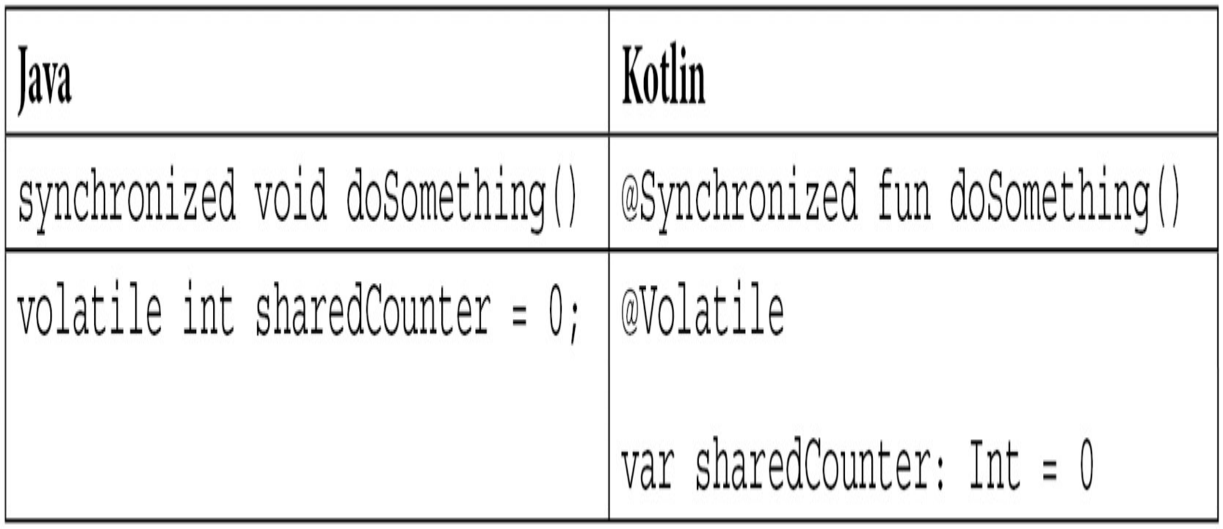
The reason this code doesn't print the correct number is that we introduced a data race since the **`++`** operation is not atomic. So, if more threads try to increment our counter, then there are more chances for data races.

Unlike Java, there's no **`synchronized`** keyword in Kotlin. The reason for this is that Kotlin designers believe that a language shouldn't be tailored to a particular concurrency model. Instead, there's a **`synchronized()`** function we can use:

```
thread {

    repeat(1000) {

        synchronized(latch) {

            counter++

            latch.countDown()
            }
        }
}
```

Now, our code prints `100,000`, as expected.

If you miss the synchronized methods from Java, there's the `@Synchronized` annotation in Kotlin. Java's `volatile` keyword is also replaced by the `@Volatile` annotation instead. The following table shows us an example of this comparison:



Java	Kotlin
`synchronized void doSomething()`	`@Synchronized fun doSomething()`

`volatile int sharedCounter = 0;`	`@Volatile`
	
	`var sharedCounter: Int = 0`

Table 6.1 – Comparison between Java and Kotlin (synchronized and volatile methods)

The reason `Synchronized` and `Volatile` are annotations and not keywords is because Kotlin can be compiled on other platforms in addition to JVM. But the concepts of `synchronized` methods or `volatile` variables exist for JVM specifically.

Why are threads expensive?

There is a price to pay whenever we create a new thread. Each thread needs a new memory stack.

What if we simulate some work inside each thread by putting it to sleep?

In the following piece of code, we'll attempt to create 10,000 threads, each sleeping for a relatively short period:

```
val counter = AtomicInteger()

try {

    for (i in 0..10_000) {

        thread {

            counter.incrementAndGet()

            Thread.sleep(100)

        }

    }

} catch (oome: OutOfMemoryError) {

    println("Spawned ${counter.get()} threads before
      crashing")

    System.exit(-42)

}
```

Each thread requires one megabyte of RAM for its stack. Creating so many threads will require lots of communication with your operating system and a lot of memory. We attempt to identify whether we ran out of memory by catching the relevant exception.

Depending on your operating system, this will result in either **`OutOfMemoryError`** or the entire system becoming very slow.

Of course, there are ways to limit how many threads are run at once using the **Executors API**. This API was introduced back in **Java 5**, so it should be pretty well-known to you.

Using that API, we can create a new thread pool of a specified size. Try setting the **`pool`** size to **`1`**, the number of cores on your machine to **`100`** and **`2000`**, and see what happens:

```
val pool = Executors.newFixedThreadPool(100)
```

Now, we would like to submit a new task. We can do this by calling **`pool.submit()`**:

```
val counter = AtomicInteger(0)

val start = System.currentTimeMillis()

for (i in 1..10_000) {

    pool.submit {

        // Do something

        counter.incrementAndGet()

        // Simulate wait on IO

        Thread.sleep(100)

        // Do something again

        counter.incrementAndGet()

    }

}
```

By incrementing **`counter`** once before **`sleep`** and once after, we are simulating some business logic – for example, preparing some JSON and then parsing the response – while **`sleep`** itself simulates a network operation.

Then, we need to make sure that the pool terminates and give it `20` seconds to do so by using the following lines:

```
pool.awaitTermination(20, TimeUnit.SECONDS)

pool.shutdown()

println("Took me ${System.currentTimeMillis() - start}   millis to
complete ${counter.get() / 2} tasks")
```

Notice that it took us 20 seconds to complete. That's because a new task cannot begin until the previous tasks *wake up* and finish their jobs.

And that's exactly what happens in a multithreaded system that is not concurrent enough.

In the next section, we'll discuss how coroutines try to solve this problem.

Introducing coroutines

In addition to the threading model provided by Java, Kotlin also has a **coroutines** model. Coroutines might be considered lightweight threads, and we'll see what advantages they provide over an existing model of threads shortly.

The first thing you need to know is that coroutines are not part of the language. They are simply another library provided by JetBrains. For that reason, if we want to use them, we need to specify this in our Gradle configuration file; that is, **`build.gradle.kts`**:

```
dependencies {

    ...

    implementation("org.jetbrains.kotlinx:kotlinx-      coroutines-
core:1.5.1")
}
```

IMPORTANT NOTE:

By the time you read this book, the latest version of the Coroutines library will be ***1.6*** *or greater.*

First, we will compare starting a new thread and a new coroutine.

Starting coroutines

We've already seen how to start a new thread in Kotlin in the *Looking deeper into threads* section. Now, let's start a new coroutine instead.

We'll create almost the same example we did with threads. Each coroutine will increment some counter, sleep for a while to emulate some kind of I/O, and then increment it again:

```
val latch = CountDownLatch(10_000)
val c = AtomicInteger()
val start = System.currentTimeMillis()
for (i in 1..10_000) {
    GlobalScope.launch {
        c.incrementAndGet()
        delay(100)
        c.incrementAndGet()
        latch.countDown()
    }
}
latch.await(10, TimeUnit.SECONDS)
```

```
println("Executed ${c.get() / 2} coroutines in
  ${System.currentTimeMillis() - start}ms")
```

The first way of starting a new coroutine is by using the `launch()` function. Again, note that this is simply another function and not a language construct.

Another interesting point here is the call to the `delay()` function, which we use to simulate some I/O-bound work, such as fetching something from a database or over the network.

Like the `Thread.sleep()` method, it puts the current coroutine to sleep. But unlike `Thread.sleep()`, other coroutines can work while it sleeps soundly. This is because `delay()` is marked with a `suspend` keyword, which we'll discuss in the *Jobs* section.

If you run this code, you'll see that the task takes about 200 ms with coroutines, while with threads, it either takes 20 seconds or runs out of memory. And we didn't have to change our code that much. That's all thanks to the fact that coroutines are highly concurrent. They can be suspended without blocking the thread that runs them. Not blocking a thread is great because we can use fewer OS threads (which are expensive) to do more work.

If you run this code in your IntelliJ IDEA, you'll notice that `GlobalScope` is marked as a **delicate API**. This means that `GlobalScope` shouldn't be used in real-world projects unless the developer understands how it works under the hood. Otherwise, it may cause unintended leaks. We'll learn about better ways of launching coroutines later in this chapter.

Although we've seen that coroutines are much more concurrent than threads, there's nothing magical in them. Now, let's learn about another way of starting a coroutine, as well as some issues coroutines may still suffer from.

The **`launch()`** function that we just discussed starts a coroutine that doesn't return anything. In contrast, the **`async()`** function starts a coroutine that returns some value.

Calling **`launch()`** is much like calling a function that returns **`Unit`**. But most of our functions return some kind of result. For that purpose, we have the **`async()`** function. It also launches a coroutine, but instead of returning a job, it returns **`Deferred<T>`**, where **`T`** is the type that you expect to get later.

For example, the following function will start a coroutine that generates a UUID asynchronously and returns it:

```
fun fastUuidAsync() = GlobalScope.async {

    UUID.randomUUID()

}

println(fastUuidAsync())
```

If we run the following code from our **`main`** method, though, it won't print the expected result. The result that this code prints instead of some UUID value is as follows:

```
> DeferredCoroutine{Active}
```

The returned object from a coroutine is called a job. Let's understand what this is and how to use it correctly.

Jobs

The result of running an asynchronous task is called a **job**. Much like the `Thread` object represents an actual OS thread, the `job` object represents an actual coroutine.

This means that what we tried to do is this:

```
val job: Job = fastUuidAsync()

println(job)
```

`job` has a simple life cycle. It can be in one of the following states:

- **New**: Created but not started yet.
- **Active**: Just created by the `launch()` function, for example. This is the default state.
- **Completed**: Everything went well.
- **Canceled**: Something went wrong.

Two more states are relevant to jobs that have child jobs:

- **Completing**: Waiting to finish executing children before completing
- **Canceling**: Waiting to finish executing children before canceling

If you want to learn more about parent and child jobs, jump to the *Parent jobs* section of this chapter.

The job we've confused with its value is in the Active state, meaning that it hasn't finished computing our UUID yet.

A job that has a value is known as being `Deffered`:

```
val job: Deferred<UUID> = fastUuidAsync()
```

We'll discuss the **Deferred** value in more detail in Chapter 8, *Designing for Concurrency*.

To wait for a job to complete and get the actual value, we can use the **await()** function:

```
val job: Deferred<UUID> = fastUuidAsync()

println(job.await())
```

This code doesn't compile, though:

```
> Suspend function 'await' should be called only from a coroutine
or another suspend function
```

The reason for this is that, as stated in the error itself, our **main()** function is not marked with a **suspend** keyword and isn't a coroutine either.

We can fix this by wrapping our code in a **runBlocking** function:

```
runBlocking {

    val job: Deferred<UUID> = fastUuidAsync()

    println(job.await())

}
```

This function will block our main thread until all the coroutines finish. It is an implementation of the Bridge design pattern from Chapter 4, *Getting Familiar with Behavioral Patterns*, which allows us to connect between regular code and code that uses coroutines.

Running this code now will produce the expected output of some random UUID.

IMPORTANT NOTE:

In this chapter, while discussing coroutines, we will sometimes omit **`runBlocking`** *for conciseness. You can always find the full working examples in this book's GitHub repository.*

The **`job`** object also has some other useful methods, which we'll discuss in the following sections.

Coroutines under the hood

So, we've mentioned the following facts a couple of times:

- Coroutines are like lightweight threads. They need fewer resources than regular threads, so you can create more of them.
- Instead of blocking an entire thread, coroutines suspend themselves, allowing the thread to execute another piece of code in the meantime.

But how do coroutines work?

As an example, let's take a look at a function that composes a user profile:

```
fun profileBlocking(id: String): Profile {

    // Takes 1s

    val bio = fetchBioOverHttpBlocking(id)

    // Takes 100ms

    val picture = fetchPictureFromDBBlocking(id)

    // Takes 500ms

    val friends = fetchFriendsFromDBBlocking(id)

    return Profile(bio, picture, friends)

}
```

Here, our function takes around 1.6 seconds to complete. Its execution is completely sequential, and the executing thread will be blocked for the entire time.

We can redesign this function so that it works with coroutines, as follows:

```
suspend fun profile(id: String): Profile {

    // Takes 1s

    val bio = fetchBioOverHttpAsync(id)

    // Takes 100ms

    val picture = fetchPictureFromDBAsync(id)

    // Takes 500ms

    val friends = fetchFriendsFromDBAsync(id)

    return Profile(bio.await(), picture.await(),
      friends.await())

}
```

Without the **`suspend`** keyword, our asynchronous code simply won't compile. We'll cover what the **`suspend`** keyword means later in this section.

To understand what each of the asynchronous functions looks like, let's take a look at one of them as an example:

```
fun fetchFriendsFromDBAsync(id: String) = GlobalScope.async

{

    delay(500)

    emptyList<String>()

}
```

Now, let's compare the performance of the two functions: one that is written in a blocking manner, and another that uses coroutines.

We can wrap both functions using a **`runBlocking`** function, as we've seen previously, and measure the time it takes them to complete using **`measureTimeMillis`**:

```
runBlocking {
    val t1 = measureTimeMillis {
        blockingProfile("123")
    }
    val t2 = measureTimeMillis {
        profile("123")
    }
    println("Blocking code: $t1")
    println("Async: $t2")
}
```

The output will be something like this:

```
> Blocking code: 1623
> Coroutines: 1021
```

The execution time of the concurrent coroutines is the maximum of the longest coroutine, while with sequential code, it's the sum of all functions.

Having understood the first two examples, let's look at another way to write the same code.

We'll mark each of the functions with the **`suspend`** keyword:

```
suspend fun fetchFriendsFromDB(id: String): List<String> {
    delay(500)
    return emptyList()
}
```

If you run this example, the performance will be the same as the blocking code. *So, why would we want to use suspendable functions?*

Suspendable functions don't block the thread. Looking at the bigger picture, by using the same number of threads, we can serve far more users, all thanks to the smart way Kotlin rewrites suspendable functions.

When the Kotlin compiler sees the **`suspend`** keyword, it knows it can split and rewrite the function, like this:

```
fun profile(state: Int, id: String, context: ArrayList<Any>):
Profile {

    when (state) {

        0 -> {

            context += fetchBioOverHttp(id)

            profile(1, id, context)

        }

        1 -> {

            context += fetchPictureFromDB(id)

            profile(2, id, context)

        }

        2 -> {

            context += fetchFriendsFromDB(id)

            profile(3, id, context)

        }

        3 -> {

            val (bio, picture, friends) = context
            return Profile(bio, picture, friends)
        }
    }
}
```

This rewritten code uses the **State design pattern** from Chapter 4, *Getting Familiar with Behavioral Patterns*, to split the execution of the function into many steps. By doing so, we can release the thread that executes coroutines at every stage of the state machine.

IMPORTANT NOTE:

This is not a perfect depiction of the generated code. The goal is to demonstrate the idea behind what the Kotlin compiler does, but some subtle implementation details are omitted for brevity.

Note that unlike the asynchronous code we produced earlier, the state machine itself is sequential and takes the same amount of time as the blocking code to execute all its steps.

It is a fact that none of these steps block any threads, which is important in this example.

Canceling a coroutine

If you are a Java developer, you may know that stopping a thread is quite complicated.

For example, the `Thread.stop()` method is deprecated. There's `Thread.interrupt()`, but not all threads are checking this flag, not to mention setting a `volatile` flag, which is often suggested but is very cumbersome.

If you're using a thread pool, you'll get **`Future`**, which has the **`cancel(boolean mayInterruptIfRunning)`** method. In Kotlin, the **`launch()`** function returns a job.

This job can be canceled. The same rules from the previous example apply, though. If your coroutine never calls another **`suspend`** function or the **`yield`** function, it will disregard **`cancel()`**.

To demonstrate that, we'll create one coroutine that yields once in a while:

```
val cancellable = launch {
    try {
        for (i in 1..10_000) {
            println("Cancellable: $i")
            yield()
        }
    }
    catch (e: CancellationException) {
        e.printStackTrace()
    }
}
```

As you can see, after each **`print`** statement, the coroutine calls the **`yield`** function. If it was canceled, it will print the stack trace.

We'll also create another coroutine that doesn't yield:

```
val notCancellable = launch {
    for (i in 1..10_000) {
        if (i % 100 == 0) {
            println("Not cancellable $i")
        }
    }
}
```

This coroutine never yields and prints its results every `100` iterations to avoid spamming the console.

Now, let's try cancelling both coroutines:

```
println("Canceling cancellable")
cancellable.cancel()
println("Canceling not cancellable")
notCancellable.cancel()
```

Then, we'll wait for the results:

```
runBlocking {
    cancellable.join()
    notCancellable.join()
}
```

By invoking `join()`, we can wait for the execution of the coroutine to complete.

Let's look at the output of our code:

```
> Canceling cancellable
> Cancellable: 1
> Not cancellable 100
>...
> Not cancellable 1000
> Canceling not cancellable
```

A few interesting points we can learn from this experiment regarding the behavior of coroutines are as follows:

- Canceling the **`cancellable`** coroutine doesn't happen immediately. It may still print a line or two before being canceled.
- We can catch **`CancellationException`**, but our coroutine will be marked as canceled anyway. Catching that exception doesn't automatically allow us to continue.

Now, let's understand what happened. The coroutine checks whether it was canceled, but only when it is switching between states. Since the non-cancellable coroutine didn't have any suspending functions, it never checked if it was asked to stop.

In the **`cancellable`** coroutine, we used a new function: **`yield()`**. We could have called **`yield()`** on every loop iteration, but decided to do that every 100th one. This function checks whether there is anybody else that wants to do some work. If there's nobody else, the execution of the current coroutine will resume. Otherwise, another coroutine will start or resume from the point where it stopped earlier.

Note that without the **`suspend`** keyword on our function or a coroutine generator, such as **`launch()`**, we can't call **`yield()`**. This is true for any function marked with **`suspend`**: it should either be called from another **`suspend`** function or from a coroutine.

Setting timeouts

Let's consider the following situation. *What if, as happens in some cases, fetching the user's profile takes too long? What if we decided that if the profile takes more than 0.5 seconds to return, we'll just show no profile?*

This can be achieved using the **`withTimeout()`** function:

```
val coroutine = async {
    withTimeout(500) {
        try {
            val time = Random.nextLong(1000)
            println("It will take me $time to do")
            delay(time)
            println("Returning profile")
            "Profile"
        }
        catch (e: TimeoutCancellationException) {
            e.printStackTrace()
        }
    }
}
```

We set the timeout to be **`500`** milliseconds, and our coroutine will delay for between **`0`** and **`1000`** milliseconds, giving it a 50 percent chance of failing.

We'll **`await`** the results from the coroutine and see what happens:

```
val result = try {
    coroutine.await()
}
```

```
catch (e: TimeoutCancellationException) {
    "No Profile"
}
println(result)
```

Here, we benefit from the fact that **`try`** is an expression in Kotlin. So, we can return a result immediately from it.

If the coroutine manages to return before the timeout, the value of **`result`** becomes **`profile`**. Otherwise, we receive **`TimeoutCancellationException`** and set the value of **`result`** to **`no profile`**.

A combination of timeouts and **`try-catch`** expressions is a really powerful tool that allows us to create robust interactions.

Dispatchers

When we ran our coroutines using the **`runBlocking`** function, their code was executed on the main thread.

You can check this by running the following code:

```
runBlocking {
    launch {
        println(Thread.currentThread().name) // Prints
          "main"
    }
}
```

In contrast, when we run a coroutine using **`GlobalScope`**, it runs on something called **`DefaultDispatcher`**:

```
GlobalScope.launch {
    println("GlobalScope.launch:
      ${Thread.currentThread().name}")
}
```

This prints the following output:

```
> DefaultDispatcher-worker-1
```

`DefaultDispatcher` is a thread pool that is used for short-lived coroutines.

Coroutine generators, such as **`launch()`** and **`async()`**, rely on default arguments, one of which is the dispatcher they will be launched on. To specify an alternative dispatcher, you can provide it as an argument to the coroutine builder:

```
runBlocking {
    launch(Dispatchers.Default) {
        println(Thread.currentThread().name)
    }
}
```

The preceding code prints the following output:

```
> DefaultDispatcher-worker-1
```

In addition to the **`Main`** and **`Default`** dispatchers, which we've already discussed, there is also an **`IO`** dispatcher, which is used for long-running tasks. You can use it similarly for other dispatchers by providing it to the coroutine builder, like so:

```
async(Dispatchers.IO) {
    // Some long running task here
}
```

Structured concurrency

It is a very common practice to spawn coroutines from inside another coroutine.

The first rule of structured concurrency is that the parent coroutine should always wait for all its children to complete. This prevents resource leaks, which is very common in languages that don't have the **structured concurrency** concept.

This means that if we look at the following code, which starts 10 child coroutines, the parent coroutine doesn't need to wait explicitly for all of them to complete:

```
val parent = launch(Dispatchers.Default) {
    val children = List(10) { childId ->
        launch {
            for (i in 1..1_000_000) {
                UUID.randomUUID()
                if (i % 100_000 == 0) {
                    println("$childId - $i")
                    yield()
                }
            }
        }
    }
}
```

Now, let's decide that one of the coroutines throws an exception after some time:

```
...
if (i % 100_000 == 0) {
    println("$childId - $i")
    yield()
}
if (childId == 8 && i == 300_000) {
    throw RuntimeException("Something bad happened")
}
...
```

If you run this code, something interesting happens. Not only does the coroutine itself terminate, but also all its siblings are terminated as well.

What happens here is that an uncaught exception bubbles up to the parent coroutine and cancels it. Then, the parent coroutine terminates all the other child coroutines to prevent any resource leaks.

Usually, this is the desired behavior. If we'd like to prevent child exceptions from stopping the parent as well, we can use **`supervisorScope`**:

```
val parent = launch(Dispatchers.Default) {
    supervisorScope {
        val children = List(10) { childId ->
            ...
        }
    }
}
```

By using `supervisorScope`, even if one of the coroutines fails, the parent job won't be affected.

The parent coroutine can still terminate all its children by using the `cancel()` function. Once we invoke `cancel()` on the parent job, all of its children are canceled too.

Now that we've discussed the benefits of structured concurrency, let's reiterate one point from the start of this chapter: using `GlobalScope` and the fact that it's marked as a **delicate API**. Although `GlobalScope` exposes functions such as `launch()` and `async()`, it doesn't benefit from structured concurrency principles and is prone to resource leaks when used incorrectly. For that reason, you should avoid using `GlobalScope` in real-world applications.

