Daniel Westheide

Scala from Scratch

Scala from Scratch: Exploration

Daniel Westheide

This book is for sale at http://leanpub.com/scala-from-scratch-exploration

This version was published on 2020-07-28

Contents

Credits

All text and source code examples (unless specified otherwise) by Daniel Westheide

This book uses three fonts:

- The body font is *Noto Serif* (licensed under the SIL Open Font License v1.10, available at https://www.google.com/get/noto/#serif-lgc)
- The title font is *Noto Sans* (licensed under the SIL Open Font License v1.10, available at https://www.google.com/get/noto/#sans-lgc)
- The code font is *Source Code Pro* (licensed under the SIL Open Font License 1.1, available at https://github.com/adobe-fonts/source-code-pro)

Cover illustration

Cover illustration by Ann-Marie Rechter (Illumarie Art)

Website: https://www.ann-marierechter.de
Twitter: https://twitter.com/illumarie
Instagram: https://instagram.com/Illumarie_Art

The font for the book title and author name on the cover is *Alegreya Sans* (licensed under the SIL Open Font License v1.10, available at https://fonts.adobe.com/fonts/alegreya-sans).

Preface

In the last few years, Scala and a bunch of technologies based on it have gained a lot of traction. Some of these technologies became crucial for dealing with big and fast data. Others paved the way when it comes to building resilient, scalable applications. As a consequence, Scala became very popular in startups living on the bleeding edge. In the wake of this, even big, conservative enterprises have been adopting it more and more.

Scala stands on the shoulders of the Java Virtual Machine (JVM). Developers can integrate with the vast pool of great libraries the Java ecosystem has to offer. Since shortly after its inception, it has been leading the way on the JVM when it comes to combining the object-oriented and functional programming paradigm. We have seen Java adopt functional language features that have proven to be a great success in Scala. Also, new languages like Kotlin emerged that take a lot of inspiration from Scala.

While Java and others have been following suit at their own pace, Scala continues to be the forerunner on the JVM. It provides one of the most sophisticated type systems in mainstream programming languages. As such, it's a great tool for writing safe, concurrency-friendly code that is easy to test, in a succinct way.

Scala originated on the JVM, but these days it's available for two alternative targets as well: Scala.js[1] compiles to JavaScript, where as Scala Native[2] allows you to write native applications, using LLVM[3].

This book is the first one in the *Scala from Scratch* series. It will give you a strong foundation and a comprehensive overview of what Scala brings to the table. The goal is to get you excited about the language and to give you an idea of what it's like to work with Scala. By the end of the book, you should be able to write small Scala programs on your own. We'll show how to do that by walking through a small, but complete example application.

If you are hungry for more, you can continue your journey in the second book, *Scala from Scratch: Understanding*[4]. It will explain many of the concepts introduced in

[1] https://www.scala-js.org/
[2] https://www.scala-native.org
[3] https://llvm.org/
[4] At the time of writing, *Scala from Scratch: Understanding*, is still in development. The goal is to have it released in the summer of 2020.

this book in more depth, and it will cover concepts and language features that are too advanced for an introductory book.

While Scala.js and Scala Native have gained a bit of traction, this book and the follow-up book focus solely on Scala for the JVM, which is where Scala is most popular.

Audience

This book is mainly targeted at people who are already somewhat familiar with an object-oriented and imperative language like Java, Ruby, or Python — whether you are a student, a software engineer, or a data scientist.

You should know a thing or two about the tools in your language's ecosystem. Yet you don't have to be an experienced software engineer. We're using the command-line. Some basic familiarity with how to execute commands in your operating system's command-line interface would be nice.

Experience with some typical object-oriented design patterns is helpful, but not required.

You needn't know anything about functional programming before reading this book. All you need is an open mind and the willingness to leave your comfort zone and think outside the box.

How to read this book

I wrote this book as a text book. It's best to read it from the first to the last chapter. This is by no means a language reference. However, it should be easy for you to get back to any chapter or section in the future if you want to refresh your knowledge on a specific topic.

Exercises

You'll get the most out of this book by trying out all the examples that I use to explain new concepts. Besides, many, but not all the sections in this book contain a few exercises that you can work on by yourself. If you do, you'll get new insights about what you have learned. Nothing in the book requires for you to have tackled a previous exercise. Even so, it's recommended to spend some time with these exercises before moving on to the next section.

Typographical conventions

This book makes use of the following typographical conventions:

- *Italic* text is used when introducing a new term, as well as for emphasis
- `Constant width` text is used for program listings, for references to code elements, file and directory names, and for shell commands or program output
- In program listings, a `$` at the beginning of the line tells you that you are in a shell in your terminal, for example bash or zsh

Feedback

If you spot any typos, factual errors, or other problems, please take the time to inform me about it. The best way to do that is the LeanPub feedback form[5].

If you like this book, I would be grateful if you could spread the word. Please tell your friends, share it on Twitter or other social media channels, or publish a review. A great place to do that is the book's GoodReads page[6].

About the author

Daniel Westheide is a software engineer living in Berlin, Germany. He is a senior consultant at INNOQ[7] and co-organiser of ScalaBridge Berlin, the Berlin chapter of the ScalaBridge[8] organisation. He cares about empathy and inclusivity, and about the ethical, social, and ecological consequences of his work.

On his website[9], he discusses functional programming, architecture, as well as anything related to the software development process. He is not only interested in programming languages, but also a human language enthusiast. Moreover, he is passionate about specialty coffee, social science fiction, and tabletop roleplaying games.

[5]https://leanpub.com/scala-from-scratch-exploration/email_author/new
[6]https://www.goodreads.com/book/show/49129385-scala-from-scratch
[7]https://innoq.com
[8]https://scalabridge.org/
[9]https://danielwestheide.com

Acknowledgements

Thanks a lot to Lars Hupel, Stefan Lauer, Timo Loist, Martin Otten, Lara Pourabdolrahim, Nicolas Rinaudo, Tina Schönborn, Jan Stępień and everyone I have forgotten for giving me valuable feedback, proofreading, and being my rubber duck when I needed one.

Thanks to my employer INNOQ, specifically Phillip Ghadir, Stefan Tilkov, and Oliver Wolf, for encouraging me and supporting me in writing this book.

Thanks to Ann-Marie Rechter for creating such a wonderful cover illustration. Seeing the finished cover motivated me to finally finish this project.

Lastly, I would like to express my deep gratitude to Saskia, my wonderful wife, for always supporting me in this lengthy endeavour, for her many suggestions, and for dragging me out of the pit when I was filled with doubts and wanted to throw all the unfinished work away and just stop.

1. With a REPL yell

When learning a new programming language, there are always a few things that you need to know early on. Those things are the very foundation that everything else you do when you write code is based on. For example, you usually want to get some understanding of the language's syntax. You also want to get to know the standard types and operators available to you.

In this chapter, you'll learn these and other fundamentals. Together, they constitute the basic building blocks you'll learn to assemble in the following chapters.

1.1 What's a REPL?

Like many modern programming languages, Scala ships with an interactive shell, a *read-eval-print loop (REPL)*. If you have never used a REPL before, the mere name will probably not tell you a lot about what a REPL actually does. So let's go through the for elements the name is composed of:

- **read:** It reads an expression you type in with your keyboard from the standard input
- **eval:** It evaluates the expression it has read
- **print:** It prints the result of evaluating the expression to the standard output
- **loop:** After printing the result of one evaluated expression, it allows you to type in another expression

Having a REPL, you can quickly try out snippets of code and see the result of executing it, without having to write a complete program or writing a test. The output provides you with a very fast feedback loop, which comes in handy when you're just experimenting or playing around with a library you're not familiar with — but it's also helpful when you start learning a new programming language.

Having a Scala REPL also means that I don't have to bore you with a dull reference chapter listing all the standard types and explaining the most important syntax. Instead, you're going to get familiar with these basic elements in a very interactive way.

1.2 Install fest

Installing a JDK

If you haven't already done so before, you may first need to install a Java Development Kit (JDK), because Scala runs on the Java Virtual Machine (JVM). This means that the Scala compiler compiles Scala source files to Java byte code, just as the Java compiler does.