Summary

In this chapter, we covered how to create threads and coroutines in Kotlin, as well as the benefits of coroutines over threads.

Kotlin has simplified syntax for creating threads, compared to Java. But it still has the overhead of memory and, often, performance. Coroutines can solve these issues; use coroutines whenever you need to execute some code concurrently in Kotlin.

At this point, you should know how to start a coroutine and how to wait for it to complete, getting its results in the process. We also covered how coroutines are structured and learned about how they interact with dispatchers.

Finally, we touched upon the topic of structured concurrency, a modern idea that helps us prevent resource leaks in concurrent code easily.

In the next chapter, we'll discuss how we can use these concurrency primitives to create scalable and robust systems that suit our needs.

Questions

1. What are the different ways to start a coroutine in Kotlin?
2. With structured concurrency, if one of the coroutines fails, all the siblings will be canceled as well. How can we prevent that behavior?
3. What is the purpose of the `yield()` function?

Chapter 7: Controlling the Data Flow

The previous chapter covered an important **Kotlin** concurrency primitive: **coroutines**. In this chapter, we'll discuss two other vital concurrent primitives in Kotlin: **channels** and **flows**. We'll also touch on **higher-order functions** for **collections**, as their API is very similar to that of channels and flows.

The idea of making extensive use of small, reusable, and composable functions comes directly from the **functional programming** paradigm, which we discussed in the previous chapter. These functions allow us to write code in a manner that describes *what* we want to do instead of *how* we want to do it.

In this chapter, we'll cover the following topics:

- Reactive principles
- Higher-order functions for collections
- Concurrent data structures
- Sequences
- Channels
- Flows

After reading this chapter, you'll be able to efficiently communicate between different coroutines and process your data with ease.

Technical requirements

In addition to the technical requirements from the previous chapters, you will also need a **Gradle**-enabled Kotlin project to be able to add the required dependencies.

You can find the source code used in this chapter on **GitHub** at the following location:

https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter07

Reactive principles

We'll start this chapter with a brief detour into **Reactive programming**, as it forms the foundation of the **data streaming** concept.

Reactive programming is a paradigm based on functional programming in which we model our logic as a set of operations in a data stream. The fundamental concepts of reactive programming are summarized nicely in *The Reactive Manifesto* (https://www.reactivemanifesto.org).

According to this manifesto, reactive programs should be all of the following:

- Responsive
- Resilient
- Elastic
- Message-driven

To understand these four principles, we'll use an example.

Let's imagine you are calling your Internet Service Provider, since your internet is slow, for example. *Do you have this picture in your mind?* Let's start then.

Responsive principle

How much time are you willing to spend waiting on the line? That depends on the urgency of the situation and how much time you have. If you're in a hurry, you'll probably drop the call sooner rather than later because you don't know how much time you'll need to wait while listening to that horrible music.

That's the system being *unresponsive* to you. This also happens with web systems. A request to a web server may get stuck in a queue when waiting for other requests to be processed.

On the other hand, a responsive call center may tell you in a pleasant voice once in a while how many people are in the queue before you – or even how much time you'll have to wait.

In both cases, the result is the same. You've wasted your time waiting on the line. But the second system was responsive to your needs, and you could make decisions based on that.

Resilient principle

Let's move on to the **resilient** principle. Imagine you're waiting on the line for 10 minutes and then the line drops. That's the system not being resilient to failures.

The Reactive Manifesto recommends several ways to achieve resiliency:

- **Delegation**: You'll probably hear, "*Our current representative is unable to resolve your slow internet; we are redirecting you to someone else.*"
- **Replication**: Then, you may hear, "*We are aware that many people are on the line; we are adding more representatives as we speak.*" This also relates to *elasticity*, which we'll cover in the next section.
- **Containment** and **isolation**: Finally, the automatic voice tells you, "*If you don't want to wait, please leave your number and we'll get back to you.*" *Containment* means that you are now decoupled from the scalability problems the system is having (that is, the system not having enough representatives). In contrast, *isolation* means that even if the system has issues with a phone line not being reliable, you don't care.

Elastic principle

In the previous section, we discussed replication. To prevent failures, our call center always has at least three representatives on shift. Maybe all of them are answering calls, or perhaps they're just patiently waiting.

What happens, though, if some rabid mole chews through the internet cable?

Suddenly, there is a surge of calls from disgruntled customers.

If our call center has only three phones, there is not much we can do about this. But if we had some extra resources, we could bring more representatives in to handle the incident and calm our customers. And after the cable was finally fixed, we could let them go back to their business. That's the system being *elastic* in response to the workload.

Elasticity builds on *scalability*. For example, we could manage all of the incoming calls if each representative could work independently by having their own phone. If we had more representatives than phones, the number of phones would become a *bottleneck*, with some representatives unable to answer any calls.

Message-driven principle

The **message-driven** principle is also referred to as **asynchronous message passing**. So, in the previous section, we saw that if you could leave a message for any representative to call back, it could make the system more resilient.

So, what if all customers only leave messages?

Then, each representative could *prioritize* those messages or *batch* them. For example, printing all of the billing receipts together instead of working through the messages in a random order.

Using messages also allows applying backpressure. If a representative receives too many messages, they may collapse from stress. To avoid that, they may text you to say that you'll have to wait a bit longer to receive your answer. Again, we're also talking about *delegation* here, as all of these principles overlap.

Messages are also *non-blocking*. After you leave the message, you don't sit there waiting for the representative's response. Instead, you usually go back to your regular tasks. The ability to perform other tasks while you wait is one of the cornerstones of *concurrency*.

In this section, we learned about the four reactive principles. Reactive applications are responsive, resilient, elastic, and message-driven. In the following sections, we'll see how these principles are applied in Kotlin. We'll start with *collections*, or as they'd be referred to in reactive programming terms, *static data streams*.

Higher-order functions on collections

We briefly touched on this topic in *Chapter 1*, *Getting Started with Kotlin*, but before we can discuss streams, let's make sure that those of us who come from languages that don't have higher-order functions on collections know what they are, what they do, and what the benefits of using them are.

We won't be able to cover all of the functions available on collections, but we'll cover the most widely used ones.

Mapping elements

The `map()` function takes each element of a collection and returns a new element of a possibly different type. To understand this idea better, let's say we have a list of letters and we would like to output their ASCII values.

First, let's implement it in an imperative way:

```
val letters = 'a'..'z'

val ascii = mutableListOf<Int>()

for (l in letters) {
    ascii.add(l.toInt())
}
```

Notice that even for such a trivial task, we had to write quite a lot of code. We also had to define our output list as mutable.

Now, the same code using the **`map()`** function would look like this:

```
val result: List<Int> = ('a'..'z').map { it.toInt() }
```

Notice how much shorter the implementation is. We don't need to define a mutable list, nor do we need to write a **`for-each`** loop ourselves.

Filtering elements

Another common task is filtering a collection. You know the drill – you iterate over it and only put values that fit your criteria in a new collection. For example, if given a range of numbers between `1` and `100`, we would like to return only those that are divisible by `3` or divisible by `5`.

In the imperative way, this function might look something like this:

```
val numbers = 1..100
val notFizzbuzz = mutableListOf<Int>()
for (n in numbers) {
    if (n % 3 == 0 || n % 5 == 0) {
        notFizzbuzz.add(n)
    }
}
```

In its functional variant, we would use the **`filter()`** function:

```
val filtered: List<Int> = (1..100).filter { it % 3 == 0 ||
  it % 5 == 0 }
```

Again, notice how much more concise our code becomes. We only specify *what* needs to be done, filtering elements that match the criteria, and not *how* this should be done (for example, using an **`if`** statement).

Finding elements

Finding the first element in a collection is another common task. If we were to write a function for finding a number that is divisible by both **`3`** and **`5`**, we could implement it like this:

```
fun findFizzbuzz(numbers: List<Int>): Int? {
    for (n in numbers) {
        if (n % 3 == 0 && n % 5 == 0) {
            return n
        }
    }
    return null
}
```

The same functionality can be achieved using the **`find`** function:

```
val found: Int? = (1..100).find { it % 3 == 0 && it % 5 ==   0 }
```

In a similar way to the preceding imperative function, the **`find`** function returns **`null`** if there is no element that meets our criteria.

There's also an accompanying **`findLast()`** method, which does the same, but which starts with the last element of the collection.

Executing code for each element

All previous families of functions had one common characteristic: they all resulted in a stream. But not all the higher-order functions return streams. Some will return a single value, such as **`Unit`** or, for example, a number. Those functions are called **terminator functions**.

In this section, we'll deal with the first terminator function. Terminator functions return something else rather than a new collection, so you can't chain the result of this call to other calls. Therefore, they *terminate* the chain.

In the case of **`forEach()`**, it returns the result of the **`Unit`** type. The **`Unit`** type is akin to **`void`** in **Java** and means that the function doesn't return anything useful. So, the **`forEach()`** function is like the plain old **`for`** loop:

```
val numbers = (0..5)

numbers.map { it * it}          // Can continue

       .filter { it < 20 }      // Can continue

       .forEach { println(it) } // Cannot continue
```

Note that **`forEach()`** has some minor performance impacts compared to the traditional **`for`** loop.

There's also **`forEachIndexed()`**, which provides an index in the collection alongside the actual value:

```
numbers.map { it * it }

        .forEachIndexed { index, value ->

    print("$index:$value, ")

}
```

The output for the preceding code will be as follows:

```
> 0:1, 1:4, 2:9, 3:16, 4:25,
```

Since Kotlin 1.1, there's also the **`onEach()`** function, which is a bit more useful because it returns the collection again:

```
numbers.map { it * it}
       .filter { it < 20 }
       .sortedDescending()
       .onEach { println(it) } // Can continue now
       .filter { it > 5 }
```

As you can see, this function is not terminating.

Summing up elements

Much like **`forEach()`**, **`reduce()`** is a terminating function. But instead of terminating with **`Unit`**, which is not very useful, it terminates with a single value of the same type as the collection it operates on.

To see how **`reduce()`** works in practice, let's summarize all numbers between **`1`** and **`100`**:

```
val numbers = 1..100
var sum = 0
for (n in numbers) {
    sum += n
}
```

Now, let's write the same code using **`reduce`**:

```
val reduced: Int = (1..100).reduce { sum, n -> sum + n }
```

Note that here it lets us avoid declaring a mutable variable for storing the sum of the elements. Unlike previous higher-order functions we've seen, **`reduce()`** receives not one but two arguments. The first argument is the accumulator. In the imperative example, it's the **`sum`** variable. The second argument is the next element. We used the same names for the arguments, so it should be relatively easy to compare both implementations.

Getting rid of nesting

Sometimes when working with collections, we may end up with a *collection of collections*. For example, consider the following code:

```
val listOfLists: List<List<Int>> = listOf(listOf(1, 2), listOf(3,
4, 5), listOf(6, 7, 8))
```

But what if we wanted to turn this collection into a single list containing all of the nested elements?

Then, the output would look like this:

```
> [1, 2, 3, 4, 5, 6, 7, 8]
```

One option is to iterate our input and use the **`addAll`** method that the mutable collections have:

```
val flattened = mutableListOf<Int>()

for (list in listOfLists) {

    flattened.addAll(list)

}
```

A better option is to use a **`flatMap()`** function, which will do the same:

```
val flattened: List<Int> = listOfLists.flatMap { it }
```

This concrete example could be simplified even further by using a **`flatten()`** function:

```
val flattened: List<Int> = listOfLists.flatten()
```

But the **`flatMap()`** function is usually more useful, as it allows you to apply other functions to each collection, in an **Adapter** like pattern.

There are many other higher-order functions declared on collections, so we couldn't cover all of them in this short section. You must browse through the official documentation and learn about them. Nevertheless, the functions discussed previously should provide a solid ground for the next topic we'll cover.

Now, when you're familiar with how to transform and iterate over the *static data streams*, let's see how we can apply the same operations to *dynamic data streams*.

Exploring concurrent data structures

Now we're familiar with some of the most common higher-order functions on collections, let's combine this knowledge with what we learned in the previous chapter about concurrency primitives in Kotlin to discuss the *concurrent data structures* Kotlin provides.

The two most essential concurrent data structures are *channels* and *flows*. However, before we can discuss them, we need to look at another data structure: **sequences**. While this data structure is not concurrent itself, it will provide us with a bridge into the concurrent world.

Sequences

Higher-order functions on collections existed in many functional programming languages for a long time. But for Java developers, the higher-order functions for collections first appeared in Java 8 with the introduction of the **Stream API**.

Despite providing developers with valuable functions such as `map()`, `filter()`, and some of the others we already discussed, there were two major drawbacks to the Stream API. First, in order to use these functions, you had to migrate to Java 8. And second, your collection had to be converted to something called a **stream**, which had all of the functions defined on it. If you want to return a collection again after mapping and filtering your stream, you can collect it back.

There is also another significant difference between streams and collections. Unlike collections, streams can be infinite. Since Kotlin doesn't limit itself to only **JVM** and is also backward-compatible to Java 6, it needed to provide another solution for the possibility of infinite collections. This solution was named **sequence** to avoid clashing with Java streams when they're available.

We can create a new sequence using the `generateSequence()` function. For example, the next function will create an infinite sequence of numbers:

```
val seq: Sequence<Long> = generateSequence(1L) { it + 1 }
```

As the first argument we specify the initial value, while the second argument is a lambda that generates the next value based on the previous one. The returned type, as you can see, is `Sequence`.

A regular collection or a range can be converted to a sequence using the **`asSequence()`** function:

```
(1..100).asSequence()
```

If we need to build a sequence using more complex logic, you can use a **`sequence()`** builder:

```
val fibSeq = sequence {
    var a = 0
    var b = 1
    yield(a)
    yield(b)
    while (true) {
        yield(a + b)
        val t = a
        a = b
        b += t
    }
}
```

In this example, we create a sequence of Fibonacci numbers. Then, we use the **`yield()`** function to return the next value in the series. Every time the sequence is used, the code will resume from the last **`yield()`** function invoked.

While the concept of sequences doesn't seem very useful in itself, there is a significant difference between sequences and collections. Sequences are *lazy*, while collections are *eager*.

This means that using higher-order functions on collections has a hidden cost for collections beyond a certain size. Most of them will copy the collection for the sake of immutability.

To understand this difference, let's look at the following code. First, we'll create a list containing a million numbers and measure how much time it takes to square each number in the list – once while operating on a *collection* and another while working on a *sequence*:

```
val numbers = (1..1_000_000).toList()
println(measureTimeMillis {
    numbers.map {
        it * it
    }.take(1).forEach { it }
}) // ~50ms
println(measureTimeMillis {
    numbers.asSequence().map {
        it * it
    }.take(1).forEach { it }
}) // ~5ms
```

We use the **`take()`** function, which is another higher-order function on collections, to *take* just the first element of the calculation.

You can see that the code that uses a sequence executes much faster. This is because sequences, being lazy, execute the chain for each element. This means that only a single number from the entire list is squared.

On the other hand, functions on collections work on the entire collection. This means that first, all of the numbers are squared, then put in a new collection, and only a single number is taken from the results.

Sequences, channels, and flows follow the *reactive principles*, so it's essential to understand them before moving on. Note that reactive principles are not tied to functional programming. You can also be reactive while writing object-oriented or procedural code. However, it's still easier to discuss these principles after learning about functional programming and its foundations.

Channels

In the previous chapter, we learned how to spawn coroutines and control them.

But, what if two coroutines need to communicate with each other?

In Java, threads communicate either by using the **`wait()`**/**`notify()`**/**`notifyAll()`** pattern or by using one of the rich set of classes from the **`java.util.concurrent`** package – for example, **`BlockingQueue`**.

In Kotlin, as you may have noticed, there are no **`wait()`**/**`notify()`** methods. Instead, to communicate between coroutines, Kotlin uses channels. **Channels** are very similar to **`BlockingQueue`**, but instead of blocking a thread, channels suspend a coroutine, which is a lot cheaper. We'll use the following steps to create a channel and a coroutine:

1. First, let's create a channel:

   ```
   val chan = Channel<Int>()
   ```

   Channels are typed. This channel can only receive integers.

2. Then, let's create a coroutine that reads from this channel:

   ```
   launch {
       for (c in chan) {
           println(c)
       }
   }
   ```

   Reading from a channel is as simple as using a **`for-each`** loop.

3. Now, let's send some values to this channel. This is as simple as using the **`send()`** function:

   ```
   (1..10).forEach {
       chan.send(it)
   }
   chan.close()
   ```

4. Finally, we close the channel. Once closed, the coroutine that listens to the channel will also break out of the **`for-each`** loop, and if there's nothing else to do, the coroutine will terminate.

This style of communication is called **Communicating Sequential Processes**, or more simply, **CSP**.

As you can see, channels are a convenient and type-safe way to communicate between different coroutines. But we had to define the channels manually. In the following two sections, we'll see how this can be further simplified.

Producers

If we need a coroutine that supplies a stream of values, we could use the **`produce()`** function. This function creates a coroutine that is backed up by **`ReceiveChannel<T>`**, where **`T`** is the type the coroutine produces.

We could rewrite the example from the previous section, as follows, by using the **`produce()`** function:

```
val chan = produce {

    (1..10).forEach {

        send(it)

    }

}

launch {

    for (c in chan) {

        println(c)

    }

}
```

Note that inside the **`produce()`** block, the **`send()`** function is readily available for us to push new values to the channel.

Instead of using a **`for-each`** loop in our consumer coroutine, we can use a **`consumeEach()`** function:

```
launch {

    chan.consumeEach {

        println(it)

    }

}
```

Now, it's time to look at another example where a coroutine is bound to a channel.

Actors

Similar to **`producer()`**, **`actor()`** is a coroutine bound to a channel. But instead of a channel going *out* of the coroutine, there's a channel going *into* the coroutine.

Let's look at the following example:

```
val actor = actor<Int> {

    channel.consumeEach {

        println(it)

    }

}

(1..10).forEach {

    actor.send(it)

}
```

In this example, our main function is again producing the values and the actors consume them through the channel. This is very similar to the first example we saw, but instead of explicitly creating a channel and a separate coroutine, we have them bundled together.

If you've worked with **Scala** or any other programming language that has actors, you may be familiar with a slightly different actor model from what we've described. For example, in some implementations, actors have both inbound and outbound channels (often called **mailboxes**). But in Kotlin, an actor has only an inbound mailbox in the form of a channel.

Buffered channels

In all of the previous examples, whether creating channels explicitly or implicitly, we in fact used their *unbuffered* version.

To demonstrate what this means, let's take a look at a slightly altered example from the previous section:

```
val actor = actor<Long> {

    var prev = 0L

    channel.consumeEach {

        println(it - prev)

        prev = it

        delay(100)

    }

}
```

Here, we have almost the same **`actor`** object, which receives timestamps and prints the difference between every two timestamps it gets. We also introduce a small delay before it can read the next value.

Instead of sending a sequence of numbers, we would send the current timestamp to this **`actor`** object:

```
repeat(10) {

    actor.send(System.currentTimeMillis())

}

actor.close().also { println("Done sending") }
```

Now, let's take a look at the output of our code:

```
> ...

> 101
> 103

> 101

> Done sending
```

Notice that our producer is suspended until the channel is ready to accept the next value. Therefore, the **`actor`** object is able to apply backpressure on the producer, telling it not to send the next value until the **`actor`** object is ready.

Now, let's make a minor change to the way we define our **`actor`** object:

```
val actor = actor<Long>(capacity = 10) {

...

}
```

Every channel has a *capacity*, which is zero by default. This means until a value is consumed from a channel, no other value can be sent over it.

Now, if we run our code again, we'll see a completely different output:

```
> Done sending

> ...

> 0

> 0
```

The producer doesn't have to wait for the consumer anymore because the channel now buffers the messages. So, the messages are sent as fast as possible and the actor is still able to consume them at its own pace.

In a similar manner, **`capacity`** could be defined on the producer channel:

```
val chan = produce(capacity = 10) {

    (1..10).forEach {
        send(it)
    }
}
```

And it could be defined on the raw channel as well:

```
val chan = Channel<Int>(10)
```

Buffered channels are a very powerful concept that allow us to *decouple* producers from consumers. You should use them carefully, though, as the larger the capacity of the channel is, the more memory it will require.

Channels are a relatively low-level concurrency construct. So, let's take a look at another type of stream, which provides us with a higher level of abstraction.

Flows

A **flow** is a cold, asynchronous stream and is an implementation of the **Observable design pattern** we covered in *Chapter 4*, *Getting Familiar with Behavioral Patterns*.

As a quick reminder, the Observable design pattern has two methods: **`subscribe()`** (which allows consumers to, well, subscribe for messages) and **`publish()`** (which sends a new message to all of the subscribers).

The publish method of the **`Flow`** object is called **`emit()`**, while the subscribe method is called **`collect()`**.

We can create a new flow using the **`flow()`** function:

```
val numbersFlow: Flow<Int> = flow {
    ...
}
```

Inside the **`flow`** constructor, we can use the **`emit()`** function to publish a new value to all listeners.

For example, here we create a flow that would publish ten numbers using the **`flow`** constructor:

```
flow {
    (0..10).forEach {
        println("Sending $it")
        emit(it)
    }
}
```

Now that we've covered how to publish a message, let's discuss how to subscribe to a flow.

For that, we can use the **`collect()`** function available on the **`flow`** object:

```
numbersFlow.collect { number ->
    println("Listener received $number")
}
```

If you run this code now, you'll see that the listener prints all the numbers it receives from the flow.

Unlike some other reactive frameworks and libraries, there is no special syntax to raise an exception to the listener. Instead, we can simply use the standard **`throw`** expression to do that:

```
flow {
    (1..10).forEach {
```

```
    ...
        if (it == 9) {
            throw RuntimeException()
        }
    }
}
```

From the listener side, handling exceptions is as simple as wrapping the **`collect()`** function in a **`try`**/**`catch`** block:

```
try {
    numbersFlow.collect { number ->
        println("Listenerreceived $number")
    }
}
catch (e: Exception) {
    println("Got an error")
}
```

Like channels, the Kotlin flows are suspending, but they are not concurrent. Flows support backpressure, although this is completely transparent to the user. To see what this means, let's create multiple subscribers for the same flow:

```
(1..4).forEach { coroutineId ->
    delay(5000)
    launch(Dispatchers.Default) {
        numbersFlow.collect { number ->
            delay(1000)
            println("Coroutine $coroutineId received
              $number")
        }
    }
}
```

Each subscriber runs in its own coroutine, with a delay of five seconds between each new subscription. This allows us to see them run concurrently.

Now, let's take a look at the output:

```
> ...
> Sending 1
> Coroutine 1 received 5
> Sending 6
> Coroutine 2 received 1
> Sending 2
> Coroutine 1 received 6
> ...
```

From this output, we can learn two important lessons:

- **Flows are cold streams**: This means for each new subscriber, the flow starts anew. In our case, each new subscriber will receive all numbers, starting from `1`.
- **Flows use backpressure**: Note that the next number is not sent until the previous number is received. This is similar to the behavior of

unbuffered channels and different from buffered channels, where the producer can send numbers faster than the consumer can consume them.

Next, let's see how these two properties of flows can be altered, if necessary.

Buffering flows

In some cases, for example, when we have plenty of available memory, we aren't interested in applying backpressure on the producer right away. To do so, each consumer can specify that the flow should be *buffered* by using the **`buffer()`** function:

```
numbersFlow.buffer().collect { number ->
    delay(1000)
    println("Coroutine $coroutineId received $number")
}
```

If we look at the output of the preceding code again, we'll see a dramatic change:

```
> ...
> Sending 8
> Sending 9
> Sending 10
> Coroutine 1 received 1
> Coroutine 1 received 2
> ...
```

With a buffer, the flow produces values without any backpressure from the consumer until the buffer is filled. Then, the consumer is still able to collect the values at its own pace. This behavior is similar to buffered channels, and in fact, the implementation uses a channel under the hood.

Buffering a flow is useful when it takes a considerable amount of time to process each message. Take uploading images from your phone as an example. Of course, the upload will take a different amount of time based on the size of the image. You don't want to block the user interface until the image is uploaded because that would be a bad user experience and against reactive principles.

Instead, you could define a buffer that fits into the memory, upload the images at your own pace, and block the user interface only once the buffer is full of tasks.

In the case of images, we are dealing with a series of elements we don't want to lose. So, let's consider a different example, where we could allow dropping some of the elements in our flow.

Conflating flows

Imagine we have a flow that produces changes in stock prices at a rate of ten times a second, and we have a UI that needs to display the latest stock values. To do this, we'll just use a number that goes up by 1 for every tick:

```
val stock: Flow<Int> = flow {

    var i = 0

    while (true) {

        emit(++i)

        delay(100)

    }

}
```

The UI itself, however, doesn't have to be refreshed ten times every second. Once every second is more than enough. If we simply try to use **`collect()`**, as in the previous example, we'll be constantly behind the producer:

```
var seconds = 0

stock.collect { number ->

    delay(1000)

    seconds++

    println("$seconds seconds -> received $number")

}
```

The preceding code outputs the following:

```
> 1 seconds -> received 1

> 2 seconds -> received 2

> 3 seconds -> received 3

> ...
```

The preceding output is incorrect. The reason for this is that we apply backpressure to the flow, slowing it down. Another option would be to buffer 10 values, as we've seen in the previous example. But since we want to refresh the UI ten times slower than the flow refreshes itself, we'll have to discard nine values out of ten. We'll leave it to the readers to try and implement that logic.

A better solution would be to *conflate* the flow. A conflated flow doesn't store all of the messages. Instead, it keeps only the most recent values. We implement this in the following code:

```
stock.conflate().collect { number ->

    delay(1000)
    seconds++
    println("$seconds seconds -> received $number")
}
```

Let's first look at the output:

```
> ...
> 4 seconds -> received 30
> 5 seconds -> received 40
> 6 seconds -> received 49
> ...
```

You can see that now the values are correct. On average, our counter is incremented ten times every second.

Now, our flow will never be suspended and the subscriber will receive only the most recent value that the flow has calculated.

Summary

This chapter was dedicated to practicing functional programming with reactive principles and learning the building blocks of functional programming in Kotlin. We also learned about the main benefits of reactive systems. For example, such systems should be responsive, resilient, elastic, and driven by messaging.

Now, you should know how to transform your data, filter your collections, and find elements within the collection that meet your criteria.

You should also better understand the difference between *cold* and *hot* streams. A cold stream, such as a *flow*, starts working only when someone subscribes to it. A new subscriber will usually receive all of the events. On the other hand, a hot stream, such as a *channel*, continuously emits events, even if nobody is listening to them. A new subscriber will receive only the events that were sent after the subscription was made.

We also discussed the concept of backpressure, which can be implemented in a flow. For example, if the consumer is not able to process all of the events, it may suspend the producer, buffer the events in the hope of catching up, or conflate the stream, handling only some of the events.

The next chapter will cover concurrent design patterns, which allow us to architect concurrent systems in a scalable, maintainable, and extensible manner, using coroutines and reactive streams as building blocks.

Questions

1. What is the difference between higher-order functions on collections and on concurrent data structures?
2. What is the difference between cold and hot streams of data?
3. When should a conflated channel or flow be used?

Chapter 8: Designing for Concurrency

Concurrent design patterns help us to manage many tasks at once and structure their life cycle. By using these patterns efficiently, we can avoid problems such as resource leaks and deadlocks.

In this chapter, we'll discuss concurrent design patterns and how they are implemented in **Kotlin**. To do this, we'll be using the building blocks from previous chapters: coroutines, channels, flows, and concepts from **functional programming**.

We will be covering the following topics in this chapter:

- Deferred value
- Barrier
- Scheduler
- Pipeline
- Fan out
- Fan in
- Racing
- Mutex
- Sidekick channel

After completing this chapter, you'll be able to work with asynchronous values efficiently, coordinate the work of different coroutines, and distribute and aggregate work, as well as have the tools needed to resolve any concurrency problems that may arise in the process.

Technical requirements

In addition to the technical requirements from the previous chapters, you will also need a **Gradle**-enabled Kotlin project to be able to add the required dependencies.

You can find the source code used in this chapter on **GitHub** at the following location:

https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter08

Deferred Value

The goal of the **Deferred Value** design pattern is to return a reference to a result of an asynchronous computation. A **Future** in **Java** and **Scala**, and a **Promise** in **JavaScript** are both implementations of the Deferred Value design pattern.

We've already discussed **deferred values** in *Chapter 6*, *Threads and Coroutines*. We've seen that the `async()` function returns a type called `Deferred`, which is also an implementation of this design pattern.

Interestingly enough, the `Deferred` value itself is an implementation of both the **Proxy** design pattern that we've seen in *Chapter 3*, *Understanding Structural Patterns*, and the **State** design pattern from *Chapter 4*, *Getting Familiar with Behavioral Patterns*.

We can create a new container for the result of an asynchronous computation using the **`CompletableDeferred`** constructor:

```
val deferred = CompletableDeferred<String>()
```

To populate the **`Deferred`** value with a result, we use the **`complete()`** function, and if an error occurs in the process, we can use the **`completeExceptionally()`** function to pass the exception to the caller. To understand it better, let's write a function that returns an asynchronous result. Half of the time the result will contain **`OK`**, and the other half of the time it will contain an exception.

```
suspend fun valueAsync(): Deferred<String> = coroutineScope {
    val deferred = CompletableDeferred<String>()
    launch {
        delay(100)
        if (Random.nextBoolean()) {
            deferred.complete("OK")
        }
        else {
            deferred.completeExceptionally(
              RuntimeException()
            )
        }
    }
    deferred
}
```

You can see that we return the **`Deferred`** value almost immediately, then we start an asynchronous computation using **`launch`** and simulate some computation using the **`delay()`** function.

Since the process is asynchronous, the results won't be ready immediately. To wait for the results, we can use the **`await()`** function that we've already discussed in Chapter 6, *Threads and Coroutines*:

```
runBlocking {
    val value = valueAsync()
    println(value.await())
}
```

It's important to make sure that you always complete your **`Deferred`** value by calling either of the **`complete()`** or **`completeExceptionally()`** functions. Otherwise, your program may wait indefinitely for the results. It is also possible to cancel **`deferred`** if you're no longer interested in its results. To do this, simply call **`cancel()`** on it:

```
deferred.cancel()
```

You'll rarely need to create your own deferred value. Usually, you would work with the one returned from the **`async()`** function.

Next, let's discuss how to wait for multiple asynchronous results at once.

Barrier

The **Barrier** design pattern provides us with the ability to wait for multiple concurrent tasks to complete before proceeding further. A common use case for this is composing objects from different sources.

For example, take the following class:

```
data class FavoriteCharacter(

    val name: String,

    val catchphrase: String,

    val picture: ByteArray = Random.nextBytes(42)

)
```

Let's assume that the **`catchphrase`** data comes from one service and the **`picture`** data comes from another. We would like to fetch these two pieces of data concurrently:

```
fun CoroutineScope.getCatchphraseAsync

(

    characterName: String

) = async { … }

fun CoroutineScope.getPicture

(

    characterName: String

) = async { … }
```

The most basic way to implement concurrent fetching would be as follows:

```
suspend fun fetchFavoriteCharacter(name: String) = coroutineScope {

    val catchphrase = getCatchphraseAsync(name).await()

    val picture = getPicture(name).await()

    FavoriteCharacter(name, catchphrase, picture)

}
```

But this solution has a major problem – we don't start fetching the **`picture`** data until the **`catchphrase`** data was fetched. In other words, the code is unnecessarily *sequential*. Let's see how this can be improved.

Using data classes as barriers

We can slightly alter the previous code to achieve the concurrency we want:

```
suspend fun fetchFavoriteCharacter(name: String) = coroutineScope {

    val catchphrase = getCatchphraseAsync(name)

    val picture = getPicture(name)

    FavoriteCharacter(name, catchphrase.await(),
      picture.await())

}
```

Moving the **`await`** function into the invocation of the data class constructor allows us to start all of the coroutines at once and then wait for them to complete, just as we wanted.

The additional benefit of using data classes as barriers is the ability to *destructure* them easily:

```
val (name, catchphrase, _) = fetchFavoriteCharacter("Inigo
Montoya")

println("$name says: $catchphrase")
```

This works well if the type of data we receive from different asynchronous tasks is heterogeneous. In some cases, we receive the same types of data from different sources.

For example, let's ask **`Michael`** (our canary product owner), **`Taylor`** (our barista), and **`Me`** who our favorite movie character is:

```
object Michael {

    suspend fun getFavoriteCharacter() = coroutineScope {

        async {

            FavoriteCharacter("Terminator",

              "Hasta la vista, baby")

        }

    }

}

object Taylor {

    suspend fun getFavoriteCharacter() = coroutineScope {

        async {

            FavoriteCharacter("Don Vito Corleone", "I'm

              going to make him an offer he can't refuse")

        }

    }

}

object Me {

    suspend fun getFavoriteCharacter() = coroutineScope {

        async {

            // I already prepared the answer!

            FavoriteCharacter("Inigo Montoya",
              "Hello, my name is...")

        }
    }
}
```

Here, we have three very similar objects that differ only in the contents of the asynchronous results they return.

In this case, we can use a list to gather the results:

```
val characters: List<Deferred<FavoriteCharacter>> =    listOf(
        Me.getFavoriteCharacter(),
        Taylor.getFavoriteCharacter(),
        Michael.getFavoriteCharacter(),
    )
```

Notice the type of the list. It's a collection of the **`Deferred`** elements of the **`FavoriteCharacter`** type. On such collections, there's an **`awaitAll()`** function available that acts as a barrier as well:

```
println(characters.awaitAll())
```

When working with a set of homogenous asynchronous results and you need all of them to complete before proceeding further, use **`awaitAll()`**.

The Barrier design pattern creates a rendezvous point for multiple asynchronous tasks. The next pattern will help us abstract the execution of those tasks.

Scheduler

The goal of the **Scheduler** design pattern is to decouple *what* is being run from *how* it's being run and optimize the use of resources when doing so.

In Kotlin, **Dispatchers** are an implementation of the Scheduler design pattern that decouple the coroutine (that is, the *what*) from underlying thread pools (that is, the *how*).

We've already seen dispatchers briefly in Chapter 6, *Threads and Coroutines*.

To remind you, the coroutine builders such as **`launch()`** and **`async()`** can specify which dispatcher to use. Here's an example of how you specify it explicitly:

```
runBlocking {
    // This will use the Dispatcher from the parent
    // coroutine
    launch {
        // Prints: main
        println(Thread.currentThread().name)
    }
    launch(Dispatchers.Default) {
        // Prints DefaultDispatcher-worker-1
        println(Thread.currentThread().name)
    }
}
```

The default dispatcher creates as many threads as you have CPUs in the underlying thread pool. Another dispatcher that is available to you is the **IO Dispatcher**:

```
async(Dispatchers.IO) {
    for (i in 1..1000) {
        println(Thread.currentThread().name)

        yield()

    }

}
```

This will output the following:

```
> …

> DefaultDispatcher-worker-2

> DefaultDispatcher-worker-1

> DefaultDispatcher-worker-1

> DefaultDispatcher-worker-1

> DefaultDispatcher-worker-3

> DefaultDispatcher-worker-3

> ...
```

The IO Dispatcher is used for potentially long-running or blocking operations and will create up to 64 threads for that purpose. Since our example code doesn't do much, the IO Dispatcher doesn't need to create many threads. That's why you'll see only a small number of workers used in this example.

Creating your own schedulers

We are not limited to the dispatchers Kotlin provides. We can also define dispatchers of our own.

Here is an example of creating a dispatcher that would use a dedicated thread pool of `4` threads based on `ForkJoinPool`, which is efficient for *divide-and-conquer* tasks:

```
val forkJoinPool = ForkJoinPool(4).asCoroutineDispatcher()


repeat(1000) {

    launch(forkJoinPool) {

        println(Thread.currentThread().name)

    }

}
```

If you create your own dispatcher, make sure that you either release it with **`close()`** or reuse it, as creating a new dispatcher and holding to it is expensive in terms of resources.

Pipeline

The **Pipeline** design pattern allows us to scale heterogeneous work, consisting of multiple steps of varying complexity across multiple CPUs, by breaking the work into smaller, concurrent pieces. Let's look at the following example to understand it better.

Back in Chapter 4, *Getting Familiar with Behavioral Patterns*, we wrote an HTML page parser. It was assumed that the HTML pages themselves were already fetched for us, though. What we would like to design now is a process that would create a possibly infinite stream of pages.

First, we would like to fetch news pages once in a while. For that, we'll have a producer:

```
fun CoroutineScope.producePages() = produce {
    fun getPages(): List<String> {
        // This should actually fetch something
        return listOf(
            "<html><body><h1>
               Cool stuff</h1></body></html>",
            "<html><body><h1>
               Even more stuff</h1></body></html>"
        )
    }
    val pages = getPages()

    while (this.isActive) {
        for (p in pages) {
            send(p)
        }
    }
}
```

The **`isActive`** flag will be true as long as the coroutine is running and hasn't been canceled. It is a good practice to check this property in loops that may run for a long time so they can be stopped between iterations if needed.

Each time we receive new titles, we send them downstream. Since tech news isn't updated very often, we can check for updates only once in a while by using **`delay()`**. In the actual code, the delay would probably be minutes, if not hours.

The next step is creating a **Document Object Model** (**DOM**) out of those raw strings containing HTML. For that, we'll have a second producer, with this one receiving a channel that connects it to the first one:

```
fun CoroutineScope.produceDom(pages: ReceiveChannel<String>) =
produce {

    fun parseDom(page: String): Document {

        // In reality this would use a DOM library to parse

        // string to DOM

        return Document(page)

    }

    for (p in pages) {

        send(parseDom(p))

    }

}
```

We can use the `for` loop to iterate over the channel as long as it's still open. This is a very elegant way of consuming data from an asynchronous source without the need to define callbacks.

We'll have a third function that receives the parsed documents and extracts the title out of each one:

```
fun CoroutineScope.produceTitles(parsedPages:
ReceiveChannel<Document>) = produce {

    fun getTitles(dom: Document): List<String> {

        return dom.getElementsByTagName("h1").map {

            it.toString()

        }
    }
    for (page in parsedPages) {
        for (t in getTitles(page)) {
            send(t)
        }
    }
}
```

We're looking for the headers, and so we use **`getElementsByTagName("H1")`**. For each header found, we turn it into its string representation.

Now, we will move on toward composing our coroutines into pipelines.

Composing a pipeline

Now that we've familiarized ourselves with the components of the pipeline, let's see how we can combine multiple components together:

```
runBlocking {
    val pagesProducer = producePages()
    val domProducer = produceDom(pagesProducer)
    val titleProducer = produceTitles(domProducer)
    titleProducer.consumeEach {
        println(it)
    }
}
```

The resulting pipeline will look as follows:

```
Input=>pagesProducer=>domProducer=>titleProducer=>Output
```

A pipeline is a great way to break a long process into smaller steps. Note that each resulting coroutine is a *pure function*, so it's also easy to test and reason about.

The entire pipeline could be stopped by calling **`cancel()`** on the first coroutine in line.

Fan Out

The goal of the **Fan Out** design pattern is to distribute work between multiple concurrent processors, also known as *workers*. To understand it better, let's look again at the previous section but consider the following problem:

What if the amount of work at the different steps in our pipeline is very different?

For example, it takes a lot more time to *fetch* the HTML content than to *parse* it. In such a case, we may want to distribute that heavy work between multiple coroutines. In the previous example, only a single coroutine was reading from each channel. But multiple coroutines can consume from a single channel too, thus dividing the work.

To simplify the problem we're about to discuss, let's have only one coroutine producing some results:

```
fun CoroutineScope.generateWork() = produce {
    for (i in 1..10_000) {
        send("page$i")
    }
    close()
}
```

And we'll have a function that creates a new coroutine that reads those results:

```
fun CoroutineScope.doWork(
    id: Int,
    channel: ReceiveChannel<String>
) = launch(Dispatchers.Default) {
    for (p in channel) {
        println("Worker $id processed $p")
    }
}
```

This function will generate a coroutine that is executed on the **`Default`** dispatcher. Each coroutine will listen to a channel and print every message it receives to the console.

Now, let's start our producer. Remember that all the following pieces of code need to be wrapped in the **`runBlocking`** function, but for simplicity, we omitted that part:

```
val workChannel = generateWork()
```

Then, we can create multiple workers that distribute the work between themselves by reading from the same channel:

```
val workers = List(10) { id ->
    doWork(id, workChannel)
}
```

Let's now examine a part of the output of this program:

```
> ...
> Worker 4 processed page9994
> Worker 8 processed page9993
> Worker 3 processed page9992
> Worker 6 processed page9987
```

Note that no two workers receive the same message and the messages are not being printed in the order they were sent. The Fan Out design pattern allows us to efficiently distribute the work across a number of coroutines, threads, and CPUs.

Next, let's discuss an accompanying design pattern that often goes hand-in-hand with Fan Out.

Fan In

The goal of the **Fan In** design pattern is to combine results from multiple workers. This design pattern is helpful when our workers produce results and we need to gather them.

This design pattern is the opposite of the Fan Out design pattern we discussed in the previous section. Instead of multiple coroutines *reading* from the same channel, multiple coroutines can *write* their results to the same channel.

Combining the Fan Out and Fan In design patterns is a good base for **MapReduce** algorithms. To demonstrate this, we'll slightly change the workers from the previous example, as follows:

```
}

private fun CoroutineScope.doWorkAsync(
    channel: ReceiveChannel<String>,
    resultChannel: Channel<String>
) = async(Dispatchers.Default) {
    for (p in channel) {
        resultChannel.send(p.repeat(2))
    }
}
```

Now, once done, each worker sends the results of its calculation to **`resultChannel`**.

Note that this pattern is different from the actor and producer builders we've seen before. Actors each have their own channels, while in this case, **`resultChannel`** is shared across all the workers.

To collect the results from the workers, we'll use the following code:

```
runBlocking {
    val workChannel = generateWork()
    val resultChannel = Channel<String>()
    val workers = List(10) {
        doWorkAsync(workChannel, resultChannel)
    }
    resultChannel.consumeEach {
        println(it)
    }
}
```

Let's now clarify what this code does:

1. First, we create **`resultChannel`**, which all our workers will share.
2. Then, we supply it to each worker. We have ten workers in total. Each worker repeats the message it received twice and sends it on **`resultChannel`**.
3. Finally, we consume the results from the channel in our main coroutine. This way, we accumulate results from multiple concurrent workers in the same place.

Here's a sample of the output from the preceding code:

```
> ...
> page9995page9995
> page9996page9996
> page9997page9997
> page9999page9999
> page9998page9998
> page10000page10000
```

Next, let's discuss another design pattern, which will help us improve the responsiveness of our code in some cases.

Racing

Racing is a design pattern that runs multiple jobs concurrently, picking the result that returns first as the *winner* and discarding others as *losers*.

We can implement Racing in Kotlin using the **`select()`** function on channels.

Let's imagine you are building a weather application. For redundancy, you fetch the weather from two different sources, *Precise Weather* and *Weather Today*. We'll describe them as two producers that return their name and temperature.

If we have more than one producer, we can subscribe to their channels and take the first result that is available.

First, let's declare the two weather producers:

```
fun CoroutineScope.preciseWeather() = produce {

    delay(Random.nextLong(100))

    send("Precise Weather" to "+25c")

}

fun CoroutineScope.weatherToday() = produce {

    delay(Random.nextLong(100))

    send("Weather Today" to "+24c")

}
```

Their logic is pretty much the same. Both wait for a random number of milliseconds and then return a temperature reading and the name of the source.

We can listen to both channels simultaneously using the **`select`** expression:

```
runBlocking {

  val winner = select<Pair<String, String>> {

    preciseWeather().onReceive { preciseWeatherResult ->
```

```
            preciseWeatherResult
        }
        weatherToday().onReceive { weatherTodayResult ->
            weatherTodayResult
        }
    }
    println(winner)
}
```

Using the **`onReceive()`** function allows us to listen to multiple channels simultaneously.

Running this code multiple times will randomly print **`(Precise Weather, +25c)`** and **`(Weather Today, +24c)`**, as there is an equal chance for both of them to arrive first.

Racing is a very useful concept when you are willing to sacrifice resources in order to get the most responsiveness from your system and we achieved that using Kotlin's **`select`** expression. Now, let's explore the **`select`** expression a little further to discover another concurrent design pattern that it implements.

Unbiased select

When using the **`select`** clause, the order is important. Because it is inherently biased, if two events happen at the same time, it will select the first clause.

Let's see what that means in the following example.