All the examples in this book have been tested with OpenJDK 11.0.8. At the time of writing, this was the latest release with long-term support (LTS). However, these days, things are moving quite quickly, and any JDK, starting from a recent version in the Java 8 series, should work without problems. However, I recommend to use JDK 11 as well, because it's known to work with the tools that I recommend to use for this book.

Fairly recent versions of OpenJDK are available in all popular package managers — for example in *Homebrew* for OS X, *Chocolatey* for Windows, *apt* for Debian- or Ubuntu-based Linux distributions, or *yum* for RedHat or Fedora Linux.

If you prefer to manually download and install the version of the OpenJDK we are using in this book, OpenJDK builds are available from AdoptOpenJDK[1]. They also provide DEB and RPM repositories that you can add to your package manager in a lot of Linux distributions. For Homebrew, there is also a package called `adoptopenjdk@11`.

You can see if Java has been installed, and which version it is, by typing the following command into a terminal:

```
$ javac --version
javac 11.0.8
```

Here, and in all future code listing, the `$` at the beginning of a line marks the prompt of your shell. You might see a different symbol there, depending on how your shell is configured. In any case, the prompt symbol (`$` in this book), is not to be typed into the terminal — only what's coming after that.

If you type `javac --version`, the JDK version available on your system will be printed to the console. In this example, it's version 11.0.8. If Java is not installed, or it's not

[1] https://adoptopenjdk.net/

in the path, you will get an error message saying that the `javac` command could not be found.

Installing sbt

Before you can get started, you also need to install the *Scala Build Tool (sbt)*. sbt is an interactive tool for building your Scala projects. It's also possible to use other build tools for the JVM, for example Maven or Gradle. Moreover, a few new Scala build tools have emerged recently, most notably Mill[2] and Fury[3]. However, right now, sbt is the de-facto standard in the Scala community, so we're going to use it in this book.

For an up-to-date explanation of how to install sbt and set it up on your operating system, please visit the sbt website[4].

sbt is also available in package managers like Homebrew and Chocolatey, if you prefer to install it that way, under the name `sbt`.

You can find out if sbt has been installed and is in your path, by typing `sbt --script-version` into your terminal, like so:

```
$ sbt --script-version
1.3.13
```

This will print the version of sbt you have installed to the console. If sbt has not been installed successfully, you will see an error message stating that the command `sbt` could not be found.

In this book, we use sbt 1.3.13, but any later version in the 1.x should be fine as well.

Once you have installed and set up Java and sbt successfully, you should be able to create a new sbt project. An sbt project is a directory containing all the Scala source files for a particular application or library you develop. sbt allows you to build your application or library and run tests — and it allows you to start a Scala REPL. This is exactly what we will do shortly, once you have created your first sbt project.

In order to create an sbt project, you need to use the `new` command from a command-line terminal. The command expects you to provide the name of an

[2]https://www.lihaoyi.com/mill/
[3]https://fury.build/
[4]https://www.scala-sbt.org/download.html

sbt project template. One such template that I created specifically for this book is `dwestheide/minimal-scala-project.g8`[5].

So, without further ado, please fire up a command-line terminal, go to a directory in which you want to save all of your Scala learning material, and create a new sbt project based on the template I provided:

```
$ sbt new dwestheide/minimal-scala-project.g8
[info] Loading global plugins from /home/daniel/.sbt/1.0/plugins
[info] Set current project to private (in build file:/home/daniel/)
[info] Set current project to private (in build file:/home/daniel/)
name [my-awesome-scala-project]: repl-yell

Template applied in /home/daniel/./repl-yell
```

sbt will ask us to provide a name for our project, which we decided will be `repl-yell`. Next, we're going to change into the newly created directory of the same name. This directory is the root of our `repl-yell` sbt project using the `cd` command:

```
$ cd repl-yell
```

With the exception of creating a new project, you always want to execute sbt from the root directory of an sbt project.

sbt allows you to launch a Scala REPL by means of the `console` command. The first time you use this command, you'll need to be a bit patient, because sbt needs to download quite a lot of stuff from the internet. Any subsequent usages of `sbt console` should be a lot faster and look similar to this:

[5]sbt project templates are based on a templating project called *Giter8*. Giter8 templates are provided as GitHub repositories, so `dwestheide/minimal-scala-project.g8` points to a GitHub repository called `minimal-scala-project.g8` by a user `dwestheide` (that's me). You can find out more about creating sbt projects from Giter8 templates in the sbt documentation: https://www.scala-sbt.org/1.0/docs/sbt-new-and-Templates.html

```
$ sbt console
[info] Loading global plugins from /home/daniel/.sbt/1.0/plugins
[info] Loading project definition from /home/daniel/repl-yell/project
[info] Loading settings for project repl-yell from build.sbt ...
[info] Set current project to repl-yell (in build file:/home/daniel/repl-yell/)
[info] Starting scala interpreter...
Welcome to Scala 2.13.3 (OpenJDK 64-Bit Server VM, Java 11.0.8).
Type in expressions for evaluation. Or try :help.

scala>
```

From now on, whenever you see `scala>` in a code example, we are in the REPL. Never copy and paste the `scala>` part from those examples into your own REPL, just what's coming after that. Otherwise, the Scala REPL will not be able to make any sense of what you pasted in there and show you an error message.

No one likes to read instructions, so we will not bother with typing `:help` for now. But remember that this command is available in the REPL. For now, the most important command you need to know when you're in the REPL is `:quit`, which will quit the REPL and get you back to wherever you came from. If you are lazy, you can also use its short version, `:q`.

1.3 Doing the math

You could say that the Scala REPL is a calculator with a command-line interface. Of course, it is much more than that, but let's start by pretending it is just that.

Our first expression

First, let's see if it's smart enough to do some basic maths:

```
scala> 3 + 5 * 4
val res0: Int = 23
```

Apparently, it is. So what can we learn from this first piece of Scala code we have written?

Operators

Addition and multiplication operators work exactly as you are probably used to from languages like Java, Ruby, or Python. Just as you may be used to from these or other languages, or from maths, multiplication takes precedence over addition.

Types

The result of adding and multiplying whole numbers is of type `Int`. `Int` is Scala's built-in integer type.

Syntax

The syntax for specifying the type `A` of a value `x` is `x: A`. This is different from Java, where you would prefix the name of the value with the type, like so:

```
int res0 = 23; // Scala's Int translates to lower-case int in Java
```

It's also different from languages like Ruby or Python, where you wouldn't specify a type at all. However, this notation might be familiar to you if you have previously worked with languages like Kotlin, TypeScript, or Rust, for instance.

Unlike in Java or most C-like languages, you don't need to end a line with a semicolon. In that regard, Scala is more similar to Ruby or Python than it is to Java.

Result identifiers

In the REPL, the result of an expression you type in is not only printed back to the console — in addition to that, it's also bound to an identifier. Since this was the first expression we typed into the REPL in our session, the result value has the identifier `res0`.

More maths

You can access the results of the expressions you type into the REPL in your following expressions:

```
scala> (res0 - 2) / 5
val res1: Int = 4
```

The subtraction and division operators are familiar as well, just as the usage of parentheses to determine the precedence of operators. Just as in Java, Ruby, or Python, dividing an `Int` by another `Int` gives you the integer quotient, maintaining the `Int` type.

If you want to get the remainder, the modulo operator `%` is available:

```
scala> (res0 - 2) % 5
val res2: Int = 1
```

In order to get a floating point result, one of the numbers needs to be a floating point number. For literal number values, this is done by appending an `f` to them. Let's try this out:

```
scala> (res0 - 2) / 5f
val res3: Float = 4.2
```

The result of this expression is `4.2`, and its type is `Float`. `Float` is the built-in number type for floating-point numbers.

If you want double precision instead, append `d` to the literal number value instead of `f`. Here is an example:

```
scala> (res0 - 2) / 5d
val res4: Double = 4.2
```

The result of the expression is now of type `Double`, the built-in type for double-precision numbers. This is also the type Scala chooses by default for floating point literals:

```
scala> 5.5
val res5: Double = 5.5
```

Exercises

1. You learned how to create a `Float` value from an integer literal. Can you also turn a floating point literal from a `Double` into a `Float`?
2. Another built-in type for whole numbers in Scala is `Long`. Try converting an integer literal into a value of type `Long`, following the approach for `Double` and `Float`.