We'll have only one producer this time, which sends over a channel which movie we should watch next:

```
fun CoroutineScope.fastProducer(
    movieName: String
) = produce(capacity = 1) {
    send(movieName)
}
```

Since we defined a non-zero capacity on the channel, the value will be available as soon as this coroutine runs.

Now, let's start the two producers and use a **`select`** expression to see which of the two movies will be selected:

```
runBlocking {
    val firstOption = fastProducer("Quick&Angry 7")
    val secondOption = fastProducer(
      "Revengers: Penultimatum")
    delay(10)
    val movieToWatch = select<String> {
        firstOption.onReceive { it }
        secondOption.onReceive { it }
    }
    println(movieToWatch)
}
```

No matter how many times you run this code, the winner will always be the same: **`Quick&Angry 7`**. This is because if both values are ready at the same time, the **`select`** clause will always pick the first channel available in the order they are declared.

Now, let's use **`selectUnbiased`** instead of the **`select`** clause:

```
...
val movieToWatch = selectUnbiased<String> {
    firstOption.onReceive { it }
    secondOption.onReceive { it }
}
...
```

Running this code now will sometimes produce **`Quick&Angry 7`** and sometimes produce **`Revengers: Penultimatum`**. Unlike the regular **`select`** clause, **`selectUnbiased`** doesn't care about the order. If more than one result is available, it will pick one randomly.

Mutex

Also known as **mutual exclusions**, **mutex** provides a means to protect a shared state that can be accessed by multiple coroutines at once.

Let's start with the same old dreaded **`counter`** example, where multiple concurrent tasks try to update the same **`counter`**:

```
var counter = 0
val jobs = List(10) {
    async(Dispatchers.Default) {
        repeat(1000) {
            counter++
        }
    }
}
jobs.awaitAll()
println(counter)
```

As you've probably guessed, the result that is printed is less than 10,000 – *totally embarrassing!*

To solve this, we can introduce a locking mechanism that will allow only a single coroutine to interact with the variable at once, making the operation *atomic*.

Each coroutine will try to obtain the ownership of the **`counter`**. If another coroutine is updating the **`counter`**, our coroutine will wait patiently and then try to acquire the lock again. Once updated, it must release the lock so that other coroutines can proceed:

```
var counter = 0
val mutex = Mutex()
val jobs = List(10) {
    launch {
        repeat(1000) {
            mutex.lock()
            counter++
            mutex.unlock()
        }
    }
}
```

Now, our example always prints the correct number: **`10,000`**.

Mutex in Kotlin is different from the Java mutex. In Java, **`lock()`** on a mutex blocks the thread, until the lock can be acquired. A Kotlin mutex suspends the coroutine instead, providing better concurrency. Locks in Kotlin are cheaper.

This is good for simple cases. But what if the code within the critical section, that is, between **`lock()`** and **`unlock()`**, throws an exception?

We would have to wrap our code in **`try...catch`**, which is not very convenient:

```
try {

    mutex.lock()

    counter++

}

finally {

    mutex.unlock()

}
```

However, if we omit the **`finally`** block, our lock will never be released and it will block all other coroutines from proceeding and creating a deadlock.

Exactly for this purpose, Kotlin also introduces **`withLock()`**:

```
mutex.withLock {

    counter++

}
```

Notice how much more concise this syntax is compared with the previous example.

Sidekick channel

The **Sidekick channel** design pattern allows us to offload some work from our main worker to a *back worker*.

Up until now, we've only discussed the use of **`select`** as a *receiver*. But we can also use **`select`** to *send* items to another channel. Let's look at the following example.

First, we'll declare **`batman`** as an actor coroutine that processes 10 messages per second:

```
val batman = actor<String> {

    for (c in channel) {

        println("Batman is beating some sense into $c")

        delay(100)

    }

}
```

Next, we'll declare **`robin`** as another actor coroutine that is a bit slower and processes only four messages per second:

```
val robin = actor<String> {

    for (c in channel) {

        println("Robin is beating some sense into $c")

        delay(250)

    }

}
```

So, we have a superhero and his sidekick as two actors. Since the superhero is more experienced, it usually takes him less time to beat the villain he's facing.

But in some cases, he still has his hands full, so a sidekick needs to step in. We'll throw five villains at the pair with a few delays and see how they fare:

```
val epicFight = launch {

    for (villain in listOf("Jocker", "Bane", "Penguin",
      "Riddler", "Killer Croc")) {

        val result = select<Pair<String, String>> {

            batman.onSend(villain) {

                "Batman" to villain

            }

            robin.onSend(villain) {

                "Robin" to villain

            }

        }

        delay(90)

        println(result)

    }

}
```

Notice that the type of parameter for **`select`** refers to what is *returned* from the block and not what is being *sent* to the channels. That's the reason we use **`Pair<String, String>`** here.

This code prints the following:

```
> Batman is beating some sense into Jocker

> (Batman, Jocker)

> Robin is beating some sense into Bane
> (Robin, Bane)

> Batman is beating some sense into Penguin

> (Batman, Penguin)

> Batman is beating some sense into Riddler

> (Batman, Riddler)

> Robin is beating some sense into Killer Croc

> (Robin, Killer Croc)
```

Using a sidekick channel is a useful technique to provide fallback values. Consider using one in cases when you need to consume a consistent stream of data and cannot easily scale your consumers.

Summary

In this chapter, we covered various design patterns related to *concurrency* in Kotlin. Most of them are based on coroutines, channels, deferred values, or a combination of these building blocks.

Deferred values are used as placeholders for asynchronous values. The Barrier design pattern allows multiple asynchronous tasks to rendezvous before proceeding further. The Scheduler design pattern decouples the code of tasks from the way they are executed at runtime.

The Pipeline, Fan In, and Fan Out design patterns help us distribute the work and collect the results. Mutex helps us to control the number of tasks that are being executed at the same time. The Racing design pattern allows us to improve the responsiveness of our application. Finally, the Sidekick Channel design pattern offloads work onto a backup task in case the main task is not able to process the incoming events quickly enough.

All of these patterns should help you to manage the concurrency of your application in an efficient and extensible manner. In the next chapter, we'll discuss Kotlin's idioms and best practices, as well as some of the anti-patterns that emerged with the language.

Questions

1. What does it mean when we say that the `select` expression in Kotlin is *biased*?
2. When should you use a *mutex* instead of a *channel*?
3. Which of the concurrent design patterns could help you implement a MapReduce or divide-and-conquer algorithm efficiently?

Section 3: Practical Application of Design Patterns

In this section, you will apply your new knowledge of design patterns to implement a real-world application and learn some best practices and anti-patterns.

The section starts with a collection of best practices and things to avoid while developing applications using Kotlin. Then, in the following two chapters, we will build two microservices, first using a concurrent framework called Ktor, and in the last chapter using a reactive framework called Vert.x.

We'll also use this opportunity to examine how design patterns we've seen in the previous chapters come into play in real-world applications.

This section comprises the following chapters:

Chapter 9: Idioms and Anti-Patterns

In the previous chapters, we discussed the different aspects of the Kotlin programming language, the benefits of functional programming, and concurrent design patterns.

This chapter discusses the best and worst practices in Kotlin. You'll learn what idiomatic Kotlin code should look like and which patterns to avoid. This chapter contains a collection of best practices spanning those different topics.

In this chapter, we will cover the following topics:

- Using the scope functions
- Type checks and casts
- An alternative to the try-with-resources statement
- Inline functions
- Implementing algebraic data types
- Reified generics
- Using constants efficiently
- Constructor overload
- Dealing with nulls
- Making asynchronicity explicit
- Validating input
- Preferring sealed classes over enums

After completing this chapter, you should be able to write more readable and maintainable Kotlin code, as well as avoid some common pitfalls.

Technical requirements

In addition to the requirements from the previous chapters, you will also need a **Gradle**-enabled **Kotlin** project to be able to add the required dependencies.

You can find the source code for this chapter here: https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter09.

Using the scope functions

Kotlin has the concept of **scoping functions**, which are available on any object and can replace the need to write repetitive code. Among other benefits, these scoping functions help us simplify single-expression functions. They are considered higher-order functions since each scoping function receives a lambda expression as an argument. In this section, we'll discuss all the necessary functions and execute their code blocks using objects as their *scope*. In this section, we'll use the terms *scope* and *context object* interchangeably to describe the objects that those functions operate on.

Let function

We can use the **`let()`** function to invoke a function on a nullable object, but only if the object is not null.

Let's take, as an example, the following map of quotes (we discussed this in *Chapter 1*, *Getting Started with Kotlin*):

```
val clintEastwoodQuotes = mapOf(
    "The Good, The Bad, The Ugly" to "Every gun makes its       own
tune.",
    "A Fistful Of Dollars" to "My mistake: four coffins."
)
```

Now, let's fetch a quote from a movie that may not exist in the collection and print it, but only if it's not null:

```
val quote = clintEastwoodQuotes["Unforgiven"]
if (quote != null) {
    println(quote)
}
```

The same code can we rewritten using the **`let`** scoping function:

```
clintEastwoodQuotes["Unforgiven"]?.let {
    println(it)
}
```

One common mistake is forgetting to use the safe navigation operator before **`let`**, because **`let()`** by itself also works on nulls:

```
clintEastwoodQuotes["Unforgiven"].let {
    println(it)
}
```

This code will print **`null`** to the console. Make sure that you don't forget the question mark (**`?`**) when you use **`let()`** for null checks.

Apply function

We have discussed **`apply()`** in previous chapters. It returns the same object it operates on and sets the context to **`this`**. You can use **`apply()`** if you need to initialize a mutable object.

Think of how many times you had to create a class that has an empty constructor, and then call a lot of setters, one after another. Let's look at the following class as an example. This may be a class that comes from a library, for example:

```
class JamesBond {

    lateinit var name: String

    lateinit var movie: String

    lateinit var alsoStarring: String

}
```

When we need to create a new instance of such a class, we could do so in a procedural manner:

```
val agent = JamesBond()

agent.name = "Sean Connery"

agent.movie = "Dr. No"
```

Alternatively, we can only set **`name`** and **`movie`**, and leave **`alsoStarring`** blank, using the **`apply()`** function:

```
val `007` = JamesBond().apply {
    this.name = "Sean Connery"

    this.movie = "Dr. No"

}

println(`007`.name)
```

Since the context of the block is set to **`this`**, we can simplify the preceding code even further:

```
val `007` = JamesBond().apply {

    name = "Sean Connery"

    movie = "Dr. No"

}
```

Using the **`apply()`** function is especially good when you're working with Java classes that usually have a lot of setters and a default empty constructor.

Also function

As we mentioned in the introduction to this section, single-expression functions are very nice and concise. Let's look at the following simple function, which multiplies two numbers:

```
fun multiply(a: Int, b: Int): Int = a * b
```

But often, you have a single-statement function that also needs to, for example, write to a log or have another side effect. To achieve this, we could rewrite our function in the following way:

```
fun multiply(a: Int, b: Int): Int {

    val c = a * b
    println(c)
    return c
}
```

We had to make our function much more verbose here and introduce another variable. Let's see how we can use the **`also()`** function instead:

```
fun multiply(a: Int, b: Int): Int =
    (a * b).also { println(it) }
```

This function will assign the results of the expression to **`it`** and return the result of the expression. The **`also()`** function is also useful when you want to have a side effect on a chain of calls:

```
val l = (1..100).toList()
l.filter{ it % 2 == 0 }
    // Prints, but doesn't mutate the collection
    .also { println(it) }
    .map { it * it }
```

Here, you can see that we can continue our chain of calls with a **`map()`** function, even though we used the **`also()`** function to print each element of a list.

Run function

The **`run()`** function is very similar to the **`let()`** function, but it sets the context of the block to **`this`** instead of using **`it`**.

Let's look at an example to understand this better:

```
val justAString = "string"
val n = justAString.run {

    this.length

}
```

In this example, **`this`** is set to reference the **`justAString`** variable.

Usually, **`this`** could be omitted, so the code will look as follows:

```
val n = justAString.run {

    length

}
```

The **`run()`** function is mostly useful when you plan to initialize an object, much like the **`apply()`** function we discussed earlier. However, instead of returning the object itself, like **`apply()`** does, you usually like to return the result of some computation:

```
val lowerCaseName = JamesBond().run {

    name = "ROGER MOORE"

    movie = "THE MAN WITH THE GOLDEN GUN"

    name.toLowerCase() // <= Not JamesBond type

}

println(lowerCaseName)
```

The preceding code prints the following output:

```
> roger moore
```

Here, the object was initialized with **`"ROGER MOORE"`**. Note that here, we operated on the **`JamesBond`** object, but our return value was a **`String`**.

With function

Unlike the other four scoping functions, **`with()`** is not an extension function. This means you cannot do the following:

```
"scope".with { ... }
```

Instead, **`with()`** receives the object you want to scope as an argument:

```
with("scope") {

    println(this.length) // "this" set to the argument of
                         // with()

}
```

And as usual, we can omit **`this`**:

```
with("scope") {

    length

}
```

Just like **`run()`** and **`let()`**, you can return any result from **`with()`**.

In this section, we learned how the various scope functions can help reduce the amount of boilerplate code by defining a code block to be executed on the object. In the next section, we'll see how Kotlin also allows us to write fewer instance checks than other languages.

Type checks and casts

While writing your code, you may often be inclined to check what type your object is using, **`is`**, and cast it using **`as`**. As an example, let's imagine we're building a system for superheroes. Each superhero has their own set of methods:

```
interface Superhero
```

```
class Batman : Superhero {
    fun callRobin() {
        println("To the Bat-pole, Robin!")
    }
}
class Superman : Superhero {
    fun fly() {
        println("Up, up and away!")
    }
}
```

There's also a function where a superhero tries to invoke their superpower:

```
fun doCoolStuff(s: Superhero) {
    if (s is Superman) {
        (s as Superman).fly()
    }
    else if (s is Batman) {
        (a as Batman).callRobin()
    }
}
```

But as you may know, Kotlin has smart casts, so implicit casting, in this case, is not needed. Let's rewrite this function using smart casts and see how they improve our code. All we need to do is remove the explicit casts from our code:

```
fun doCoolStuff(s: Superhero) {
    if (s is Superman) {
        s.fly()
    }
    else if (s is Batman) {
        s.callRobin()
    }
}
```

Moreover, in most cases, using **`when()`** while smart casting produces cleaner code:

```
fun doCoolStuff(s : Superhero) {
    when(s) {
        is Superman -> s.fly()
        is Batman -> s.callRobin()
        else -> println("Unknown superhero")
    }
}
```

As a rule of thumb, you should avoid using casts and rely on smart casts most of the time:

```
// Superhero is clearly not a string
val superheroAsString = (s as String)
```

But if you absolutely must, there's also a safe cast operator:

```
val superheroAsString = (s as? String)
```

The **safe cast operator** will return null if the object cannot be cast, instead of throwing an exception.

```
}
```

An alternative to the try-with-resources statement

Java 7 added the notion of **`AutoCloseable`** and the try-with-resources statement.

This statement allows us to provide a set of resources that will be automatically closed once the code is done with them. So, there will be no more risk (or at least less risk) of forgetting to close a file.

Before Java 7, this was a total mess, as shown in the following code:

```
BufferedReader br = null; // Nulls are bad, we know that

try {

    br = new BufferedReader(new FileReader

      ("./src/main/kotlin/7_TryWithResource.kt "));

    System.out.println(br.readLine());

}

finally {

    if (br != null) { // Explicit check

        br.close(); // Boilerplate

    }

}
```

After **Java 7** was released, the preceding code could be written as follows:

```
try (BufferedReader br = new BufferedReader(new
  FileReader("/some/path"))) {

    System.out.println(br.readLine());

}
```

Kotlin doesn't support this syntax. Instead, the try-with-resource statement is replaced with the **`use()`** function:

```
val br = BufferedReader(FileReader("./src/main
  /kotlin/7_TryWithResource.kt"))

br.use {

    println(it.readLines())

}
```

An object must implement the **`Closeable`** interface for the **`use()`** function to be available. The **`Closeable`** object will be closed as soon as we exit the **`use{}`** block.

Inline functions

You can think of **`inline`** functions as instructions for the compiler to copy and paste your code. Each time the compiler sees a call to a function marked with the **`inline`** keyword, it will replace the call with the concrete function body.

It makes sense to use the **`inline`** function if it's a higher-order function that receives a lambda as one of its arguments. This is the most common use case where you would like to use **`inline`**.

Let's look at such a higher-order function and see what pseudocode the compiler will output.

First, here is the function definition:

```
inline fun logBeforeAfter(block: () -> String) {

    println("Before")
    println(block())
    println("After")
}
```

Here, we pass a lambda, or a **`block`**, to our function. This **`block`** simply returns the word **`"Inlining"`** as a **`String`**:

```
logBeforeAfter {
    "Inlining"
}
```

If you were to view the Java equivalent of the decompiled bytecode, you'd see that there's no call to our **`makesSense`** function at all. Instead, you'd see the following:

```
String var1 = "Before"; <- Inline function call
System.out.println(var1);
var1 = "Inlining";
System.out.println(var1);
var1 = "After";
System.out.println(var1);
```

Since the **`inline`** function is a copy/paste of your code, you shouldn't use it if you have more than a few lines of code. It would be more efficient to have it as a regular function. But if you have single-expression functions that receive a lambda, it makes sense to mark them with the **`inline`** keyword to optimize performance. In the end, it's a trade-off between the size of your application and its performance.

Implementing Algebraic Data Types

Algebraic Data Types, or **ATDs** for short, is a concept from functional programming and is very similar to the **Composite design pattern** we discussed in Chapter 3, *Understanding Structural Patterns*.

To understand how ADTs work and what their benefits are, let's discuss how we can implement a simple binary tree in Kotlin.

First, let's declare an interface for our tree. Since a tree data structure can contain any type of data, we can parameterize it with a type (`T`):

```
sealed interface Tree<out T>
```

The type is marked with an `out` keyword, which means that this type is *covariant*. If you aren't familiar with this term, we'll cover it later, while implementing the interface.

The opposite of a covariant is a *contravariant*. Contravariant types should be marked using the `in` keyword.

We can also mark this interface with a `sealed` keyword. We saw this keyword applied to regular classes in Chapter 4, *Getting Familiar with Behavioral Patterns*, while discussing the **Visitor pattern**. But `sealed` interfaces are a relatively new feature and were introduced in **Kotlin 1.5**.

The meaning is the same, though: only the owner of the interface can implement it. This means that all the implementations of the interface are known at compile time.

Next, let's declare what an empty tree looks like:

```
object Empty : Tree<Nothing> {

    override fun toString() = "Empty"

}
```

Since all empty trees are the same, we declare it as an object. This is another use of the **Singleton design pattern**, which we discussed in Chapter 2, *Working with Creational Patterns*. We can also use **`Nothing`** as the type of an empty tree. This is a special class in Kotlin's object hierarchy.

IMPORTANT NOTE:

*There is some confusion between **`Any`**, which represents any class and is similar to **`Object`** in Java, and **`Nothing`**, which represents no class. We'll see why **`Any`** wouldn't work in this case later in this chapter.*

Next, let's define a non-empty node of a tree:

```
data class Node<T>(

    val value: T,

    val left: Tree<T> = Empty,

    val right: Tree<T> = Empty

) : Tree<T>
```

`Node` also implements the **`Tree`** interface, but it is a data class and not an object since every node is different. The type of the value of a **`Node`** is **`T`**, which means it can contain any type of value, but all the nodes in the same tree will contain the same type of value. This is the real power of generics.

A node also has two children, left and right, since it's a binary tree. By default, both of them are empty.

We can specify the default values for the children of a node thanks to the fact that the type is covariant and **`Empty`** is of the **`Nothing`** type. **`Nothing`** is at the bottom of the class hierarchy, while **`Any`** is at the very top.

When we declared the type of our **`Tree`** as **`out T`**, we meant that our **`Tree`** could contain values of type **`T`** or anything that inherits from that type.

Since `Nothing` is at the bottom of a class hierarchy, it *inherits* from all types.

Now that everything has been set, let's learn how to create a new instance of the tree we just defined:

```
val tree = Node(
    1,
    Empty,
    Node(
        2,
        Node(3)
    )
)
println(tree)
```

Here, we created a tree with **1** as the value of the root node and a right node with a value of **2**. The right node has a left child with a value of **3**. This is what our tree looks like:

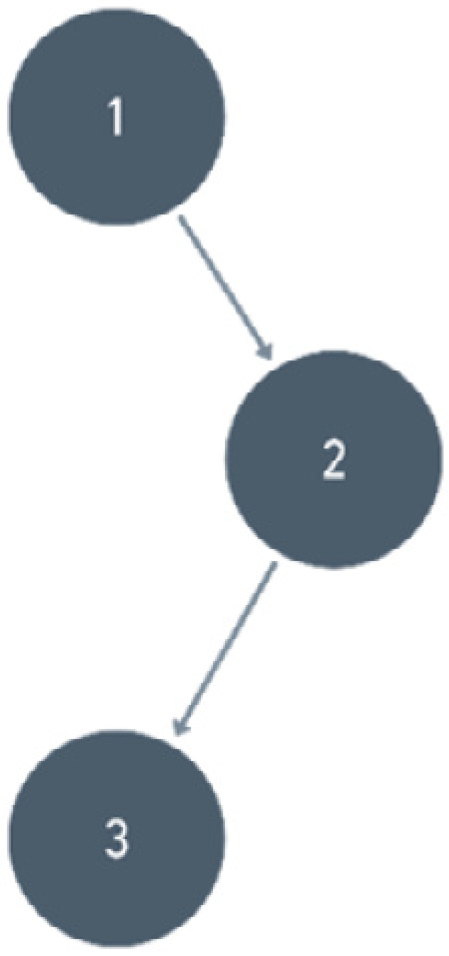


Figure 9.1 – Tree diagram

The preceding code outputs the following:

```
> Node(value=1, left=Empty, right=Node(value=2, left=Node(value=3,
left=Empty, right=Empty), right=Empty))
```

However, printing the tree in such a form is not very interesting. So, let's implement a function that will summarize all the nodes of a tree if it's numeric:

```
fun Tree<Int>.sum(): Long = when (this) {
    Empty -> 0
    is Node -> value + left.sum() + right.sum()
}
```

This is also called an **operation** on an ADT. This is an extension function that is declared only on trees that contain integers.

For each node, we check whether it's `Empty` or `Node`. That's the beauty of `sealed` classes and interfaces. Since the compiler knows that the `Tree` interface has exactly two implementations, we don't need an `else` block in our `when` expression.

If it's an `Empty` node, we use `0` as a neutral value. If it's not empty, then we sum its values with the left and right children.

This function is also another example of a recursive algorithm, which we discussed in *Chapter 5*, *Introducing Functional Programming*.

Now, let's discuss another topic related to generics in Kotlin.

Reified generics

Previously in this chapter, we mentioned `inline` functions. Since `inline` functions are copied, we can get rid of one of the major JVM limitations: **type erasure**. After all, inside the function, we know exactly what type we're getting.

Let's look at the following example. We would like to create a generic function that will receive a `Number` (`Number` can either be `Int` or `Long`), but will only print it if it's of the same type as the function type.

We'll start with a naïve implementation, simply trying the instance check on the type directly:

```
fun <T> printIfSameType(a: Number) {
    if (a is T) { // <== Error
        println(a)
    }
}
```

However, this code won't compile and we'll get the following error:

```
> Cannot check for instance of erased type: T
```

What we usually do in Java, in this case, is pass the class as an argument. We can try a similar approach in Kotlin. If you've worked with Android before, you'll recognize this pattern immediately, since it's used a lot in the standard library:

```
fun <T : Number> printIfSameType(clazz: KClass<T>, a:
  Number) {
    if (clazz.isInstance(a)) {
        println("Yes")
    } else {
        println("No")
    }
}
```

We can check that the code works correctly by running the following lines:

```
printIfSameType(Int::class, 1) // Prints yes, as 1 is Int
printIfSameType(Int::class, 2L) // Prints no, as 2 is Long
printIfSameType(Long::class, 3L) // Prints yes, as 3 is Long
```

This code works but has a few downsides:

- We cannot use the **`is`** operator and must use the **`isInstance()`** function instead.
- We must pass the correct class; that is, **`clazz: KClass<T>`**.

This code could be improved by using a **`reified`** type:

```
inline fun <reified T : Number> printIfSameReified(a:   Number) {

    if (a is T) {

        println("Yes")

    } else {

        println("No")

    }

}
```

This function works the same as the previous one but doesn't need a class as input to work. A function that uses a **`reified`** type must be declared as **`inline`**. This is due to type erasure on the JVM.

We can test that our code still works as expected:

```
printIfSameReified<Int>(1) // Prints yes, as 1 is Int

printIfSameReified<Int>(2L) // Prints no, as 2 is Long

printIfSameReified<Long>(3L) // Prints yes, as 3 is Long
```

Notice that now, we specify the type that the function operates on, such as **`Int`** or **`Long`**, between *angular brackets*, instead of passing a class to it as an argument. We get the following benefits from using the **`reified`** functions:

- A clear method signature, without the need to pass a class as an argument.

- The ability to use the **`is`** construct inside the function.
- It's type-inference friendly, which means that if the *type* parameter can be inferred by the compiler, it can be completely omitted.

Of course, the same rules for the regular **`inline`** functions apply here. This code would be replicated, so it shouldn't be too large.

Now, let's consider another case for **`reified`** types – **function overloading**. We'll try to define two functions with the same name that differ only in terms of the types they operate on:

```
fun printList(list: List<Int>) {

    println("This is a list of Ints")

    println(list)

}

fun printList(list: List<Long>) {

    println("This is a list of Longs")

    println(list)

}
```

This won't compile because there's a platform declaration clash. Both have the same signature in terms of JVM: **`printList(list: List)`**. This is because types are erased during compilation.

But with **`reified`**, we can achieve this easily:

```
inline fun <reified T : Any> printList(list: List<T>) {

    when {

        1 is T -> println("This is a list of Ints")

        1L is T -> println("This is a list of Longs")
        else -> println("This is a list of something else")
    }
    println(list)
}
```

Since the entire function is *inlined*, we can check the actual type of the list and output the correct result.

Using constants efficiently

Since everything in Java is an object (unless it's a primitive type), we're used to putting all the constants inside our objects as static members.

And since Kotlin has **`companion`** objects, we usually try putting them there:

```
class Spock {
    companion object {
        val SENSE_OF_HUMOR = "None"
    }
}
```

This will work, but you should remember that **`companion object`** is an object, after all.

So, this will be translated into the following code, more or less:

```
public class Spock {
    private static final String SENSE_OF_HUMOR = "None";
    public String getSENSE_OF_HUMOR() {
        return Spock.SENSE_OF_HUMOR;
    }
    ...
}
```

In this example, the Kotlin compiler generates a getter for our constant, which adds another level of indirection.

If we look at the code using the constant, we'll see the following:

```
String var0 = Spock.Companion.getSENSE_OF_HUMOR();
System.out.println(var0);
```

We can invoke a method to get the constant value, which is not very efficient.

Now, let's mark this value as constant and see how the code produced by the compiler changes:

```
class Spock {
    companion object {
        const val SENSE_OF_HUMOR = "None"
    }
}
```

Here are the bytecode changes:

```
public class Spock {
   public static final String SENSE_OF_HUMOR = "None";
   ...
}
```

And here is the call:

```
String var1 = "None";
System.out.println(var1);
```

Notice that there's no reference for our **`Spock`** class in the code anymore. The compiler has already *inlined* its value for us. After all, it's constant, so it will never change and can be safely *inlined*.

If all you need is a constant, you can also set it up outside of any class:

```
const val SPOCK_SENSE_OF_HUMOR = "NONE"
```

And if you need namespacing, you can wrap it in an object:

```
object SensesOfHumor {

    const val SPOCK = "None"

}
```

Now that we've learned how to use constants more efficiently, let's learn how to work with constructors in an idiomatic manner.

Constructor overload

In Java, we're used to having overloaded constructors. For example, let's look at the following Java class, which requires the **`a`** parameter and defaults the value of **`b`** to **`1`**:

```
class User {

    private final String name;

    private final boolean resetPassword;

    public User(String name) {

        this(name, true);

    }

    public User(String name, boolean resetPassword) {

        this.name = name;
        this.resetPassword = resetPassword;
    }
}
```

We can simulate the same behavior in Kotlin by defining multiple constructors using the **`constructor`** keyword:

```
class User(val name: String, val resetPassword: Boolean) {
    constructor(name: String) : this(name, true)
}
```

The secondary constructor, as defined in the class body, will invoke the primary constructor, providing **`1`** as the default value for the second argument.

However, it's usually better to have default parameter values and named arguments instead:

```
class User(val name: String, val resetPassword: Boolean =   true)
```

Note that all the secondary constructors must delegate to the primary constructor using the **`this`** keyword. The only exception is when you have a default primary constructor:

```
class User {
    val resetPassword: Boolean
    val name: String
    constructor(name: String, resetPassword: Boolean =
      true) {
        this.name = name
        this.resetPassword = resetPassword
    }
}
```

Next, let's discuss how to efficiently handle nulls in Kotlin code.

Dealing with nulls

Nulls are unavoidable, especially if you work with Java libraries or get data from a database. We've already discussed that there are different ways to check whether a variable contains `null` in Kotlin; for example:

```
// Will return "String" half of the time and null the other

// half

val stringOrNull: String? = if (Random.nextBoolean())

  "String" else null

// Java-way check

if (stringOrNull != null) {

    println(stringOrNull.length)

}
```

We could rewrite this code using the `Elvis` operator (`?:`):

```
val alwaysLength = stringOrNull?.length ?: 0
```

If the length is not `null`, this operator will return its value. Otherwise, it will return the default value we supplied, which is `0` in this case.

If you have a nested object, you can chain those checks. For example, let's have a `Response` object that contains a `Profile`, which, in turn, contains the first name and last name fields, which can be nullable:

```
data class Response(

    val profile: UserProfile?
)
data class UserProfile(
    val firstName: String?,
    val lastName: String?
)
```

This chaining will look like this:

```
val response: Response? = Response(UserProfile(null, null))
println(response?.profile?.firstName?.length)
```

If any of the fields in the chain are null, our code won't crash. Instead, it will print **`null`**.

Finally, you can use the **`let()`** block for null checks, as we briefly mentioned in the *Using the scope functions* section. The same code, but using the **`let()`** function instead, will look like this:

```
println(response?.let {
    it.profile?.let {
        it.firstName?.length
    }
})
```

If you want to get rid of **`it`** everywhere, you can use another scoping function, **`run()`**:

```
println(response?.run {
    profile?.run {
        firstName?.length
    }
})
```

Try to avoid using the unsafe `!!` null operator in production code:

```
println(json!!.User!!.firstName!!.length)
```

This will result in **`KotlinNullPointerException`**. However, during tests, the **`!!`** operator may prove useful, as it will help you spot null-safety issues faster.

Making asynchronicity explicit

As you saw in the previous chapter, it is very easy to create an asynchronous function in Kotlin. Here is an example:

```
fun CoroutineScope.getResult() = async {

   delay(100)

   "OK"

}
```

However, this asynchronicity may be an unexpected behavior for the user of the function, as they may expect a simple value.

What do you think the following code prints?

```
println("${getResult()}")
```

For the user, the preceding code somewhat unexpectedly prints the following instead of **`"OK"`**:

```
> Name: DeferredCoroutine{Active}@...
```

Of course, if you have read *Chapter 6*, *Threads and Coroutines*, you will know that what's missing here is the **`await()`** function:

```
println("${getResult().await()}")
```

But it would have been a lot more obvious if we'd named our function accordingly, by adding an **`async`** suffix:

```
fun CoroutineScope.getResultAsync() = async {
   delay(100)
   "OK"
}
```

Kotlin's convention is to distinguish asynchronous functions from regular ones by adding **`Async`** to the end of the function's name. If you're working with **IntelliJ IDEA**, it will even suggest you that rename it.

Now, let's talk about some built-in functions for validating the user's input.

Validating input

Input validation is a necessary but very tedious task. *How many times did you have to write code like the following?*

```
fun setCapacity(cap: Int) {
    if (cap < 0) {
        throw IllegalArgumentException()
    }
    ...
}
```

Instead, you can check arguments with the **`require()`** function:

```
fun setCapacity(cap: Int) {
    require(cap > 0)
}
```

This makes the code a lot more fluent. You can use **`require()`** to check for nulls:

```
fun printNameLength(p: Profile) {
    require(p.firstName != null)
}
```

But there's also **`requireNotNull()`** for that:

```
fun printNameLength(p: Profile) {
    requireNotNull(p.firstName)
}
```

Use **`check()`** to validate the state of your object. This is useful when you are providing an object that the user may not have set up correctly:

```
class HttpClient {
    var body: String? = null
    var url: String = ""
    fun postRequest() {
        check(body != null) {
            "Body must be set in POST requests"
        }
    }
}
```

And again, there's a shortcut for this as well: **`checkNotNull()`**.

The difference between the **`require()`** and **`check()`** functions is that **`require()`** throws **`IllegalArgumentException`**, implying that the input that was provided to the function was incorrect. On the other hand, **`check()`**

throws **`IllegalStateException`**, which means that the state of the object is corrupted.

Consider using functions such as **`require()`** and **`check()`** to improve the readability of your code.

Finally, let's discuss how to efficiently represent different states in Kotlin.

Preferring sealed classes over enums

Coming from Java, you may be tempted to overload your **`enum`** with functionality.

For example, let's say you build an application that allows users to order a pizza and track its status. We can use the following code for this:

```
// Java-like code that uses enum to represent State

enum class PizzaOrderStatus {

    ORDER_RECEIVED, PIZZA_BEING_MADE,
OUT_FOR_DELIVERY,      COMPLETED;

    fun nextStatus(): PizzaOrderStatus {

        return when (this) {

            ORDER_RECEIVED -> PIZZA_BEING_MADE

            PIZZA_BEING_MADE -> OUT_FOR_DELIVERY

            OUT_FOR_DELIVERY -> COMPLETED

            COMPLETED -> COMPLETED

        }

    }

}
```

Alternatively, you can use the **`sealed`** class:

```
sealed class PizzaOrderStatus(protected val orderId: Int) {

    abstract fun nextStatus(): PizzaOrderStatus

}

class OrderReceived(orderId: Int) :

  PizzaOrderStatus(orderId) {

    override fun nextStatus() = PizzaBeingMade(orderId)

}

class PizzaBeingMade(orderId: Int) :

  PizzaOrderStatus(orderId) {

    override fun nextStatus() = OutForDelivery(orderId)

}

class OutForDelivery(orderId: Int) :

  PizzaOrderStatus(orderId) {

    override fun nextStatus() = Completed(orderId)

}

class Completed(orderId: Int) : PizzaOrderStatus(orderId) {

    override fun nextStatus() = this

}
```

Here, we created a separate class for each object state, extending the **`PizzaOrderStatus`** sealed class.

The benefit of this approach is that we can now store the state, along with its **`status`**, more easily. In our example, we can store the ID of the order:

```
var status: PizzaOrderStatus = OrderReceived(123)

while (status !is Completed) {
    status = when (status) {
        is OrderReceived -> status.nextStatus()
        is PizzaBeingMade -> status.nextStatus()
        is OutForDelivery -> status.nextStatus()
        is Completed -> status
    }
}
```

In general, **`sealed`** classes are good if you want to have data associated with a state, and you should prefer them over enums.

Summary

In this chapter, we reviewed the best practices in Kotlin, as well as some of the caveats of the language. Now, you should be able to write more idiomatic code that is also performant and maintainable.

You should make use of the scoping functions where necessary, but make sure not to overuse them as they may make the code confusing, especially for those newer to the language.

Be sure to handle nulls and type casts correctly, with **`let()`**, the **`Elvis`** operator, and the smart casts that the language provides. Finally, generics and **`sealed`** classes and interfaces are powerful tools that help describe complex relationships and behaviors between different classes.

In the next chapter, we'll put those skills to use by writing a real-life microservice Reactive design pattern.

Questions

1. What is the alternative to Java's try-with-resources in Kotlin?
2. What are the different options for handling nulls in Kotlin?
3. Which problem can be solved by reified generics?

Chapter 10: Concurrent Microservices with Ktor

In the previous chapter, we explored how we should write idiomatic Kotlin code that will be readable and maintainable, as well as performant.

In this chapter, we'll put the skills we've learned so far to use by building a microservice using the **Ktor framework**. We also want this microservice to be reactive and to be as close to real life as possible. For that, we'll use the Ktor framework, the benefits of which we'll list in the first section of this chapter.

In this chapter, we will cover the following topics:

- Getting started with Ktor
- Routing requests
- Testing the service
- Modularizing the application
- Connecting to a database
- Creating new entities
- Making the test consistent
- Fetching entities
- Organizing routes in Ktor
- Achieving concurrency in Ktor

By the end of this chapter, you'll have a microservice written in Kotlin that is well tested and can read data from a PostgreSQL database and store data in it.

Technical requirements

This is what you'll need to get started:

- **JDK 11** or later
- IntelliJ IDEA
- **Gradle 6.8** or later
- **PostgreSQL 14** or later

This chapter will assume that you have `PostgreSQL` already installed and that you have the basic knowledge for working with it. If you don't, please refer to the official documentation: https://www.postgresql.org/docs/14/tutorial-install.html.

You can find the source code for this chapter here: https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter10.

Getting started with Ktor

You're probably tired of creating to-do or shopping lists.

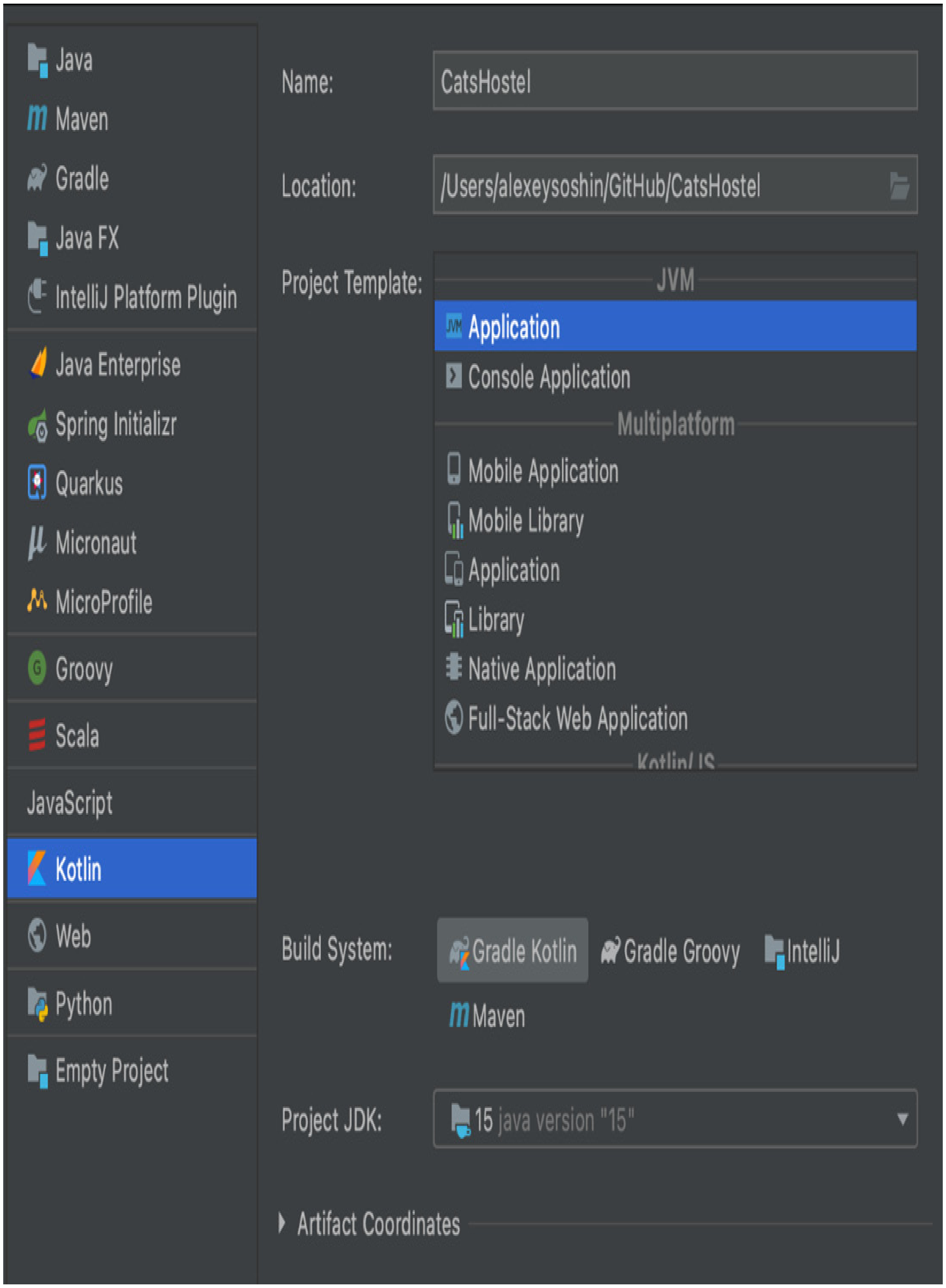
So, instead, in this chapter, the microservice will be for a `cat shelter`. The microservice should be able to do the following:

- Supply an endpoint we can ping to check whether the service is up and running
- List the cats currently in the shelter
- Provide us with a means to add new cats

The framework we'll be using for our microservice in this chapter is called **Ktor**. It's a concurrent framework that's developed and maintained by the creators of the Kotlin programming language.