1.4 Greetings from Mr. Boole

Now that you know about the basic numeric types available in Scala, let's quickly delve into the realm of boolean logic. If you come from Java, Scala doesn't come with any surprises. We begin by using the familiar comparison operators:

```
scala> 5 > 4
val res6: Boolean = true

scala> 5 < 4
val res7: Boolean = false

scala> 5 >= 5
val res8: Boolean = true

scala> 5 <= 4
val res9: Boolean = false
```

The result is always of type `Boolean`, a type of which there are exactly two possible values: `true` and `false`.

We can use the standard operators on boolean expressions that you might know from other languages:

```
scala> 5 > 4 && 5 < 4
val res10: Boolean = false

scala> 5 < 4 || 5 > 4
val res11: Boolean = true

scala> 5 > 4 && 5 >= 5
val res12: Boolean = true

scala> 5 < 4 || 5 <= 4
val res13: Boolean = false

scala> !res12
val res13: Boolean = false
```

Exercises

1. Explore the precedence of the boolean operators by experimenting in the REPL.
2. Find out how strict boolean operators in Scala are. Do they consider certain values, like `0` or `1`, to be falsy or truthy, or do they only accept values of type `Boolean`?

1.5 Values and variables

So far, you have mostly written expressions using literal values and a few result values from previous expressions. Sooner or later, you'll want to be able to define your own values and variables.

What is the difference between values and variables anyway? Variables allow for reassigning, effectively binding their identifier to a different value. Values, on the other hand, can only be defined once, not permitting any reassignment. In that regard they are similar to a `final` variable definition in Java.

You already came across the `identifier: Type = value` syntax when the REPL displayed the result of an expression you typed in. To define a value, you can use the same syntax, but you start with the `val` keyword. In the example below, we define a value with the identifier `theAnswer` and the type `Int` and bind it to the integer value `42`. You can access this value in any following expressions in the same REPL session:

```
scala> val theAnswer: Int = 42
val theAnswer: Int = 42

scala> theAnswer * 4 > 100
val res14: Boolean = true
```

Please note that defining a value is not an expression, but a statement. This means defining a value does not return anything, and hence, it doesn't lead to a new result value in the REPL. Instead, when you define a value in the REPL, it displays that value, as you can see in the example above.

It's good practice to use *lowerCamelCase* for identifiers. You can use all kinds of funny characters, including the whole range of characters from the Unicode Basic Multilingual Plane[6]. For now, sticking to alphanumeric camel-cased identifiers is a reasonable strategy.

Defining a variable works the same, except for the keyword, which is `var` instead of `val`. Reassigning to an existing variable is straightforward as well, following the usual `name = value` syntax:

```
scala> var theAnswer: Int = 0
var theAnswer: Int = 0

scala> theAnswer = 42
// mutated theAnswer
```

You have probably noticed that defining immutable values doesn't require more boilerplate than defining mutable variables. This is intentional. In fact, as you will see in Chapter 2, Scala favours immutability. Once you have adopted a functional mindset, you will rarely need to work with variables in practice.

You may wonder why we were able to define a variable with the name `theAnswer` although we have already defined a value with that name in the same REPL session. One special feature of the REPL is that you can *redefine* any value or variable identifier. This is great for experimenting interactively. In normal Scala programs, this is not possible.

[6]https://www.sttmedia.com/unicode-basiclingualplane

1.6 The thin line between compile time and runtime

While Scala is not an interpreted language, it may seem so when you use the REPL, where the expressions you type in are immediately evaluated. Nevertheless, every piece of code you send to the REPL is actually first compiled to Java byte code. Only after that that the compiled code is executed. Keep that in mind, especially when reasoning about errors. A lot of errors that would be runtime errors in an interactive Ruby shell, for example, are compile-time errors in Scala.

As the lines between compilation and execution are somewhat blurred in the REPL, you might mistake one kind of error for the other in the beginning. Once we start writing our code in files and explicitly compiling it, the differences will become a lot clearer.

Exercises

1. Define a new value of type `Int` in the REPL and, after that, try reassigning a new integer literal to it.
2. Earlier in this chapter, it was mentioned that results are stored as values, not as variables. Verify this by reassigning to `res14` from the example above, changing its value from `true` to `false`.
3. Find out what happens if you define a value of one of the types covered so far and assign a literal of a different type to it.
4. Are the errors you encountered in these exercises runtime errors or compile-time errors?

1.7 Local type inference

If you come from a dynamically typed programming language, or if you want to learn Scala because you have grown tired of all the boilerplate code you have to write in some other statically typed languages, you may wonder if it's necessary to specify types when defining values or variables.

It is not. The Scala compiler features a local type inference mechanism. This often allows it to infer the type of an expression. Later throughout the book, we will see how this type inference mechanism works in more complex scenarios, what its limitations are, and when you should not rely on it. First, however, let's see how it works in the simplest of cases, when defining values:

```
scala> val x = 5
val x: Int = 5

scala> val y = 3.0
val y: Double = 3.0

scala> val z = x + y
val z: Double = 8.0

scala> val b = true
val b: Boolean = true
```

Nice, the Scala compiler was able to infer the correct types for us. Without us explicitly providing type information in our code, it was able to infer that `x` is an `Int`, `y` is a `Double`, `z` is a `Double`, and `b` is a `Boolean` value.

1.8 Calling methods

You will find that apart from the types we have seen so far, there are plenty of others available to you in Scala by default. Which those are, and the mechanism by which they are made available automatically, will be explained in Chapter 3.

For now, let's content ourselves with the `String` type, which is just an alias for Java's `String` class. It is a good example for demonstrating how to call methods defined on existing classes, whether they are Java or Scala classes.

Java's `String` class has a lot of useful methods defined on it. For example, if we want to find the index of the first occurrence of a specific character in a string, we can call the `indexOf` method. Let's experiment with this and some other methods:

```
scala> val screamedGreeting = "HELLO WORLD!!!!"
val screamedGreeting: String = HELLO WORLD!!!!

scala> val firstBangPos = screamedGreeting.indexOf('!')
val firstBangPos: Int = 11

scala> val bang = screamedGreeting.charAt(firstBangPos)
val bang: Char = !

scala> val friendlyGreeting =
     | screamedGreeting.toLowerCase.replaceAll("!", "")
friendlyGreeting: String = hello world

scala> val chars = friendlyGreeting.toCharArray
val chars: Array[Char] = Array(h, e, l, l, o,  , w, o, r, l, d)

scala> val bytes = "hello".getBytes("UTF-8")
val bytes: Array[Byte] = Array(104, 101, 108, 108, 111)
```

Calling these methods looks exactly like calling methods in Java and many other object-oriented languages, using the common dot notation between the instance and the method name and parentheses enclosing the argument list. If the argument list has more than one argument, each argument is separated by a comma, as the `replaceAll` example demonstrates. If, on the other hand, a Java method has an empty argument list, it is not only possible, but also customary, to leave out the empty parentheses in Scala. Examples of this in the code above are `toLowerCase` and `toCharArray`.

Please note that in the `replaceAll` example, our expression spans two lines. If you press *RETURN* before finishing an expression in the REPL, you can continue it in the next line. The REPL will display a `|` character on each new line. If you paste this code snippet into your REPL, make sure not to copy the `|`. This is merely displayed by the REPL and not valid Scala syntax.

Also, in this code example, we have seen a bunch of new types. `Char` is the built-in type for characters. When you call a Java method returning a `char[]` or `byte[]`, these get represented as an `Array[Char]` or `Array[Byte]`, respectively.

In general, any Java arrays are represented by an `Array` in Scala. `Array` is a generic type, taking one type parameter. An `Array[Byte]` contains elements of type `Byte`, while an `Array[Char]` contains elements of type `Char`. You will learn more about type parameters and generic types in Chapter 7. For know, just read these as *array of* *characters* and *array of bytes*, respectively.

1.9 Summary

In this chapter, you were introduced to the Scala REPL, an interactive shell that makes it easy to experiment and try out some code. You also learned about some of the basic types available in Scala, some common operators you will need in everyday Scala programming, and how to call existing methods, using the `String` type as an example.

In the next chapter, you will get to know a few concepts of the functional programming paradigm as well as some of Scala's language constructs that have been shaped by these ideas.

2. Scala, the functional language

In the previous chapter, you have been writing short expressions in which you called methods defined on built-in types, used operators, and accessed values and variables you previously defined. This will not get you very far if you want to write anything resembling a real-world application. You'll need to make control-flow decisions and assemble certain functionality into reusable units of code.

In this chapter, you will learn how to do all of that. At the same time, you will get an understanding of some of the core principles of the functional programming paradigm, and how it has shaped the Scala language.

2.1 The purely functional way

Let's assume we want to define a mathematical function that, when given the lengths of the two catheti, returns the area of a right-angled triangle:

$$f(a, b) = \frac{a \cdot b}{2}$$

Mathematical functions such as this have some interesting properties:

First of all, for the same arguments *a* and *b*, evaluating *f* will *always* yield the same result. For example, *f(5, 4)* will always be *10*:

$$f(5, 4) = 10$$

Secondly, because of that, it is safe to *substitute* a function application with its result:

$$f(5, f(4, 3)) = f(5, 6)$$

Finally, this example also shows that it's straightforward to *compose* more complex mathematical expressions from simpler ones consisting of function applications.

Functional programming is all about carrying those desirable properties of mathematical functions into the world of programming. The three properties described above mean that we can use a divide-and-conquer approach in order to reason about what a complex mathematical function returns for a given input value. It also means that we can compose several simple functions in order to create a more complex one.

Wouldn't it be great if we could reason about the behaviour of a computer program in the same straightforward way? And if we could assemble new programs by composing a few simple existing programs? This is what functional programming is all about.

Before we delve deeper into that, however, let's see how we can define the Scala equivalent of the mathematical function above.

Defining a Scala function

Here is how we define this as a Scala function in the REPL:

```
scala> def areaOfRightTriangle(a: Int, b: Int): Double = a * b / 2.0
def areaOfRightTriangle(a: Int, b: Int): Double
```

After defining a function in the REPL, its name and type will be displayed, just as you already know from value and variable definitions.

Depending on what language you come from, the way we define a function in Scala may look strange to you. If you compare this to the mathematical function definition above, though, there is not such a big difference, and with good reason.

Here is what we can learn about defining functions in Scala from the example above:

1. A function definition starts with the `def` keyword.
2. It is followed by the name of the function. The established convention is to use `lowerCamelCase` names.
3. After the name, the comma-separated parameter list follows. The parameter type is specified after the name, separated by a colon, just as the left side of a value definition.
4. Next up, we need to specify the return type, following the syntax you already know from value definitions.
5. Just as for mathematical functions, we have to provide an expression on the right side of the `=` character that returns a value matching the specified return type. No need to `return` anything.

Deterministic behaviour

Applying this function to different input values yields different results, whereas applying it multiple times to the same input values always yields exactly the same result:

```
scala> areaOfRightTriangle(5, 4)
val res0: Double = 10.0

scala> areaOfRightTriangle(5, 5)
val res1: Double = 12.5

scala> areaOfRightTriangle(5, 5)
val res2: Double = 12.5
```

This is exactly the behaviour you would expect from a mathematical function, and also from a Scala function like this. The function we defined is *deterministic*.

No side effects

We just saw that the `areaOfRightTriangle` function is deterministic: For the same arguments, it always returns the same value. One important aspect of this is this: Computing a return value from the given arguments is all it does. It does not modify any mutable application state or introduce randomness, nor does it interact with the outside world, for example with a local file system or with a remote database server. In other words, it does not have any *side effects*.

Imperative programming is all about side effects. This is done by executing *statements*: First do this, then do that. Such a statement usually mutates the application's internal state or interacts with the outside world. In contrast, functional programming is all about *expressions*.

Purity

A function that is both deterministic *and* free of effects is called a *pure function*. Since `areaOfRightTriangle` has both of these properties, it is indeed a pure function.

Pure functions have one important property, which is known as *referential transparency*. This means that, at any time, you can replace a pure expression (an expression in which only pure functions are called) with the value the expression evaluates to — without changing the behaviour of the program containing the expression.

The nice thing about this is that purity and referential transparency compose: If a function *a* applies a function *b* that does not perform any side-effects itself, it, too, is pure and referentially transparent:

```
scala> def areaOfRightTriangle(a: Int, b: Int): Double = a * b / 2.0
def areaOfRightTriangle(a: Int, b: Int): Double

scala> def isLarge(area: Double): Boolean = area > 100
def isLarge(area: Double): Boolean

scala> def isLargeTriangle(a: Int, b: Int): Boolean =
     | isLarge(areaOfRightTriangle(a, b))
def isLargeTriangle(a: Int, b: Int): Boolean
```

Here, `isLargeTriangle` is composed of two pure functions. If we can compose our programs of pure functions, which in turn are also composed of pure functions, this means that we can use a divide-and-conquer approach at reasoning about our program, substituting certain expressions with their results. Moreover, pure functions are also a lot easier to test than effectful method calls on mutable objects.

Immutability

There is one important prerequisite for referential transparency. All the data structures and objects we pass to or return from such a function must be immutable. It would not be possible to use substitution of expressions in a mathematical function otherwise. Imagine the consequences if a `3` passed as an argument to a function could change into a `4` or another number at any time.

This is why we usually work with *immutable data structures* in Scala. Scala's lists, for example, are immutable, and the only way to "change" them is to create a new list that reflects the intended change. This is in stark contrast to imperative programming. In this paradigm, we change lists and other collections in-place, using statements.

Immutable data structures are not as expensive as it sounds. Since lists are immutable, it's safe to have the new list share the common parts with the old version of the list. We don't have to create a complete copy. This principle is called *structural* *sharing*. It’s used in the implementation of many Scala collection types, not only lists.

There is an extra benefit to immutable data structures and objects. Since they cannot change, they are safe to pass around even to other threads. This is why functional programming is also a great fit for concurrent and parallel programming.

Separating logic from effects

At this point, you may argue that any real-world application is a lot more complex than the Scala function I have shown. Also, you might point out that it’s impossible to create such an application from pure functions alone. And yes, any meaningful application has to perform effects — be it reading from a file, writing to the network, or printing to the console.

In functional programming, we do not pretend that the necessity for effects does not exist. Rather, we aim to isolate those parts of the code that need to perform effects from the ones that implement business logic as pure functions. If we take this further, we can define pure functions that return *descriptions* of effects. These descriptions will then be interpreted to actually perform these effects.

We’re going to revisit this principle in a more real-world scenario in the case study presented in Chapter 10. There are functional languages that enforce purity. You’ll notice that unlike those languages, Scala takes a rather pragmatic stance.

Before we move on, though, it’s time to take a closer look at how to write functions in Scala again. The functions we have defined so far have only shown you a part of what you need to know about it.

2.2 More about writing functions

The example in the previous section already showed you a lot about how to write functions in Scala. Nevertheless, there are a few other things you should know about.

Type inference

First of all, you may wonder if it’s possible to leave out the return type of a function, just as you can leave out the type of a value. The answer is yes. It’s perfectly possible to define the function from the previous section like so:

```
scala> def areaOfRightTriangle(a: Int, b: Int) = a * b / 2.0
def areaOfRightTriangle(a: Int, b: Int): Double
```

This time, the compiler needs to infer the return type, and the result is the same as before: We defined a function that takes two `Int`s and returns a `Double`. It's *not* possible to leave out the types of the parameters, `a` and `b` in this example.

Also, now that you know that you *can* leave out the return type, the question is whether you *should*. The answer is "usually not". Your default choice should be to always specify a return type. Firstly, this helps readers of your code. Secondly, there are cases where the compiler will infer a different type from the one you actually intended. Moreover, it's often helpful to think about the types and the signature of a function first, before thinking about its implementation. Seeing the types can guide your implementation — and the compiler can tell you if if your implementation doesn't match signature you defined, helping you to spot problems in your code early on.