Let's start by creating a new Kotlin Gradle project:

1. From your IntelliJ IDEA, select **File** | **New** | **Project** and choose **Kotlin** from **New Project** and **Gradle Kotlin** as your **Build System**.
2. Give your project a descriptive name – `CatsHostel`, in my case – and choose **Project JDK** (in this case, we are using **JDK 15**):



Name: CatsHostel

Location: /Users/alexeysoshin/GitHub/CatsHostel

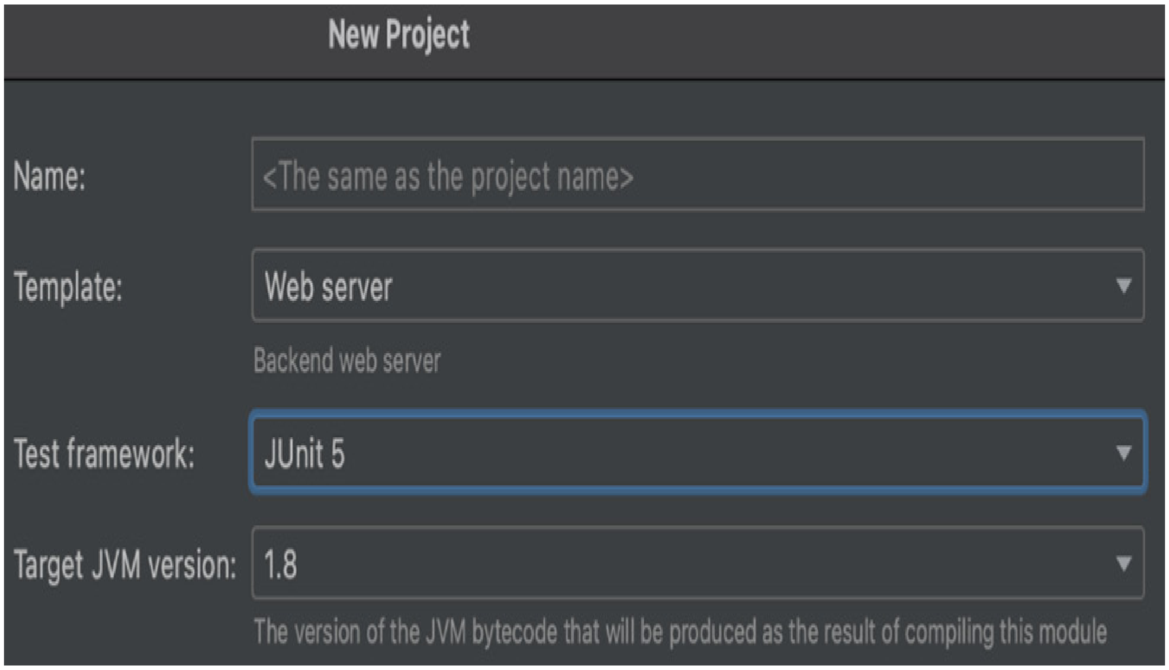
Project Template: Application

Build System: Gradle Kotlin

Project JDK: 15 java version "15"

Figure 10.1 – Selecting the Project JDK type

3. On the next screen, select **JUnit 5** as your **Test framework** and set **Target JVM version** to **1.8**. Then, click **Finish**:


New Project

Name: <The same as the project name>

Template: Web server

Backend web server

Test framework: JUnit 5

Target JVM version: 1.8

The version of the JVM bytecode that will be produced as the result of compiling this module

Figure 10.2 – Selecting Test framework and Target JVM version

4. Now, you should see the following structure:

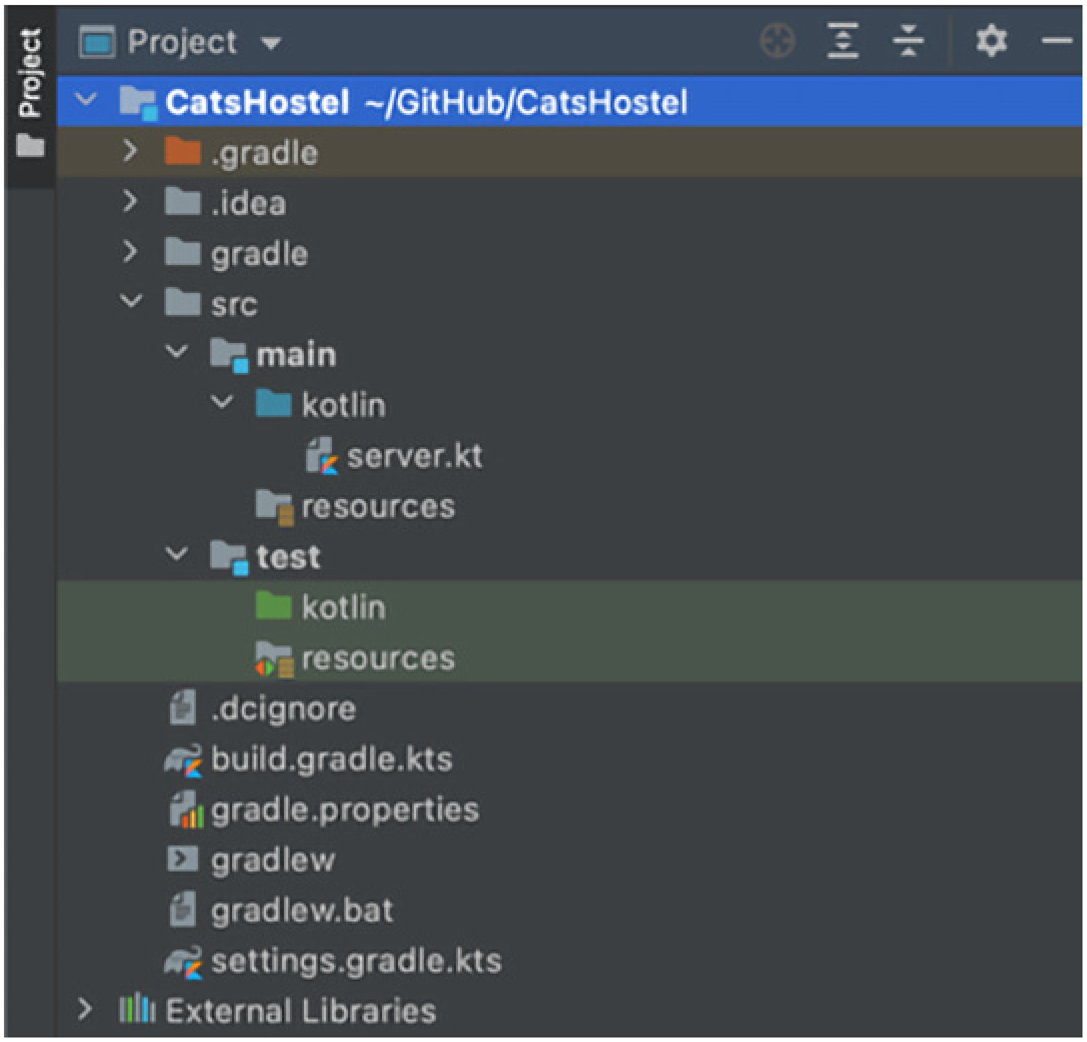


Figure 10.3 – Project structure

Next, let's open `build.gradle.kts`. This file controls how your project is built, its dependencies, and the libraries that the project is going to use. Depending on the version of your IntelliJ IDEA, the file's contents may differ a bit, but the general structure stays the same.

The `.kts` extension means that the configuration file for our Kotlin project is written in Kotlin, or to be precise, in **Kotlin Script**. Now, we would like to start using the Ktor framework to write our server. To do that, let's find our `dependencies` block, which should look like this:

```
dependencies {

    implementation(...)

    testImplementation("org.junit.jupiter:junit-
jupiter-        api:5.6.0")

    testRuntimeOnly("org.junit.jupiter:junit-
jupiter-        engine:5.6.0")


}
```

The preceding code mentions all the libraries that your project will be using. The **`implementation()`** configuration means that the library will be used at all times. The **`testImplementation()`** configuration means that the library will only be used during tests.

Now, let's take a look at how a library is defined in the following example:

```
"org.junit.jupiter:junit-jupiter-api:5.6.0"
```

This is a regular string that has been separated into three parts, as follows:

```
"group:name:version"
```

The **`group`** and **`name`** strings identify the library; the **`version`** configuration should be self-explanatory.

Now, let's modify the **`dependencies`** block, as follows:

```
val ktorVersion = "1.6.0"

dependencies {

    implementation("io.ktor:ktor-server-

        netty:$ktorVersion")

    ...

}
```

Since the files with **`.kts`** extensions are Kotlin files, we can use regular Kotlin syntax in them. In this case, we are using values and string interpolation to extract the version of our library.

The latest version of Ktor to date is **1.6.4**, but when you read this book, it will be greater than this. You can find the latest version here: https://ktor.io/.

As a general rule, all Ktor libraries should be the same version and that's when the variable becomes useful.

TIP:

*If you have followed the steps from the beginning of this section, you should have a file called **`server.kt`** in the **`src/main/kotlin`** folder in your project. If you don't, create one now.*

Now, let's add the following content to the **`server.kt`** file:

```
fun main() {
    embeddedServer(Netty, port = 8080) {
        routing {
            get("/") {
                call.respondText("OK")
            }
        }
    }.start(wait = true)
    println("open http://localhost:8080")
}
```

That's all the code we need to write to start a web server that will respond with **`OK`** when you open **`http://localhost:8080`** in your browser.

Now, let's understand what happens here:

- To interact with the request and return a response, we can use the `call` object, also known as the **context**. This object provides all the convenient methods for parsing requests and returning responses in different formats, and we'll see the different methods available for it throughout this chapter.
- The `embeddedServer()` function is an implementation of the Builder pattern, which we discussed in *Chapter 2*, *Working with Creational Patterns*. It allows us to configure our server. Most of the arguments have the same defaults. We override `port` to `8080` just for convenience.
- We specify the `wait` argument to be `true` so that our server will wait for incoming requests.
- The only mandatory argument to the `embeddedServer` function is the server engine. In our example, we use `Netty`, which is a very well-known JVM library, but there are others as well. The most interesting of them is `CIO`, which was developed by JetBrains themselves.

Now, let's understand what `CIO` and `Netty` are. They are both **Factory** patterns that create the actual instance of our server when invoked. This is a really interesting combination of different design patterns in one place to create a very flexible and extensible architecture.

To switch to using `CIO`, all we need to do is add a new dependency:

```
dependencies {
    ...
    implementation("io.ktor:ktor-server-cio:$ktorVersion")
    ...
}
```

Then, we need to pass another server engine, **`CIO`**, to our **`embeddedServer`** function:

```
embeddedServer(CIO, port = 8080) {
    ...
}.start(wait = true)
```

Notice that we didn't have to change anything else in our code when we switched the server engine. That is because **`embeddedServer()`** uses the Bridge design pattern to make components interchangeable.

Now that our server has been started, let's investigate how we define different responses for each request to the server.

Routing requests

Now, let's take a look at the **`routing`** block:

```
routing {
    get("/") {
        call.respondText("OK")
    }
}
```

This block describes all the URLs that will be handled by our server. In this case, we only handle the root URL. When that URL is requested, a text response, **`OK`**, will be returned to the user.

The following code returns a text response. Now, let's see how we can return a JSON response instead:

```
get("/status") {

    call.respond(mapOf("status" to "OK"))

}
```

Instead of using the **`respondText()`** method, we'll use **`respond()`**, which receives an object instead of a string. In our example, we're passing a map of strings to the **`respond()`** function. If we run this code, though, we'll get an exception.

This is because, by default, objects are not serialized into JSON. Multiple libraries can do this for us. In this example, we'll use the **`kotlinx-serialization`** library. Let's start by adding it to our dependencies:

```
dependencies {

    ...

    implementation("org.jetbrains.kotlinx:kotlinx-        serializa
tion-json-jvm:1.3.0")

    ...

}
```

Next, we need to add the following lines before our **`routing`** block:

```
install(ContentNegotiation) {

    json()

}
```

Now, if we run our code again, it will output this on our browser:

```
> {"status":"OK"}
```

We've just created our first route, which returns an object serialized as JSON. Now, we can check whether our application works by opening **`http://localhost:8080/status`** in our browser. But that is a bit cumbersome. In the next section, we'll learn how to write a test for the **`/status`** endpoint.

Testing the service

To write our first test, let's create a new file called **`ServerTest.kt`** under the **`src/test/kotlin`** directory.

Now, let's add a new dependency:

```
dependencies {

    ...

    testImplementation("io.ktor:ktor-server-

        tests:$ktorVersion")

}
```

Next, let's add the following contents to our **`ServerTest.kt`** file:

```
internal class ServerTest {

    @Test

    fun testStatus() {

        withTestApplication {

            val response = handleRequest(HttpMethod.Get,
                "/status").response

            assertEquals(HttpStatusCode.OK,
                response.status())
            assertEquals("""{"status": "OK"}""",
                response.content)
        }
    }
}
```

Tests in Kotlin are grouped into classes, and each test is a method in the class, which is marked with the **`@Test`** annotation.

Inside the **`test`** method, we start a test server, issue a **`GET`** request to the **`/status`** endpoint, and check that the endpoint responds with a correct status code and JSON body.

If you run this test now, though, it will fail, because we haven't started our server yet. To do so, we'll need to refactor it a bit, which we'll do in the next section.

Modularizing the application

So far, our server has been started from the **`main()`** function. This was simple to set up, but this doesn't allow us to test our application.

In Ktor, the code is usually organized into modules. Let's rewrite our **`main`** function, as follows:

```
fun main() {
    embeddedServer(
        CIO,
        port = 8080,
        module = Application::mainModule
    ).start(wait = true)
}
```

Here, instead of providing the logic of our server within a block, we specified a module that will contain all the configurations for our server.

This module is defined as an extension function on the **`Application`** object:

```
fun Application.mainModule() {

    install(ContentNegotiation) {

        json()

    }

    routing {

        get("/status") {

            call.respond(mapOf("status" to "OK"))

        }

    }

    println("open http://localhost:8080")

}
```

As you can see, the content of this function is the same as that of the block that we passed to our **`embeddedService`** function earlier.

Now, all we need to do is go back to our test and specify which module we would like to test:

```
@Test

fun testStatus() {

    withTestApplication(Application::mainModule) {

        ...

    }
}
```

If you run this test now, it should pass, because our server has started properly in test mode.

So far, we've only dealt with the infrastructure of our service; we haven't touched on its business logic: *managing cats*. To do so, we'll need a database. In the next section, we'll discuss how Ktor solves this problem using the Exposed library.

Connecting to a database

To store and retrieve cats, we'll need to connect to a database. We'll use PostgreSQL for that purpose, although using another SQL database won't be any different.

First, we'll need a new library to connect to the database. We'll use the Exposed library, which is also developed by JetBrains.

Let's add the following dependency to our **`build.gradle.kts`** file:

```
dependencies {

    implementation("org.jetbrains.exposed:exposed:0.17.14")

    implementation("org.postgresql:postgresql:42.2.24")

    ...

}
```

Once the libraries are in place, we need to connect to them. To do that, let's create a new file called **`DB.kt`** under **`/src/main/kotlin`** with the following contents:

```
object DB {
    private val host=System.getenv("DB_HOST")?:"localhost"

    private val port =
        System.getenv("DB_PORT")?.toIntOrNull() ?: 5432

    private val dbName = System.getenv("DB_NAME") ?:
        "cats_db"

    private val dbUser = System.getenv("DB_USER") ?:
        "cats_admin"

    private val dbPassword = System.getenv("DB_PASSWORD")
        ?: "abcd1234"

  fun connect() =
Database.connect(      "jdbc:postgresql://$host:$port/$dbName",
  driver = "org.postgresql.Driver",      user =
dbUser,      password = dbPassword

  )

}
```

Since our application needs exactly one instance of a database, the `DB` object can use the Singleton pattern, which we discussed in Chapter 2, *Working with Creational Patterns*. For that, we will use the `object` keyword.

Then, for each of the variables that we need to connect to the database, we will attempt to read them from our environment. If the `environment` variable is not set, we will use a default value using the **Elvis** operator.

TIP:

Creating a database and a user is beyond the scope of this book, but you can refer to the official documentation for this, at https://www.postgresql.org/docs/14/app-createuser.html and https://www.postgresql.org/docs/14/app-createdb.html.

Alternatively, you can simply run the following two commands in your command line:

```
$ createuser cats_admin -W –d
$ createdb cats_db -U cats_admin
```

The first command creates a database user called **`cats_admin`** and asks you to specify a password for this user. Our application will use this **`cats_admin`** user to interact with the database. The second command creates a database called **`cats_db`** that belongs to the **`cats_admin`** user. Now that our database has been created, all we need to do is create a table that will store our cats in it.

For that, let's define another Singleton object in our **`DB.kt`** file that will represent a table:

```
object CatsTable : IntIdTable() {
    val name = varchar("name", 20).uniqueIndex()
    val age = integer("age").default(0)
}
```

Let's understand what the preceding definition means:

- **`IntIdTable`** means that we want to create a table with a primary key of the **`Int`** type.
- In the body of our object, we define the columns. In addition to the **`ID`** column, we'll have a **`name`** column that is of the **`varchar`** type, or in other words, a string, and is **`20`** characters at the most.
- The cat's **`name`** column is also unique, meaning that no two cats can have the same name.
- We also have a third column that is of the **`integer`** type, or **`Int`** in Kotlin terms, and is defaulted to **`0`**.

We'll also have a **data** class to represent a single cat:

```
data class Cat(val id: Int,
               val name: String,
               val age: Int)
```

The only thing that is left for us to do is add the following lines of code to our **mainModule()** function:

```
DB.connect()
transaction {
    SchemaUtils.create(CatsTable)
}
```

Each time our application starts, the preceding code will connect to the database. Then, it will attempt to create a table that stores our entities. If a table already exists, nothing will happen.

Now that we have established a connection to our database, let's examine how we can use this connection to store a few cats in it.

Creating new entities

Our next task is adding the first cat to our virtual shelter.

Following the REST principles, it should be a **POST** request, where the body of the request may look something like this:

```
{"name": "Meatloaf", "age": 4}
```

We'll start by writing a new test:

```
@Test
fun `POST creates a new cat`() {
    ...
}
```

Backticks are a useful Kotlin feature that allows us to have spaces in the names of our functions. This helps us create descriptive test names.

Next, let's look at the body of our test:

```
withTestApplication(Application::mainModule) {
    val response = handleRequest(HttpMethod.Post, "/cats") {
        addHeader(
          HttpHeaders.ContentType,
          ContentType.Application.FormUrlEncoded.toString()
        )
        setBody(
            listOf(
                "name" to "Meatloaf",
                "age" to 4.toString()
            ).formUrlEncode()
        )
    }.response
    assertEquals(HttpStatusCode.Created, response.status())
}
```

We discussed the **`withTestApplication`** and **`handleRequest`** functions in the previous section. This time, we are using a **`POST`** request. These types of requests should have the correct header, so we must set those headers using the **`addHeader()`** function. We must also set the body to the contents discussed previously.

Finally, we must check whether the response header is set to the **`Created`** HTTP code.

If we run this test now, it will fail with an HTTP code of **`404`** since we haven't implemented the **`post /cats`** endpoint yet.

Let's go back to our **`routing`** block and add a new endpoint:

```
post("/cats") {

    ...

    call.respond(HttpStatusCode.Created)

}
```

To create a new cat, we'll need to read the body of the **`POST`** request. We'll use the **`receiveParameters()`** function for this:

```
val parameters = call.receiveParameters()

val name = requireNotNull(parameters["name"])

val age = parameters["age"]?.toInt() ?: 0
```

The **`receiveParameters`** function returns a case-insensitive map. First, we will attempt to fetch the cat's **`name`** from this map, and if there's no name in the request, we will fail the call. This will be handled by Ktor.

Then, if we didn't receive **`age`**, we will default it to **`0`** using the Elvis operator.

Now, we must insert those values into the database:

```
transaction {

    CatsTable.insert { cat ->

        cat[CatsTable.name] = name

        cat[CatsTable.age] = age
    }
}
```

Here, we open a **`transaction`** block to make changes to the database. Then, we use the **`insert()`** method, which is available on every table. Inside the **`insert`** lambda, the **`cat`** variable refers to the new row we are going to populate. We set the name of that row to the value of the **`name`** parameter and do the same for **`age`**.

If you run your test now, it should pass. But if you run it again, it will fail. That's because the name of a cat in the database is unique. Also, we don't clean the database between test runs. So, the first run creates a cat named **`Meatloaf`**, while the second run fails. This is because such a cat already exists.

To make our tests consistent, we need a way to clean our database between runs.

Making the tests consistent

Let's go back to our test and add the following piece of code:

```
@BeforeEach
fun setup() {
    DB.connect()
    transaction {
        SchemaUtils.drop(CatsTable)
    }
}
```

Here, we are using the **`@BeforeEach`** annotation on a function. As its name suggests, this code will run before each test. The function will establish a connection to the database and drop the table completely. Then, our application will recreate the table.

Now, our tests should pass consistently. In the next section, we'll learn how to fetch a cat from the database using the Exposed library.

Fetching entities

Following the REST practices, the URL for fetching all cats should be **`/cats`**, while for fetching a single cat, it should be **`/cats/123`**, where **`123`** is the ID of the cat we are trying to fetch.

Let's add two new routes for that:

```
get("/cats") {

    ...

}

get("/cats/{id}") {

    ...

}
```

The first route is very similar to the **`/status`** route we introduced earlier in this chapter. But the second round is slightly different: it uses a query parameter in the URL. You can recognize query parameters by the curly brackets around their name.

To read a query parameter, we can access the **`parameters`** map:

```
val id = requireNotNull(call.parameters["id"]).toInt()
```

If there is an ID on the URL, we need to try and fetch a cat from the database:

```
val cat = transaction {
    CatsTable.select {
        CatsTable.id.eq(id)
    }.firstOrNull()
}
```

Here, we open a transaction and use the **`select`** statement to get a cat with an ID equal to what we were provided previously.

If an object was returned, we would convert it into JSON. Otherwise, we would return an HTTP code of **`404`**, **`Not Found`**:

```
if (row == null) {
    call.respond(HttpStatusCode.NotFound)
} else {
    call.respond(
        Cat(
            row[CatsTable.id].value,
            row[CatsTable.name],
            row[CatsTable.age]
        )
    )
}
```

Now, let's add a test for fetching a single cat as well:

```
@Test
```

```
fun `GET with ID fetches a single cat`() {
    withTestApplication(Application::mainModule) {
        val id = transaction {
            CatsTable.insertAndGetId { cat ->
                cat[name] = "Fluffy"
            }
        }
        val response = handleRequest(HttpMethod.Get,
            "/cats/$id").response
        assertEquals("""{"id":1,"name":
            "Fluffy","age":0}""", response.content)
    }
}
```

In this test, we create a cat using Exposed. Here, we're using a new method called **`insertAndGetId`**. As its name suggests, it will return the ID of a newly created row. Then, we try to fetch that cat using our newly created endpoint.

If we try to run this test, though, it will fail with the following exception:

```
> Serializer for class 'Cat' is not found.
```

By default, Ktor doesn't know how to turn our custom data class into JSON. To fix that, we'll need to add a new plugin to our **`build.gradle.kts`** file:

```
plugins {
    kotlin("jvm") version "1.5.10"
    application
    kotlin("plugin.serialization") version "1.5.10"
}
```

This plugin will create serializers at compile time for any class marked with the **`@Serializable`** annotation. All we need to do now for the test to pass is add that annotation to our **`Cat`** class:

```
@Serializable
data class Cat(
    val id: Int,
    val name: String,
    val age: Int
)
```

That's it; now, our test for fetching a cat by its ID should pass.