Functions without a parameter list

So far, you have only seen functions that take parameters. It's also possible to define functions that don't take any parameters. For instance, if two `Int` values `a` and `b` have already been defined in the REPL session, we can define a function that accesses these two values. We can then call such a function just by typing its name into the REPL, without any parameter list:

```
scala> val a = 3
val a: Int = 3

scala> val b = 5
val b: Int = 5

scala> def areaOfRightTriangle: Double = a * b / 2.0
def areaOfRightTriangle: Double

scala> val area = areaOfRightTriangle
val area: Double = 7.5
```

In a real application, `a`, `b` and the function might be defined in the same class instead. We will discuss these object-oriented features of Scala in the next chapter.

Functions with an empty parameter list

It’s also possible to define functions with an empty parameter list. What is the difference between that and a function with no parameter list? Technically, none at all, apart from the way you apply them. However, it is often assumed that a function without a parameter list is pure. When defined in a class, for example, we think of it as a field or property.

In contrast, some people prefer to use an empty parameter list for functions that do have a side effect.

Consider the following example:

```
scala> def uuid(): String = java.util.UUID.randomUUID().toString
def uuid(): String

scala> val id1 = uuid()
val id1: String = bf050c2e-9383-48a6-8c5e-83a4d207a143

scala> val id2 = uuid()
val id2: String = 2185b3a2-9af7-4c66-9839-8780525dba5a
```

Here, we have defined a function `uuid()` with an empty parameter list and call it twice, each time with an empty argument list. This function delegates to `java.util.UUID` to generate a new, random UUID and then returns its `String` representation. The whole purpose of this function is to generate a new value every time it is called. As such, this is *not* a pure or referentially transparent function.

Some Scala developers take a pragmatic approach of annotating impure functions like `uuid()` with an empty parameter list. This merely serves as a visual hint that applying the function twice will likely not yield the same value.

Others prefer to encode such impure behaviour on the type level. While Scala’s type system does not force you to implement your functions in a pure way, there are techniques for representing effects such as these in the return type of a function. This, however, goes far beyond the scope of this book, let alone this introductory chapter.

Multi-line expressions

As you already know, the right side of a function definition has to be an expression. It may surprise you to learn that quite often,you will be able to break down your problem into tiny functions that fit into a single line.

However, not all functions can and should be written in a single line. Sometimes, your code becomes more readable if you assign the result of an expression to a named value first. Then, you can use that value in a following expression that will compute the value returned by your function.

As an example, consider a function that calculates the area for any triangle for which the lengths of its three sides are known, instead of the more specific right-angled triangle area we implemented before:

```
def triangleArea(a: Int, b: Int, c: Int): Double = {
  val s = (a + b + c) / 2.0
  Math.sqrt(s * (s - a) * (s - b) * (s - c))
}
```

Here, the right side of the function definition is actually a *code block*, enclosed by curly braces. Such a code block can contain an arbitrary number of statements, but it must end with an expression, making the whole code block one multi-line expression. Any value or variable defined in a code block is local to that block. This means that it's not visible from outside that block. For such a local value or variable, it's common practice and perfectly fine to not specify a type explicitly and let the Scala compiler infer it for you.

In the example, the first line of the block is an assignment statement to a local value `s`. In the last line, that local value `s` is used multiple times in an expression that determines the value of the whole multi-line expression.

Working with Scala source files

It's a bit cumbersome to write functions consisting of multiple lines in the REPL. This is why we'll now start to define our functions in Scala source files. The `repl-yell` directory looks like this:

```
.
├── build.sbt
├── project
│   └── build.properties
└── src
    ├── main
    │   └── scala
    └── test
        └── scala
```

This is a typical directory structure for an sbt project. For now, we ignore most of the directories and files in there. The only one we care about is the `src/main/scala` directory. This is where sbt expects us to put our Scala source code.

Text editors

If you want to create and modify Scala source files, you'll need some tool for doing that. There is Scala syntax highlighting support and more for most popular text editors, for example Visual Studio Code, Atom, Vim, or Emacs. If you are not already a believer in one of the other editors, I recommend installing Visual Studio Code[1] as well as the Scala (Metals) extension[2]. Scala Metals[3] is available for other text editors as well. What you get is syntax highlighting, code completions, go-to-definition, and a few other neat features.

I don't recommend using a full-blown IDE at this point in the learning curve, unless you're already used to working with an IDE. If you are a long-time user of IntelliJ IDEA[4], for example, there is a really good Scala plugin[5]. However, in this book, we assume a developer workflow of editing source code in a text editor and using sbt from the command line to compile it and try it out.

Top-level definitions

Now, please use your preferred text editor and save the following into a file called `Triangle.scala` in the `src/main/scala` directory of the `repl-yell` sbt project:

1 https://code.visualstudio.com/Download
2 https://marketplace.visualstudio.com/items?itemName=scalameta.metals
3 https://scalameta.org/metals/
4 https://www.jetbrains.com/idea/
5 https://plugins.jetbrains.com/plugin/1347-scala

```
object Triangle {
  def triangleArea(a: Int, b: Int, c: Int): Double = {
    val s = (a + b + c) / 2.0
    Math.sqrt(s * (s - a) * (s - b) * (s - c))
  }
}
```

You'll see a new keyword here, `object`. The thing is that function or value definitions cannot be at the top-level in a Scala source file. Instead, they have to be defined in a class or an object. You will learn a lot more about classes and objects in the following chapter. For now, you can think of objects as a mechanism to define modules, allowing you to put related functions into the same module, and unrelated ones in separate modules.

Source files and sbt

When you start a Scala REPL from sbt, using `sbt console`, all the Scala source files in `src/main/scala` will be compiled. This means that all objects and the functions defined in them are now available in the REPL:

```
scala> val area = Triangle.triangleArea(4, 5, 5)
val area: Double = 9.16515138991168
```

Be aware though that once you have started a Scala REPL this way, it won't pick up any changes you make to the source code or any new source files in the `src/main/scala` directory. For that to happen, you will have to quit the REPL by typing `:q` and start it again using sbt's `console` command. This can be a bit cumbersome, so we'll try to keep the need for doing this to a minimum. Soon, we're going to explore other techniques of testing the behaviour of your functions.

The Unit type

While you should always strive to keep your functions pure and referentially transparent, you already saw that it is possible to violate this principle in Scala by introducing side effects into functions whose signature indicates no such thing.

Scala also has the notion of purely side-effecting functions. These are functions that do not return anything meaningful — their sole purpose is the execution of an effect. For example, Scala provides a function called `println` that prints the given `String` to the standard output:

```
scala> println("Hello world!")
Hello world!
```

Let's store the result of applying this function in a value:

```
scala> val result = println("Hello world!")
Hello world!
val result: Unit = ()
```

Apparently, `println` returns a value of type `Unit`, so that its signature is the following:

```scala
def println(s: String): Unit
```

The concrete value returned by `println` is `()`. It turns out that `()` is the only possible value of type `Unit`.

2.3 If-expressions

Since functional programming is based on expressions instead of statements, many of the built-in control structures are also expressions. One such control structure you will probably not be able to do without is the *if-expression*.

To illustrate how if-expressions are written in Scala, imagine you need to implement a pluralization function that, given a text in singular form and a number of units, returns a pluralised text. Here are two example function applications and their expected results to illustrate the desired functionality:

- `pluralise("new comment", 1) -> "1 new comment"`
- `pluralise("new comment", 3) -> "3 new comments"`

Writing tests

Illustrating the desired functionality of a function in terms of examples is pretty much what happens when we write automated tests in the traditional way. Scala does not have a test framework built into the standard library. However, there are numerous third-party test libraries or frameworks. One of the most minimalistic ones is called, well, Minitest[6]. If you have created your sbt project from

[6]https://github.com/monix/minitest

the `minimal-scala-project` template (see Section 1.2), this test library is already available in your project.

This means that we can translate the textual examples from above into automatically testable code examples, taking the shape of a Minitest test suite. Let's add a file called `StringsTest.scala` to the `src/test/scala` directory of our sbt project. The file should look like this:

```scala
import minitest._

object StringsTest extends SimpleTestSuite {

  test("pluralise one comment") {
    assertEquals(Strings.pluralise("new comment", 1), "1 new comment")
  }

  test("pluralise two comments") {
    assertEquals(Strings.pluralise("new comment", 2), "2 new comments")
  }
}
```

The test cases and their assertions, written as Scala code, don't look too differently from the textual examples. However, there are a few new things here. For now, you'll have to take them for what they are. But rest assured that they will be explained later in the book:

1. We use Scala's `import` keyword to import everything contained in the `minitest` package. Packages are a means of modularising your code. Both packages and imports will be explained in Chapter 5.
2. Minitest test suites need to be defined as objects and they must inherit from Minitest's `SimpleTestSuite`. This is where the `object` keyword and the `extends` keyword come into play. Objects and inheritance will be explained in more detail in Chapter 3.
3. `SimpleTestSuite` provides a function called `test` for specifying your test cases. The description of the test needs to be provided as a `String`. This is followed by a code block (one or more lines in curly braces). In this code block, you can place the implementation of the respective test case. In Chapter 8, you will learn more about what is happening here and how you can write your own functions that can be used in a similar way.

4. Minitest provides a few functions for specifiying assertions. For now, we stick to `assertEquals`, which expects two arguments: the first one is the actual result, the second one the expected result.

Providing a skeleton for the function under test

Right now, our test would neither succeed nor fail — since Scala is a statically-typed language, it wouldn't even compile. We need to add a `Strings` object with a `pluralise` function to our main Scala sources. Let's do that by adding a file `Strings.scala` to the `src/main/scala` directory:

```
object Strings {
  def pluralise(singular: String, n: Int): String = "not-implemented"
}
```

This should satisfy the compiler. We have provided a skeleton for the `pluralise` function that has the signature expected by our test cases.

Using sbt interactively

What we probably want to do now is execute our tests and check that the `StringsTest` indeed fails. After that, we want to change the implementation of our `pluralise` function and run the tests again. Rinse and repeat, until our tests are all green *and* we are satisfied with the implementation of our function under test.

So far, we have only used sbt in a non-interactive way: We provided the `console` command to sbt directly on the command-line. Once we quit the REPL started by the command, this stopped the whole sbt process as well, and we were back in the shell.

When you are in this test-implement-refactor loop, you want that loop to be fast. That's why, during development, sbt is usually used in an interactive mode. The interactive sbt shell is started if you don't provide any commands to sbt:

```
$ sbt
[info] Loading settings for project global-plugins from metals.sbt ...
[info] Loading global plugins from /home/daniel/.sbt/1.0/plugins
[info] Loading project definition from /home/daniel/repl-yell/project
[info] Loading settings for project repl-yell from build.sbt ...
[info] Set current project to repl-yell (in build file:/home/daniel/repl-yell/)
[info] sbt server started at local:///home/daniel/.sbt/1.0/server/
fd3038e4e69787d4ae29/sock
sbt:repl-yell>
```

You know you are in the interactive sbt shell when you see a prompt that shows the name of your sbt project — in our case, the prompt is `sbt:repl-yell`. In this interactive sbt shell, you can launch sbt commands directly. After running the commands, you're still in the sbt shell, so there is no overhead of starting sbt again and again. For example, you could type `console` in the sbt shell, quit the REPL with `:q` and you'd be back in the interactive sbt shell. To quit sbt itself, you can type in the `exit` command.

Running tests

In order to run all the tests in our sbt project, we an use sbt's `test` command:

```
sbt:repl-yell> test
[info] Compiling 1 Scala source to /home/daniel/projects/repl-yell/target/
scala-2.13/classes ...
StringsTest
- pluralise one comment *** FAILED ***
  received not-implemented != expected 1 new comment (StringsTest.scala:6)
  ...
- pluralise two comments *** FAILED ***
  received not-implemented != expected 1 new comment (StringsTest.scala:10)
  ...
```

Since we haven't actually implemented our `pluralise` function yet, all the tests we wrote fail — no surprise there. For each failing test case, you will see a helpful error message, followed by a long stacktrace, omitted in the example above.

sbt also allows you to run only specific tests, instead of all of them, using the `testOnly` command. To only execute the `StringsTest`, try this:

```
sbt:repl-yell> testOnly StringsTest
...
```

We can also run only tests matching a pattern. To run all tests starting with `Str` and ending with `est`, use the following command:

```
sbt:repl-yell> testOnly Str*est
```

The sbt interactive shell is great because it provides tab completions for most of the commands and their arguments. Just try it out.

If you want an even faster feedback loop, you can even have sbt run your tests or a specific test automatically as soon as a source file changes: Just add the tilde symbol (~) before the command, and sbt will watch for file changes and automatically re-trigger the respective command. To continuously execute the `StringsTest`, for example, you would execute the following:

```
sbt:repl-yell> ~testOnly StringsTest
...
[info] 1. Monitoring source files for updates...
[info] Project: repl-yell
[info] Command: testOnly StringsTest
[info] Options:
[info]   <enter>: return to the shell
[info]   'r': repeat the current command
[info]   'x': exit sbt
```

If you press the `ENTER` key, sbt will stop watching for file changes and return to the interactive shell.

Implementing our function

Let's get back to our `pluralise` function, and to the topic of if expressions. A crude implementation of this function that satisfies our two test cases, looks like this:

```
def pluralise(singular: String, n: Int): String = {
  val text = if (n > 1) singular + "s" else singular
  "%d %s".format(n, text)
}
```

Here is what you can learn from the example:

- The if-expression has the form *if (cond) e1 else e2*, where *cond* is a boolean expression, *e1* is the expression evaluated if *cond* is true, and *e2* is the expression evaluated if *cond* is false.
- The boolean expression *cond* must be placed in parentheses.
- The value of the if-expression is either *e1* or *e2*.

The string formatting syntax you see here is the one used by Java's extensively documented[7] `Formatter`.

Exercises

1. Can the expressions *e1* and *e2* in an if-expression be multi-line expressions? Come up with an example and try it out.
2. Use the REPL to find out what the result type of an if-expression is in which you return a `String` in the `if` branch, and an `Int` in the `else` branch. What does this tell you about Scala's type inference mechanism?
3. Is it possible to leave out the `else` branch? What is the result type of the if-expression `if (6 > 4) "yes"`? What is the result type of `if (6 > 4) println("yes")`? Do you have an explanation? Again, use the REPL to explore this.

2.4 Functions as values

So far, our journey into Scala has brought us to the point where we know how to implement functions that are pure and referentially transparent — an important principle of functional programming. However, Scala wouldn't be much of a functional programming language if it didn't have the notion of functions as values.

[7] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Formatter.html

The idea behind this is that you can treat functions the same way you would, for example, numbers or strings: You can assign them to variables or values, and you can pass them to another function as a parameter, or have one function return another function.

Defining functions values

Let's see what Scala has to offer in this regard. First of all, we want to define a function that checks if a given integer number is even or not. We can define a function providing this functionality the way you have learned earlier in this chapter:

```scala
def isEven(x: Int): Boolean = x % 2 == 0
```

This function, however, is not a value. Functions that can be passed around as values are defined in a slightly different manner. Let's start a new REPL session and have a look at how we can implement the same logic as a *function literal*, which we then bind to the `isEven` identifier:

```
scala> val isEven: (Int) => Boolean = (x: Int) => x % 2 == 0
val isEven: Int => Boolean = $$Lambda$4183/0x0000000802726840@2ef9017
```

What you see here is the least succinct way of writing a function literal, but if you want to understand the shorter alternatives that are about to follow, it helps a lot to start from here.

What happens in the example above is that we are defining a new value, just as we have done before with integer or string literals. In this case, the identifier of our value is `isEven`. We chose to be explicit about the type of this value: Since this is supposed to be a function that receives an `Int` and returns a `Boolean`, we annotated our value with the type `(Int) => Boolean`.