Finally, we would like to be able to fetch all the cats from the database. To do that, we must change our test setup a little:

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class ServerTest {
    @BeforeAll
    fun setup() {
        DB.connect()
        transaction {
            SchemaUtils.create(CatsTable)
        }
    }
    @AfterAll
    fun cleanup() {
        DB.connect()
        transaction {
            SchemaUtils.drop(CatsTable)
        }
    }
    ...
}
```

Here, we changed the setup of our test to drop the table once all the tests have been executed. So, instead of the **`@BeforeEach`** annotation, which executes the function before each test, we use the **`@AfterAll`** annotation, which executes the function after all tests have been executed.

For this annotation to work, we also need to add the **`@TestInstance`** annotation to our class. The default for that is **`PER_METHOD`**, but since we want to execute multiple tests at once, and then clean up after, we need to set the life cycle of our test class to **`PER_CLASS`**.

Next, let's wrap our test into a nested class, like this:

```
@Nested
inner class `With cat in DB` {

    @Test
    fun `GET with ID fetches a single cat`() {
        ...
    }
}
```

Nested test classes are a great way to encapsulate specific test situations. In our case, we would like to run two tests when there is a cat in our database already.

Now, let's add the following setup code to our nested test:

```
lateinit var id: EntityID<Int>

@BeforeEach

fun setup() {

    DB.connect()

    id = transaction {

        CatsTable.insertAndGetId { cat ->

            cat[name] = "Fluffy"

        }

    }

}

@AfterEach

fun teardown() {

    DB.connect()

    transaction {

        CatsTable.deleteAll()

    }

}
```

Before we execute each test in this nested class, we will create a cat in the database and after each test, we will delete all the cats from our database. Since we would like to keep track of the ID of the cat that we create, we will store it in a variable.

Now, our test class for fetching a single entity looks like this:

```
@Test

fun `GET with ID fetches a single cat`() {

    withTestApplication(Application::mainModule) {

        val response = handleRequest(HttpMethod.Get,
            "/cats/$id").response         assertEquals("""
{"id":$id,"name":"Fluffy",            "age":0}""",
response.content)

    }

}
```

Notice that we interpolate the cat's ID into our expected response since it will change with each test execution.

The test for fetching all the cats from the database will look almost the same:

```
@Test

fun `GET without ID fetches all cats`() {

    withTestApplication(Application::mainModule) {

        val response = handleRequest(HttpMethod.Get,
            "/cats").response         assertEquals("""
[{"id":$id,"name":"Fluffy",            "age":0}]""",
response.content)

    }

}
```

We just don't specify the ID, and the response is wrapped into a JSON array, as you can see by the square brackets around it.

Now, all we need to do is implement this new route:

```
get("/cats") {
    val cats = transaction {
        CatsTable.selectAll().map { row ->
            Cat(
                row[CatsTable.id].value,
                row[CatsTable.name],
                row[CatsTable.age]
            )
        }
    }
    call.respond(cats)
}
```

If you followed the example for fetching a single entity from the database (from the beginning of this section), then this example won't be very different for you. We use the **`selectAll()`** function to fetch all the rows from the table. Then, we map each row to our **`data`** class. The only problem that is left for us to solve is that our code is quite messy and resides in a single file. It would be better if we split all the cat routes into a separate file. We'll do that in the next section.

Organizing routes in Ktor

In this section, we'll see what the idiomatic approach in Ktor is for structuring multiple routes that belong to the same domain.

Our current **`routing`** block looks like this:

```
routing {
    get("/status") {
        ...
    }
    post("/cats") {
        ...
    }
    get("/cats") {
        …
    }
    get("/cats/{id}") {
        ...
    }
}
```

It would be good if we could extract all the routes that are related to cats into a separate file. Let's start by replacing all the cat routes with a function:

```
routing {
    get("/status") {
        ...
    }
    cats()
}
```

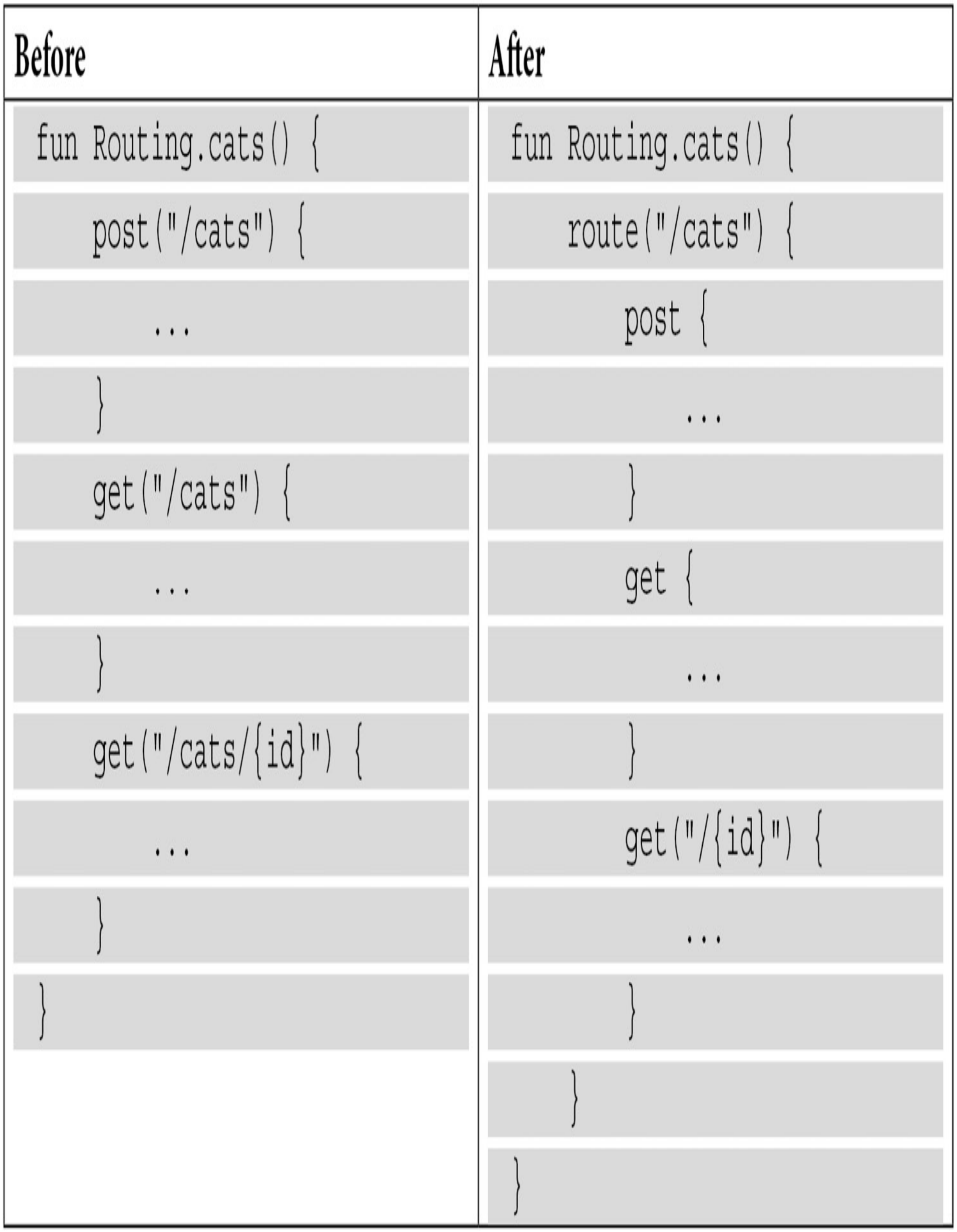
If you are using IntelliJ IDEA, it will even suggest that you generate an extension function on the **`Routing`** class:

```
fun Routing.cats() {
    ...
}
```

Now, we can move all our cat routes to this function:

```
fun Routing.cats() {

    post("/cats") {

        ...

    }

    get("/cats") {

        ...

    }

    get("/cats/{id}") {

        ...

    }

}
```

Now, you can see that the **`/cats`** URL is repeated many times. We can lift it using the **`route()`** block:



Before	After
`fun Routing.cats() {`	`fun Routing.cats() {`
`    post("/cats") {`	`    route("/cats") {`
`        ...`	`        post {`
`    }`	`            ...`
`    get("/cats") {`	`        }`
`        ...`	`        get {`
`    }`	`            ...`
`    get("/cats/{id}") {`	`        }`
`        ...`	`        get("/{id}") {`
`    }`	`            ...`
`}`	`        }`
	`    }`
	`}`

Table 10.1 - Cleaner code after using the route() block

Notice how much cleaner our code has become now.

Now, there's one last important topic for us to cover. At the beginning of this chapter, we mentioned that Ktor is a highly concurrent framework. And in Chapter 6, *Threads and Coroutines*, we said that concurrency in Kotlin is mainly achieved by using coroutines. But we have started a single coroutine in this chapter. We'll look at this in the next section.

Achieving concurrency in Ktor

Looking back at the code we've written in this chapter, you may be under the impression that the Ktor code is not concurrent at all. However, this couldn't be further from the truth.

All the Ktor functions we've used in this chapter are based on coroutines and the concept of **suspending functions**.

For every incoming request, Ktor will start a new coroutine that will handle it, thanks to the CIO server engine, which is based on coroutines at its core. Having a concurrency model that is performant but not obtrusive is a very important principle in Ktor.

In addition, the `routing` blocks we used to specify all our endpoints have access to `CoroutineScope`, meaning that we can invoke suspending functions within those blocks.

One of the examples for such a suspending function is `call.respond()`, which we were using throughout this chapter. Suspending functions provide our application with opportunities to context switch, and to execute other code concurrently. This means that the same number of resources can serve far more requests than they would be able to otherwise. We'll stop here and summarize what we've learned about developing applications using Ktor.

Summary

In this chapter, we have built a well-tested service using Kotlin that uses the Ktor framework to store entities in the database. We've also discussed how the multiple design patterns that we encountered at the beginning of this book, such as Factory, Singleton, and Bridge, are used in the Ktor framework to provide a flexible structure for our code.

Now, you should be able to interact with the database using the Exposed framework. We've learned how we can declare, create, and drop tables, how to insert new entities, and how to fetch and delete them.

In the next chapter, we'll look at an alternative approach to developing web applications, but this time using a Reactive framework called **Vert.x**. This will allow us to compare the concurrent and Reactive approaches for developing web applications and discuss the tradeoffs of each of the approaches.

Questions

1. How are the Ktor applications structured and what are their benefits?
2. What are plugins in Ktor and what are they used for?
3. What is the main problem that the Exposed library solves?

Chapter 11: Reactive Microservices with Vert.x

In the previous chapter, we familiarized ourselves with the Ktor framework. We created a web service that could store cats in its database.

In this chapter, we'll continue working on the example from the previous chapter, but this time using the Vert.x framework and Kotlin. **Vert.x** is a Reactive framework that is built on top of Reactive principles, which we discussed in *Chapter 7*, *Controlling the Data Flow*. We'll list some of the other benefits of the Vert.x framework in this chapter. You can always read more about Vert.x by going to the official website: https://vertx.io.

The microservice we'll develop in this chapter will provide an endpoint for health checks – the same as the one we created in Ktor – and will be able to delete and update the cats in our database.

In this chapter, we will cover the following topics:

- Getting started with Vert.x
- Routing in Vert.x
- Verticles
- Handling requests
- Testing Vert.x applications
- Working with databases
- Understanding Event Loop

- Communicating with Event Bus

Technical requirements

For this chapter, you'll need the following:

- **JDK 11** or later
- IntelliJ IDEA
- **Gradle 6.8** or later
- **PostgreSQL 14** or later

Like the previous chapter, this chapter will also assume that you have PostgreSQL already installed and that you have basic knowledge of working with it. We'll also use the same table structure we created with Ktor.

You can find the full source code for this chapter here: https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter11.

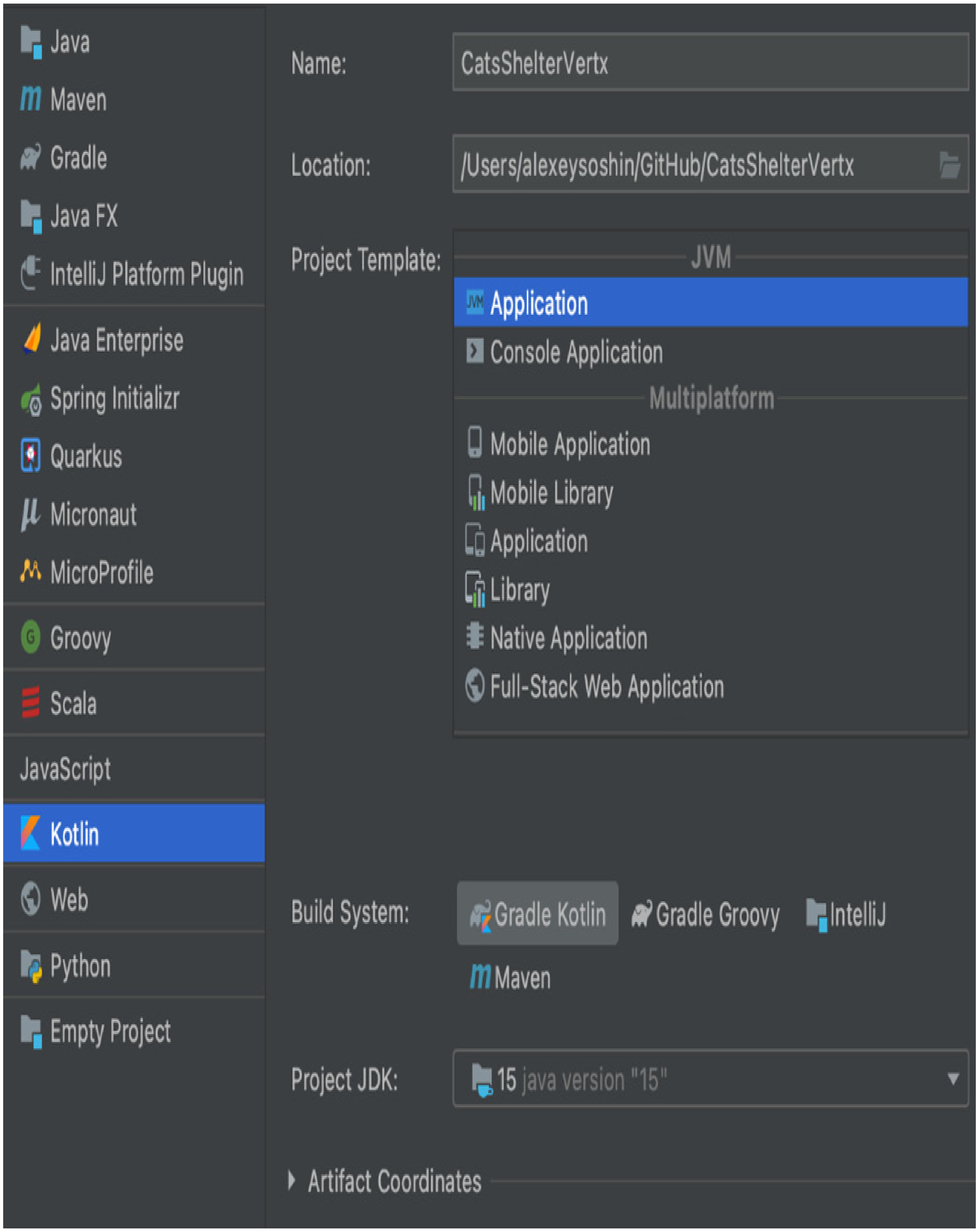
Getting started with Vert.x

Vert.x is a Reactive framework that is asynchronous and non-blocking. Let's understand what this means by looking at a concrete example.

We'll start by creating a new Kotlin Gradle project or by using start.vertx.io:

1. From your IntelliJ IDEA application, select **File** | **New** | **Project** and choose **Kotlin** from the **New Project** wizard.

2. Then, specify a name for your project – `CatsShelterVertx`, in my case – and choose **Gradle Kotlin** as your **Build System**.
3. Then, select the **Project JDK** version that you have installed from the dropdown. The output should look as follows:



Name: CatsShelterVertx

Location: /Users/alexeysoshin/GitHub/CatsShelterVertx

Project Template:

JVM
- Application
- Console Application

Multiplatform
- Mobile Application
- Mobile Library
- Application
- Library
- Native Application
- Full-Stack Web Application

Build System: Gradle Kotlin | Gradle Groovy | IntelliJ | Maven

Project JDK: 15 java version "15"

Artifact Coordinates

Figure 11.1 – Creating a Kotlin application

Next, add the following dependencies to your **`build.gradle.kts`** file:

```
val vertxVersion = "4.1.5"

dependencies {

    implementation("io.vertx:vertx-core:$vertxVersion")

    implementation("io.vertx:vertx-web:$vertxVersion")

    implementation("io.vertx:vertx-lang-

        kotlin:$vertxVersion")

    implementation("io.vertx:vertx-lang-kotlin-

        coroutines:$vertxVersion")

    ...

}
```

Similar to what we discussed in the previous chapter, all the dependencies must be of the same version to avoid any conflicts. That's the reason we are using a variable for the library version – to be able to change all of them together.

The following is an explanation of each dependency:

- **`vertx-core`** is the core library.
- **`vertx-web`** is needed since we want our service to be REST-based.
- **`vertx-lang-kotlin`** provides idiomatic ways to write Kotlin code with Vert.x.
- Finally, **`vertx-lang-kotlin-coroutines`** integrates with the coroutines, which we discussed in detail in Chapter 6, *Threads and Coroutines*.

Then, we must create a file called **`server.kt`** in the **`src/main/kotlin`** folder with the following content:

```
fun main() {
    val vertx = Vertx.vertx()

    vertx.createHttpServer().requestHandler{ ctx ->

        ctx.response().end("OK")

    }.listen(8081)

    println("open http://localhost:8081")

}
```

That's all you need to start a web server that will respond with **OK** when you open **http://localhost:8081** in your browser.

Now, let's understand what happens here. First, we create a Vert.x instance using the Factory method from Chapter 3, *Understanding Structural Patterns*.

The **requestHandler** method is just a simple listener or a subscriber. If you don't remember how it works, check out Chapter 4, *Getting Familiar with Behavioral Patterns*, for the Observable design pattern. In our case, it will be called for each new request. That's the asynchronous nature of Vert.x in action.

Next, let's learn how to add routes in Vert.x.

Routing in Vert.x

Notice that no matter which URL we specify, we always get the same result. Of course, that's not what we want to achieve. Let's start by adding the most basic endpoint, which will only tell us that the service is up and running.

For that, we'll use **Router**:

```
val vertx = Vertx.vertx()

val router = Router.router(vertx)

...
```

`Router` lets you specify handlers for different HTTP methods and URLs.

Now, let's add a **`/status`** endpoint that will return an HTTP status code of **`200`** and a message stating **`OK`** to our user:

```
router.get("/status").handler { ctx ->

    ctx.response()

        .setStatusCode(200)

        .end("OK")

}

vertx.createHttpServer()

    .requestHandler(router)

    .listen(8081)
```

Now, instead of specifying the request handler as a block, we will pass this function to our **`router`** object. This makes our code easier to manage.

We learned how we return a flat text response in the very first example. So, now, let's return JSON instead. Most real-life applications use JSON for communication. Let's replace the body of our status handler with the following code:

```
val json = json {

    obj(

        "status" to "OK"

    )

}
```

```
ctx.response()
    .setStatusCode(200)
    .end(json.toString())
```

Here, we are using a DSL, which we discussed in Chapter 4, *Getting Familiar with Behavioral Patterns*, to create a JSON object.

You can open **`http://localhost:8081/status`** in your browser and make sure that you get **`{"status": "OK"}`** as a response.

Now, let's discuss how we can structure our code better with the Vert.x framework.

Verticles

Our current code resides in the **`server.kt`** file, which is getting bigger and bigger. We need to find a way to split it. In Vert.x, code is split into classes called **verticles**.

You can think of a verticle as a lightweight actor. We discussed Actors in Chapter 5, *Introducing Functional Programming*.

Let's see how we can create a new verticle that will encapsulate our server:

```
class ServerVerticle : CoroutineVerticle() {
    override suspend fun start() {
        val router = router()
        vertx.createHttpServer()
            .requestHandler(router)
            .listen(8081)
        println("open http://localhost:8081")
    }

    private fun router(): Router {

        // Our router code comes here now

        val router = Router.router(vertx)

        ...

        return router

    }

}
```

Every verticle has a **`start()`** method that handles its initialization. As you can see, we moved all the code from our **`main()`** function to the **`start()`** method. If we run the code now, though, nothing will happen. That's because the verticle hasn't been started yet.

There are different ways to start a verticle, but the simplest way is to pass the instance of the class to the **`deployVerticle()`** method. In our case, this is the **`ServerVerticle`** class:

```
fun main() {

    val vertx = Vertx.vertx()

    vertx.deployVerticle(ServerVerticle())

}
```

Here is another, more flexible way to specify the class name as a string:

```
fun main() {

    val vertx = Vertx.vertx()

    vertx.deployVerticle("ServerVerticle")

}
```

If our verticle class is not in the default package, we'll need to specify the fully qualified path for Vert.x to be able to initialize it.

Now, our code has been split into two files, **`ServerVerticle.kt`** and **`server.kt`**, and is organized better. Next, we'll learn how we can do the same refactoring to organize our routes in a better way.

Handling requests

As we discussed earlier in this chapter, all requests in Vert.x are handled by the **`Router`** class. We covered the concept of routing in the previous chapter, so now, let's just discuss the differences between the Ktor and Vert.x approaches to routing requests.

We'll declare two endpoints to delete cats from the database and update information about a particular cat. We'll use the **`delete`** and **`put`** verbs, respectively, for this:

```
router.delete("/cats/:id").handler { ctx ->

    // Code for deleting a cat

}

router.put("/cats/:id").handler { ctx ->

    // Code for updating a cat

}
```

Both endpoints receive a URL parameter. In Vert.x, we use a colon notation for this.

To be able to parse JSON requests and responses, Vert.x has a **`BodyHandler`** class. Now, let's declare it as well. This should come just after the instantiation of our router:

```
val router = Router.router(vertx)

router.route().handler(BodyHandler.create())
```

This will tell Vert.x to parse the request body into JSON for any request.

Notice that the **`/cat`** prefix is repeated multiple times in our code now. To avoid that and make our code more modular, we can use a subrouter, which we'll discuss in the next section.

Subrouting the requests

Subrouting allows us to split routes into multiple classes to keep our code more organized. Let's move the new routes to a new function by following these steps:

1. We'll leave the **`/alive`** endpoint as is, but we'll extract all the other endpoints into a separate function:

   ```
   private fun catsRouter(): Router {

       val router = Router.router(vertx)

       router.delete("/:id").handler { ctx ->

           // Code for deleting a cat

       }

       router.put("/:id").handler { ctx ->

           // Code for updating a cat

       }

       return router

   }
   ```

Inside this function, we create a separate **`Router`** object that will only handle the routes for cats, not the status routes.

2. Now, we need to connect **`SubRouter`** to our main router:

   ```
   router.mountSubRouter("/cats", catsRouter())
   ```

   Keeping our code clean and well separated is very important. Extracting routes into subrouters helps us with that.

Now, let's discuss how this code can be tested.

Testing Vert.x applications

To test our Vert.x application, we'll use the **JUnit 5** framework, which we discussed in the previous chapter.

You'll need the following two dependencies in your **`build.gradle.kts`** file:

```
dependencies {
    ...
    testImplementation("org.junit.jupiter:junit-jupiter-
        api:5.6.0")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-
        engine:5.6.0")
}
```

Our first test will be located in the **`/src/test/kotlin/ServerTest.kt`** file.

The basic structure of all the integration tests looks something like this:

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class ServerTest {
```

```
    private val vertx: Vertx = Vertx.vertx()

    @BeforeAll

    fun setup() {

        runBlocking {

            vertx.deployVerticle(ServerVerticle()).await()

        }

    }

    @AfterAll

    fun tearDown() {

        // And you want to stop your server once

        vertx.close()

    }


    @Test

    fun `status should return 200`() {

    }

}
```

This structure is different from what we've seen in Ktor. Here, we start the server ourselves, in the **`setup()`** method.

Since Vert.x is Reactive, the **`deployVerticle()`** method will return a **`Future`** object immediately, releasing the thread, but that doesn't mean that the server verticle has started yet.