Next up, on the right side, we need to provide a value of type `(Int) => Boolean`. Here, Scala's function literal syntax comes into play. A function consists of two parts: the parameter list in parentheses, and an expression that evaluates to a value matching the return type of the function. These two parts are always separated by the `=>` symbol you have already seen in the type, `Int => Boolean`.

If you look at the type that the REPL printed for the value `isEven`, you will see that it shows it as `Int => Boolean`. If a parameter list consists of only one parameter, the canonical representation of its type leaves out the parentheses. We will shortly see what the type of a function with multiple parameters looks like.

You may wonder why there are two different syntaxes for defining functions — function literals on the one hand, and the `def` syntax on the other. Well, while functions created in these two ways behave quite similar in many ways, there are some differences in how they are represented in the byte code the compiler produces. However, those differences are not important for now, and we will get back to them in Chapter 3.

Applying a function value

Before we move on, you probably want to know what you can do with a function value like `isEven`. Well, you can apply it, just as you would apply mathematical functions:

```
scala> val no = isEven(3)
val no: Boolean = false

scala> val yes = isEven(2)
val yes: Boolean = true
```

You have probably noticed that the syntax for applying function values is not only similar to mathematical function application, but also to how we apply functions defined with the `def` keyword in Scala. In fact, it looks exactly the same. For all practical purposes, this means that all the functions we have defined in this chapter could be written as function literals instead.

Type inference and syntactic sugar

The `isEven` function definition above is a bit on the verbose side. You'd probably be disappointed if I told you that this is how you have to write function literals in Scala, and it won't get any shorter. Luckily for you, that's not the case. Once again, Scala's type inference mechanism can help us to keep our code succinct. In addition, there is some syntactic sugar that can make your function literals even shorter. Let's explore some alternative ways to write our `isEven` function:

```
scala> val isEven2: Int => Boolean = (x) => x % 2 == 0
val isEven2: Int => Boolean = $$Lambda$1052/469465633@607e8fc1

scala> val isEven3: Int => Boolean = x => x % 2 == 0
val isEven3: Int => Boolean = $$Lambda$1061/426125293@53f77ac7

scala> val isEven4: Int => Boolean = _ % 2 == 0
val isEven4: Int => Boolean = $$Lambda$1062/60293496@2253fdb9

scala> val isEven5 = (x: Int) => x % 2 == 0
val isEven5: Int => Boolean = $$Lambda$1063/1113306007@12306904
```

These are the key observations:

1. Since the type of `isEven2` is known to be `Int => Boolean`, the Scala compiler can infer the types of the parameters of our functions.
2. If a function only has a single parameter, and its type can be inferred, we can leave out the parentheses around the parameter list (`isEven3`).
3. If the type of our function is known, and we only access each function parameter a single time in our function implementation, we can leave out the parameter list completely. Every underscore in our function body represents one function parameter, starting from left to right. We will see more examples of this later, with functions that take more than one parameter. For `isEven4`, this means that the compiler infers that _ is the `Int` parameter that the function expects, since its type is known to be `Int => Boolean`.
4. If the type of the function value is not explicitly set on the left side of a function definition, the types of the parameters must be specified. Even if the function only takes a single parameter, if we specify that parameter's type, the parameter list must be surrounded with parentheses. The Scala compiler can now infer the type of the `isEven5` value to be `Int => Boolean`.

Ultimately, which of these options you choose when defining a function value is up to you. Each of them is certainly better than the original `isEven`, which contains redundant type annotations. In general, people leave out the parentheses if they are not necessary, so you will rarely see function values like `isEven2` in practice. Some developers find an abundance of underscores difficult to read and argue that something like `isEven4` is virtually too succinct. Since it's preferable to be explicit about the type of a value or variable, unless the value or variable is local, `isEven3` is usually a better choice than `isEven5`.

Let's add a new Scala source file now in order to create a version of `isEven` that survives the end of our current REPL session. In the `src/main/scala` directory of your sbt project, add a file `Predicates.scala` that looks like this:

```scala
object Predicates {
  val isEven: Int => Boolean = _ % 2 == 0
}
```

Passing function values around

Now, if all we did with function values was to apply them, there wouldn't be any benefit over functions defined with the `def` syntax. As I already hinted at, having functions as values means that you can do everything with them that you can do with other values. This specifically includes passing them to other functions that expect such a function value as a parameter.

Consider the function literal again that we assigned to the `isEven` identifier. What if we want to check for a pair of numbers whether both of them are even? We could write a function literal that reuses the `isEven` function and bind it to the `areEven` identifier:

```scala
object Predicates {
  val isEven: Int => Boolean = _ % 2 == 0
  val areEven: (Int, Int) => Boolean = (x, y) => isEven(x) && isEven(y)
}
```

This is an example of a function literal with two parameters, so the parameters in its type are enclosed in parentheses: `(Int, Int) => Boolean`.

This is all fine and well. However, what if we want to have a more generic version? Let's try to define a function that works for any given condition *cond* and checks if that condition is met for two integer values? To achieve this, we can write a function `forBoth` that takes two `Int` parameters and a third parameter whose type is `Int => Boolean`. Let's add this to the `Predicates` object as well:

```
object Predicates {
  // omitted what's already defined in the Predicates object
  def forBoth(x: Int, y: Int, cond: Int => Boolean): Boolean =
    cond(x) && cond(y)
}
```

Since the type of the function literal we bound to the `isEven` identifier is `Int => Boolean`, we can pass it to the `forBoth` function, like so:

```
scala> val yes = Predicates.forBoth(2, 4, Predicates.isEven)
yes: Boolean = true

scala> val no = Predicates.forBoth(2, 3, Predicates.isEven)
no: Boolean = false
```

It's not necessary to first assign function literals to a value identifier. Instead, we can define the function literal when applying the `forBoth` function:

```
scala> val no = Predicates.forBoth(2, 3, x => x % 2 != 0)
val no: Boolean = false
```

Here, we want to check if both integers passed to `forBoth` are odd. Often, when a function is short and you do not plan to use it again in the same unit of code, this is a good choice.

Exercises

1. Try out different notations for the function literal we passed to `forBoth`, leaving or adding type annotations, and making use of the underscore shortcut notation.
2. Write a function literal with this signature: `(Int, Int, Int => Boolean) => Boolean`. It should implement the same logic as the `forBoth` function we defined with the `def` syntax.
3. Write a function with this signature: `def acceptOrElse(x: Int, default: Int, cond: Int => Boolean): Int`. It should return `x` if `cond(x)` is `true`, otherwise returning `default`.
4. In a new file `PredicatesTest.scala` in the `src/test/scala` directory, write tests for all the functions and methods you defined in the `Predicates` object. Use examples of input and expected output values, trying to cover all the possible paths in the tested functions.

2.5 Escaping the loop

So far, you have already seen a few principles of functional programming, like the focus on expressions and referential transparency, and the ability to treat functions as values. Another cornerstone of functional programming is recursion.

To understand why you need recursion and what you need to know to use it in Scala, let's try to implement a function that calculates the fibonacci number for a given natural number *n*. An informal specification looks like this:

- if *n* = *0*: *0*
- if *n* = *1*: *1*
- if *n* = *2*: *1*
- otherwise: *fibonacci(n - 1)* + *fibonacci(n - 2)*

What's interesting about this is that the definition of the fibonacci number is recursive. In other words, it's defined in terms of itself. Nevertheless, if you are coming from an imperative language, you will probably be used to implementing problems like this using a loop construct. Scala allows you to do that as well.

Let's create a file `Fibonacci.scala` in the `src/main/scala` directory of our `repl-yell` sbt project, containing an object called `Fibonacci`. In that object, we're going to add a first implementation of the fibonacci function:

```scala
object Fibonacci {
  def fib1(n: Long): Long = {
    var previous = 0L
    var current  = 1L
    var result   = previous
    var i        = 1L
    while (i < n) {
      i += 1
      result = previous + current
      previous = current
      current = result
    }
    result
  }
}
```

The syntax of the while loop looks similar to what you may be used to from other languages. In practice, you will rarely need it, once you have grown accustomed to a more functional style of programming. We use `Long` as the type of our input and output values because fibonacci numbers can grow large quite quickly.