To avoid this race, we can use the **`await()`** method, which will block the execution of our tests until the server is ready to receive requests.

Now, we want to issue an actual HTTP call to our **`/status`** endpoint, for example, and check the response code. For that, we'll use the Vert.x web client.

Let's add it to our **`build.gradle.kts`** dependencies section:

```
testImplementation("io.vertx:vertx-web-client:$vertxVersion")
```

Since we only plan to use **`WebClient`** in tests, we specify **`testImplementation`** instead of **`implementation`**. But **`WebClient`** is so useful that you'll probably end up using it in your production code anyway.

After adding this new dependency, we need to instantiate our web client in the **`setup`** method:

```
lateinit var client: WebClient

@BeforeAll

fun setup() {

    vertx.deployVerticle(ServerVerticle())

    client = WebClient.create(

        vertx,

        WebClientOptions()

            .setDefaultPort(8081)

            .setDefaultHost("localhost")

    )

}
```

The **`setup()`** method will be called once before all the tests start. In this method, we are deploying our server verticle and creating a web client with some defaults for all our tests to share.

Now, let's write a test to check that our server is up and running:

```
@Test

fun `status should return 200`() {

    runBlocking {

        val response = client.get("/status").send().await()

        assertEquals(201, response.statusCode())

    }

}
```

Now, let's understand what happens in this test:

- **`client`** is an instance of **`WebClient`** that is shared by all our tests. We invoke the **`/status`** endpoint using the **`get`** verb. This is a Builder design pattern, so to issue our request, we need to use the **`send()`** method. Otherwise, nothing will happen.
- Vert.x is a Reactive framework, so instead of blocking our thread until a response is received, the **`send()`** method returns a Future. Then, we use **`await()`**, which adapts a Future to a Kotlin coroutine to be able to wait for the results concurrently.
- Once the response is received, we check it in the same way that we did in other tests – by using the **`assertEquals`** function, which comes from JUnit.

Now that we know how to write tests in Vert.x, let's discuss how we can work with databases in a Reactive manner.

Working with databases

To be able to progress further with our tests, we need the ability to create entities in the database. For that, we'll need to connect to the database.

First, let's add the following two lines to our **`build.gradle.kts`** dependencies section:

```
implementation("org.postgresql:postgresql:42.3.0")
```

```
implementation("io.vertx:vertx-pg-client:$vertxVersion")
```

The first line of code fetches the PostgreSQL driver. The second one adds the Vert.x JDBC client, which allows Vert.x, which has the driver, to connect to any database that supports JDBC.

Managing configuration

Now, we want to hold the database configuration somewhere. For local development, it may be fine to have those configurations hardcoded. We'll execute the following steps to do this:

1. When we connect to the database, we need to specify the following parameters at the very least:
    - Username
    - Password
    - Host
    - Database name

   We'll store the preceding parameters in a **`Singleton`** object:

   ```
   object Db {
   ```

```
    val username = System.getenv("DATABASE_USERNAME")         ?:
"cats_admin"

    val password = System.getenv("DATABASE_PASSWORD")         ?:
"abcd1234"

    val database = System.getenv("DATABASE_NAME")         ?:
"cats_db"

    val host = System.getenv("DATABASE_HOST")         ?:
"localhost"

}
```

Our **`Singleton`** object has four members. For each, we check whether an environment variable was set, and if there's no such environment variable, we provide a default value using the Elvis operator.

2. Now, let's add a function that will return a connection pool:

```
fun connect(vertx: Vertx): SqlClient {

    val connectOptions = PgConnectOptions()

        .setPort(5432)

        .setHost(host)

        .setDatabase(database)

        .setUser(username)

        .setPassword(password)

    val poolOptions = PoolOptions()

        .setMaxSize(20)

    return PgPool.client(

        vertx,

        connectOptions,

        poolOptions
    )
}
```

Our **`connect()`** method creates two configuration objects: **`PgConnectOptions`** sets the configuration for the database we want to connect to, while **`PoolOptions`** specifies the configuration of the connection pool.

3. Now, all we need to do is instantiate the database client in our test:

```
...
lateinit var db: SqlClient
@BeforeAll
fun setup() {
    runBlocking {
        ...
        db = Db.connect(vertx)
    }
}
```

4. Having done that, let's create a new **`Nested`** class in our test file for cases where we expect to have a cat in our database:

```
@Nested
inner class `With Cat` {
    @BeforeEach
    fun createCats() {
        ...
    }
    @AfterEach
    fun deleteAll() {
        ...
    }
}
```

Unlike the Exposed framework, which we discussed in the previous chapter, the database client in Vert.x doesn't have specific methods for insertion, deletion, and so on. Instead, it provides a lower-level API that allows us to execute any type of query on the database.

5. First, let's write a query that will clean our database:

```
@AfterEach
fun deleteAll() {
    runBlocking {
        db.preparedQuery("DELETE FROM
cats")            .execute().await()
    }
}
```

The basic structure for working with the database client in Vert.x is to pass a query to the **`prepareQuery()`** method, then execute it using **`execute()`**.

We want to wait for the query to complete before we move on to the next test, so we use the **`await()`** function to wait for the current coroutine, and we use the **`runBlocking()`** adapter method to have a coroutine context to do so.

6. Now, let's write another query that will add a cat to the database before each test runs:

   ```
   lateinit var catRow: Row
   @BeforeEach
   fun createCats() {
       runBlocking {
           val result = db.preparedQuery(
               """INSERT INTO cats (name, age)
               VALUES ($1, $2)
               RETURNING ID""".trimIndent()
           ).execute(Tuple.of("Binky", 7)).await()
           catRow = result.first()
       }
   }
   ```

   Here, we are using the **`preparedQuery()`** method once more, but this time, our SQL query string contains placeholders. Each placeholder starts with a dollar sign and their indexes start with **`1`**.

   Then, we pass the values for those placeholders to the **`execute()`** method. **`Tuple.of`** is a Factory method design pattern that you should be able to recognize well by now.

   We also want to remember the ID of the cat that we create since we'll use that ID to delete or update the cat. For this reason, we store the created row in a **`lateinit`** variable.

7. We now have everything prepared to write our test:

```
@Test

fun `delete deletes a cat by ID`() {

    runBlocking {

        val catId = catRow.getInteger(0)

        client.delete("/cats/${catId}").send().await()

        val result = db.preparedQuery("SELECT * FROM
            cats WHERE id =
$1")            .execute(Tuple.of(catId)).await()

        assertEquals(0, result.size())

    }

}
```

First, we get the ID of the cat we want to delete from the database row using the **`getInteger()`** method. Unlike parameters that start with **`1`**, the columns of a database row start with **`0`**. So, by getting an integer at index **`0`**, we get the ID of our cat.

Then, we invoke the web client's **`delete()`** method and wait for it to complete.

Afterward, we execute a **`SELECT`** statement on our database, checking that the row was indeed deleted.

If you run this test now, it will fail, because we haven't implemented the **`delete`** endpoint yet. We'll do that in the next section.

Understanding Event Loop

The goal of the **Event Loop** design pattern is to continuously check for new events in a queue, and each time a new event comes in, to quickly dispatch it to someone who knows how to handle it. This way, a single thread or a very limited number of threads can handle a huge number of events.

In the case of web frameworks such as Vert.x, events may be requests to our server.

To understand the concept of the Event Loop better, let's go back to our server code and attempt to implement an endpoint for deleting a cat:

```
val db = Db.connect(vertx)

router.delete("/:id").handler { ctx ->

    val id = ctx.request().getParam("id").toInt()

    db.preparedQuery("DELETE FROM cats WHERE ID =
$1")        .execute(Tuple.of(id)).await()

    ctx.end()

}
```

This code is very similar to what we've written in our tests in the previous section. We read the URL parameter from the request using the **`getParam()`** function, then we pass this ID to the prepared query. This time, though, we can't use the **`runBlocking`** adapter function, since it will block the Event Loop.

Vert.x uses a limited number of threads, as many as twice the number of your CPU cores, to run all its code efficiently. However, this means that we cannot execute any blocking operations on those threads since it will negatively impact the performance of our application.

To solve this issue, we can use a coroutine builder we're already familiar with: **`launch()`**. Let's see how this works:

```
router.delete("/:id").handler { ctx ->
```

```
    launch {
        val id = ctx.request().getParam("id").toInt()
        db.preparedQuery("DELETE FROM cats WHERE ID =
$1")            .execute(Tuple.of(id)).await()
        ctx.end()
    }
}
```

Since our verticle extends **`CoroutineVerticle`**, we have access to all the regular coroutine builders that will run on the Event Loop.

Now, all we need to do is mark our routing functions with the **`suspend`** keyword:

```
private suspend fun router(): Router {
    ...
}
private suspend fun catsRouter(): Router {
    ...
}
```

Now, let's add another test for updating a cat:

```
@Test
fun `put updates a cat by ID`() {
    runBlocking {
        val catId = catRow.getInteger(0)
        val requestBody = json {
            obj("name" to "Meatloaf", "age" to 4)
        }
        client.put("/cats/${catId}")

            .sendBuffer(Buffer.buffer(requestBody.toString()))

            .await()

        val result = db.preparedQuery("SELECT * FROM cats

            WHERE id = $1")

            .execute(Tuple.of(catId)).await()

        assertEquals("Meatloaf",
            result.first().getString("name"))

        assertEquals(4, result.first().getInteger("age"))

    }

}
```

This test is very similar to the deletion test, with the only major difference being that we use **`sendBuffer`** and not the **`send()`** method, so we can send a JSON body to our **`put`** endpoint.

We create the JSON similarly to what we saw when we implemented the **`/status`** endpoint earlier in this chapter.

Now, let's implement the **`put`** endpoint for the test to pass:

```
router.put("/:id").handler { ctx ->

    launch {

        val id = ctx.request().getParam("id").toInt()

        val body = ctx.bodyAsJson

        db.preparedQuery("UPDATE cats SET name = $1, age =
            $2 WHERE ID = $3")

            .execute(

                Tuple.of(
                    body.getString("name"),
                    body.getInteger("age"),
                    id
                )
            ).await()
        ctx.end()
    }
}
```

Here, the main difference from the previous endpoint we've implemented is that this time, we need to parse our request **`body`**. We can do that by using the **`bodyAsJson`** property. Then, we can use the **`getString`** and **`getInteger`** methods, which are available in JSON, to get the new values for **`name`** and **`age`**.

With this, you should have all the required knowledge to implement other endpoints as needed. Now, let's learn how to structure our code in a better way using the concept of Event Bus since it all resides in a single large class.

Communicating with Event Bus

Event Bus is an implementation of the Observable design pattern, which we discussed in Chapter 4, *Getting Familiar with Behavioral Patterns*.

We've already mentioned that Vert.x is based on the concept of verticles, which are isolated actors. We've already seen the other types of actors in Chapter 6, *Threads and Coroutines*. Kotlin's **`coroutines`** library provides the `actor()` and `producer()` coroutine generators, which create a coroutine bound to a channel.

Similarly, all the verticles in the Vert.x framework are bound by Event Bus and can pass messages to one another using it. Now, let's extract the code from our `ServerVerticle` class into a new class, which we'll call `CatVerticle`.

Any verticle can send a message over Event Bus by choosing between the following methods:

- `request()` will send a message to only one subscriber and wait for a response.
- `send()` will send a message to only one subscriber, without waiting for a response.
- `publish()` will send a message to all subscribers, without waiting for a response.

No matter which method is used to send the message, you subscribe to it using the `consumer()` method on Event Bus.

Now, let's subscribe to an event in our `CatsVerticle` class:

```
class CatsVerticle : CoroutineVerticle() {
    override suspend fun start() {
        val db = Db.connect(vertx)
        vertx.eventBus().consumer<Int>("cats:delete"){req->
            launch {
                val id = req.body()
```

```
                    db.preparedQuery("DELETE FROM
                      cats WHERE ID = $1")
                        .execute(Tuple.of(id)).await()
                    req.reply(null)
                }
            }
        }
}
```

The generic type of the **`consumer()`** method specifies the type of message we'll receive. In this case, it's **`Int`**.

The string that we provide to the method – in our case, **`cats:delete`** – is the address we subscribe to. It can be any string, but it is good to have some convention, such as what type of object we operate on and what we want to do with it.

Once the delete action has been executed, we respond to our publisher with the **`reply()`** method. Since we don't have any information to send back, we simply send **`null`**.

Now, let's replace our previous **`delete`** route with the following code:

```
router.delete("/:id").handler { ctx ->
    val id = ctx.request().getParam("id").toInt()
    vertx.eventBus().request<Nothing>("cats:delete", id) {
        ctx.end()
    }
}
```

Here, we send the ID of the cat we received from the request to one of our listeners using the **`request()`** method, and we specify that the type of our message is **`Int`**. We also use the same address we specified in the consumer code.

Since we have split our code into a new verticle, we need to remember to start it as well. Add the following line to both the **`main()`** function and the **`setup()`** method in your test:

```
vertx.deployVerticle(CatsVerticle())
```

Next, let's learn how to send complex objects over Event Bus.

Sending JSON over Event Bus

As our final exercise, let's learn how to update a cat. For that, we'll need to send more than just an ID over Event Bus.

Let's rewrite our **`put`** handler, as follows:

```
router.put("/:id").handler { ctx ->
    launch {
        val id = ctx.request().getParam("id").toInt()
        val body: JsonObject =
ctx.bodyAsJson.mergeIn(json{            obj("id" to id)
        })
        vertx.eventBus().request<Int>("cats:update", body)
          { res ->
            ctx.end(res.result().body().toString())
        }
    }
}
```

Here, you can see that we can send JSON objects over Event Bus easily. We merge the ID we receive as a URL parameter with the rest of the request **`body`** and send this JSON over an Event Bus. When a response is received, we output it back to the user.

Now, let's see how we consume the event we just sent:

```
vertx.eventBus().consumer<JsonObject>("cats:update"){req -
>    launch {
        val body = req.body()
        db.preparedQuery("UPDATE cats SET name = $1, age =
            $2 WHERE ID = $3")
            .execute(
                Tuple.of(
                    body.getString("name"),
                    body.getInteger("age"),
                    body.getInteger("id")
                )
            ).await()
        req.reply(body.getInteger("id"))
    }
}
```

We moved our logic from **`Router`** to our **`CatsVerticle`** class, but since we use JSON to communicate, the code stayed almost the same. In our verticle, we listen to the **`cats:update`** event, and once we receive the response, we extract **`name`**, **`age`**, and **`id`** from the JSON object to confirm that the operation was successful.

This concludes our chapter. There is still much for you to learn about the Vert.x framework in case you're curious, but with the knowledge you've gained from this chapter at hand, you should be able to do so with some confidence.

Summary

This chapter concludes our journey into the design patterns in Kotlin. Vert.x uses actors, called verticles, to organize the logic of the application. Actors communicate between themselves using Event Bus, which is an implementation of the Observable design pattern.

We also discussed the Event Loop pattern, how it allows Vert.x to process lots of events concurrently, and why it's important not to block its execution.

Now, you should be able to write microservices in Kotlin using two different frameworks, and you can choose what approach works best for you.

Vert.x provides a lower-level API than Ktor, which means that we may think more about how we structure our code, but the resulting application may be more performant as well. Since this is the end of this book, all that's left is for me to wish you the best of luck in learning about Kotlin and its ecosystem. You can always get some help from me and other Kotlin enthusiasts by going to https://stackoverflow.com/questions/tagged/kotlin and https://discuss.kotlinlang.org/.

Happy learning!

Questions

1. What's a verticle in Vert.x?
2. What's the goal of the Event Bus?
3. Why shouldn't we block the Event Loop?

Assessments

Chapter 1, Getting Started with Kotlin

Question 1

What's the difference between **`var`** and **`val`** in Kotlin?

Answer

The **`val`** keyword declares an immutable value that cannot be modified once assigned. The **`var`** keyword declares a mutable variable that can be assigned multiple times.

Question 2

How do you extend a class in Kotlin?

Answer

To extend a class, you can specify its name and constructor after a semicolon. If it's a regular class, it must be declared **`open`** for your code to be able to extend it.

Question 3

How do you add functionality to a **`final`** class?

Answer

To add functionality to a class that we cannot inherit from, we can use an extension function. The extension function will have access only to the class itself and to its public fields and functions.

Chapter 2, Working with Creational Patterns

Question 1

Name two uses for the `object` keyword we learned about in this chapter.

Answer

The `object` keyword is used to declare a singleton if it's used in a global scope or as a collection of static methods if it's used in a conjunction with the `companion` keyword inside a class.

Question 2

What is the `apply()` function used for?

Answer

The `apply()` function is used when we want to change the state of an object and then return it immediately.

Question 3

Provide one example of a static factory method that we discussed in this chapter.

Answer

The JVM **`valueOf()`** method on the **`Long`** objects is a static factory method.

Chapter 3, Understanding Structural Patterns

Question 1

What differences are there between the implementations of the Decorator and Proxy design patterns?

Answer

The Decorator and Proxy design patterns could be implemented in the same manner. The only difference is in their intent – the Decorator design pattern adds functionality to an object, while the Proxy design pattern may change an object's functionality.

Question 2

What is the main goal of the Flyweight design pattern?

Answer

The goal of the Flyweight design pattern is to conserve memory by reusing the same immutable state across multiple lightweight objects.

Question 3

What is the difference between the Facade and Adapter design patterns?

Answer

The Facade design pattern creates a new interface to simplify working with complex code, while the Adapter design pattern allows one interface to substitute another interface.

Chapter 4, Getting Familiar with Behavioral Patterns

Question 1

What's the difference between Mediator and Observer design patterns?

Answer

Both serve a similar purpose. Mediator introduces tight coupling between components that may serve different purposes, while Observer operates on similar components that are loosely coupled.

Question 2

What is a **Domain-Specific Language** (**DSL**)?

Answer

A DSL is a language that focuses on solving problems in a specific domain. This is different from a general-purpose language, such as Kotlin, that can be applied to different domains. Kotlin encourages developers to create DSLs for their needs.

Question 3

What are the benefits of using a sealed class or interface?

Answer

Since all types of a sealed class are known at compile time, Kotlin compiler can verify that the **`when`** statement covers all cases or, in other words, is exhaustive.

Chapter 5, Introducing Functional Programming

Question 1

What are higher order functions?

Answer

A higher order function is any function that either receives another function as input or returns a function as output.

Question 2

What is the `tailrec` keyword in Kotlin?

Answer

The purpose of the `tailrec` keyword is to allow the Kotlin compiler to optimize tail recursion and avoid stack overflow.

Question 3

What are pure functions?

Answer

Pure functions are functions that don't have any side effects, such as I/O.

Chapter 6, Threads and Coroutines

Question 1

What are the different ways to start a coroutine in Kotlin?

Answer

A coroutine in Kotlin could be started with either the `launch()` or `async()` functions. The difference is that `async()` also returns a value, while `launch()` doesn't.

Question 2

With structured concurrency, if one of the coroutines fails, all the siblings will be canceled as well. How can we prevent that behavior?

Answer

We can prevent canceling siblings by using `supervisorScope` instead of `coroutineScope`.

Question 3

What is the purpose of the `yield()` function?

Answer

The `yield()` function returns a value and suspends the coroutine until it has been resumed.

Chapter 7, Controlling the Data Flow

Question 1

What is the difference between higher order functions on collections and on concurrent data structures?

Answer

Higher order functions on collections will process the entire collection, creating a copy of it, before proceeding to the next step. Higher order functions on concurrent data structures are reactive, processing one element after the other.

Question 2

What is the difference between cold and hot streams of data?

Answer

A cold stream repeats itself for each new consumer, while the hot stream will only send the available data to the new consumer from the time of subscription.

Question 3

When should a conflated channel/flow be used?

Answer

A conflated flow can be used in situations when the consumer is slower than the producer and some of the messages could be dropped, leaving only the most recent message for consumption.

Chapter 8, Designing for Concurrency

Question 1

What does it mean when we say that the `select` expression in Kotlin is biased?

Answer

A biased `select` expression means that in case of a *draw* between two channels, the first channel listed in the `select` expression will always be picked.

Question 2

When should you use a mutex instead of a channel?

Answer

Mutexes are used to protect a resource shared between multiple coroutines. Channels are used to pass data between coroutines.

Question 3

Which of the concurrent design patterns could help you implement **MapReduce** or a **divide and conquer** algorithm efficiently?

Answer

For divide and conquer algorithms, the fan-out design pattern could be used to split the data and a fan-in design pattern could be used to combine the results.

Chapter 9, Idioms and Anti-Patterns

Question 1

What is the alternative to Java's `try`-with-resources in Kotlin?

Answer

In Kotlin, the `use()` function works on the `Closeable` interface to make sure that resources are released after use.

Question 2

What are the different options for handling nulls in Kotlin?

Answer

There are multiple options to handle nulls: the Elvis operator, smart casts, and the `let` and `run` scope functions can help with that.

Question 3

Which problem can be solved by reified generics?

Answer

On JVM, types are erased at runtime. By inlining the generic function body into the call site, it allows preservation of the actual types used by the compiler.

Chapter 10, Concurrent Microservices with Ktor

Question 1

How are the Ktor applications structured and what are their benefits?

Answer

Ktor applications are divided into modules, each module being an extension function on the `Application` object. Modularizing our application allows us to test different aspects of it separately.

Question 2

What are plugins in Ktor and what are they used for?

Answer

Plugins are a way Ktor addresses cross-cutting concerns. They are used for serializing and deserializing requests and responses, and setting headers, and even routing itself is a plugin.

Question 3

What is the main problem that the `Exposed` library solves?

Answer

The `Exposed` library provides a higher-level API for working with databases.

Chapter 11, Reactive Microservices with Vert.x

Question 1

What's a verticle in Vert.x?

Answer

A verticle is a lightweight actor that allows us to separate our business logic into small reactive units.

Question 2

What's the goal of the Event Bus in Vert.x?

Answer

The Event Bus allows verticles to communicate with each other indirectly by sending and consuming messages.

Question 3

Why shouldn't we block the event loop?

Answer

The event loop uses a limited number of threads to process many requests concurrently. If even one of the threads is blocked, it reduces the performance of a Vert.x app.

Other Books You May Enjoy

- Understand the Android development lifecycle
- Simplify app development with Google architecture components
- Use standard libraries for dependency injection and data parsing
- Apply the repository pattern to retrieve data from outside sources
- Publish your app on the Google Play store

Packt is searching for authors like you

Share Your Thoughts