So what is the problem with this implementation? After all, `fib1` does not violate referential transparency, right? All the mutable state is purely local and never exposed to the outside world. That is indeed true. If you don't break referential transparency, using local mutable state inside of a function can be perfectly fine. In fact, it's a good choice if it leads to at least one of the following outcomes:

- It makes your code easier to read
- It solves a performance problem in a critical path

The main problem with `fib1` is that it is focussed a lot on the low-level details of *how* to calculate the fibonacci number, instead of describing *what* is to be calculated. It is certainly not easier to read than a recursive implementation. In order to see the two alternatives next to each other, let's add the following function to the `Fibonacci` object as well:

```scala
def fib2(n: Long): Long =
  if (n > 1) fib2(n - 1) + fib2(n - 2)
  else n
```

I think this is much more readable than `fib1`. In fact, it is almost as if we are reading the specification — we are describing what we want to do, instead of getting lost in implementation details like switching variables around and incrementing a counter. One of the big benefits of functional programming is exactly that you tend to describe what you want to do instead of how to arrive there. As a result, the intent behind your code becomes clearer than in imperative programming.

Measuring runtime performance

Let's see if we can still argue in favour of the imperative implementation in `fib1` for performance reasons. For that purpose, we are going to rely on the notation of functions as values, which you learned about earlier in this chapter. In the `Fibonacci` object, we're going to define a function that expects a function from `Long` to `Long` and returns a function of the same type. It merely acts as a wrapper for the passed-in function, doing two things:

1. measuring how long it takes to execute the function
2. printing the execution time to the standard output

We'll call this function `timed`, and this is what it looks like:

```scala
def timed(f: Long => Long): Long => Long = x => {
  val start   = System.nanoTime
  val result  = f(x)
  val end     = System.nanoTime
  val elapsed = (end - start) / 1000000.0
  println("Elapsed time: " + elapsed + " milliseconds")
  result
}
```

The type of the input parameter is `Long => Long`, and the return type is the same. The implementation returns a new function literal that takes one parameter of type `Long`, which we call `x`. It uses a Java method called `System.nanoTime` to get the current time before and after applying `x` to the passed-in original function `f`. After logging the elapsed time, the result of applying `f` to `x` is returned.

Please note that we're not doing any rock-solid performance benchmarking here. However, our `timed` function is good enough to see the notable performance difference between a few different fibonacci implementations.

Let's use `timed` to see if our recursive function `fib2` can compete with `fib1`, which is implemented with a loop:

```
scala> val timedFib1 = Fibonacci.timed(x => Fibonacci.fib1(x))
val timedFib1: Long => Long =
  Fibonacci$$$Lambda$4756/0x000000080192f840@16d98ee5

scala> val timedFib2 = Fibonacci.timed(x => Fibonacci.fib2(x))
val timedFib2: Long => Long =
  Fibonacci$$$Lambda$4756/0x000000080192f840@350f6fb

scala> val x = timedFib1(50)
Elapsed time: 0.017437 milliseconds
val x: Long = 12586269025

scala> val y = timedFib2(50)
Elapsed time: 41889.603047 milliseconds
val y: Long = 12586269025
```

Oh no! That doesn't look good. The problem with recursion is that we are putting a lot of stack frames onto the call stack — `fib2(50)` cannot complete before `fib2(48)` and `fib2(49)` have completed, and these, in turn, need to wait for `fib2(46)` and `fib2(47)`, or `fib(47)` and `fib(48)`, respectively. And so on. This is not only bad for performance, but can also lead to stack overflow errors at runtime.

Tail-recursion to the rescue!

Luckily, we are back in the game if we can rewrite our recursive implementation using *tail recursion*, which means that the recursive call is the last thing that happens in the function. If that is the case, the Scala compiler will automatically optimise the generated byte code and turn the whole thing into a loop for us — without us having to write and think in terms of loops.

A tail-recursive version of our fibonacci function is shown below, and you're invited to add that to the `Fibonacci` object as well:

```
def fib3(n: Long): Long = {
  def iterate(previous: Long, current: Long, n: Long): Long =
    if (n == 0) current
    else iterate(current, previous + current, n - 1)
  if (n > 1) iterate(1, 1, n - 2) else n
}
```

Here, we are making use of a powerful feature of Scala — the possibility to define functions inside of other functions. This is not necessary for correctness reasons, but it allows us to hide the recursive helper function. A function defined inside of another one is never visible to the outside, which means it will not become a part of your API.

Please note that for recursive functions, an explicit return type is always required by the Scala compiler.

While this tail-recursive implementation doesn't read exactly like the informal specification, it is still much closer to it than the imperative version that relied on the while loop.

To verify that this function is as fast as the imperative version, let's time it as well. Make sure to start a new REPL session from the interactive sbt shell, using the `console` command.

```
scala> val timedFib3 = Fibonacci.timed(x => Fibonacci.fib3(x))
val timedFib3: Long => Long =
  Fibonacci$$$Lambda$4827/0x0000000800ffa040@6ec21f37

scala> val x = timedFib3(50)
Elapsed time: 0.010558 milliseconds
val x: Long = 12586269025
```

Apparently, the Scala compiler indeed optimised our tail-recursive function.

If your intention is to write a tail-recursive function, you can make sure at compile-time that this is actually the case by adding an appropriate annotation to your function:

```
def fib(n: Long): Long = {
  @scala.annotation.tailrec
  def iter(previous: Long, current: Long, n: Long): Long =
    if (n == 0) current
    else iter(current, previous + current, n - 1)
  if (n > 1) iter(1, 1, n - 2) else n
}
```

This annotation will not change your byte code. The tail-call optimization will happen regardless of this annotation. The sole benefit is that it gives you a compile error if the annotated function is *not* tail-recursive.

Exercises

1. Add the `tailrec` annotation to the `fib1` function in your `fibonacci.scala` file and load it into the REPL. Does the compiler complain?
2. Define a tail-recursive function `factorial(n: Long): Long` that calculates the factorial of a given positive integer. See https://en.wikipedia.org/wiki/Factorial for the specification. Make sure at compile-time that your function is tail-recursive and experiment with it in the REPL.

2.6 Summary

In this chapter, you got a first glimpse at Scala's functional side. You learned about pure functions and referential transparency, how to define functions and how to use some of the built-in control structures, which are all implemented as expressions instead of statements. In addition, you learned about an essential building block of functional programming, and how to use it in Scala: functions as values, using Scala's function literal syntax.

Being able to treat functions as values is what allows us to create abstractions that are generic and reusable. These abstractions enable us to write highly expressive and declarative code that reveals its intent, instead of having readers get lost in the low-level steps taken to achieve it.

In the next chapter, we're going to explore the most important aspects of Scala's object-oriented side, and you'll see that object-oriented and functional programming don't have to be at odds with each other.

3. Scala, the object-oriented language

Scala is a hybrid language combining two paradigms: functional and object-oriented programming. After a brief introduction to Scala's functional side, we are now going to look at Scala's most important object-oriented aspects. While some of them may seem familiar to you from other object-oriented languages, a closer look reveals a few differences.

Before we start, let's create a new sbt project called `boardgame` based on the same template we have been using when we created our `repl-yell` project:

```
$ sbt new dwestheide/minimal-scala-project.g8
[info] Loading global plugins from /home/daniel/.sbt/1.0/plugins
[info] Set current project to private (in build file:/home/daniel/)
[info] Set current project to private (in build file:/home/daniel/)
name [my-awesome-scala-project]: boardgame

Template applied in /home/daniel/./boardgame

$ cd boardgame
```

We will put all of the example code for this chapter into this project.

3.1 Classes and objects

Just like in most object-oriented languages, objects in Scala are instances of classes. In other words, the latter form some kind of template from which concrete objects can be instantiated.

Let's say we want to implement a digital version of a board game. In most board games, players can move around meeples[1] on the board, small wooden figures representing characters in the game. Let's say that in our game the board consists of twenty times twenty square fields. A meeple can move up to five steps in a single turn. Movement is only possible horizontally and vertically, not diagonally.

[1] https://www.happymeeple.com/en/media/what-is-a-meeple/

We should probably start with a `Position` class representing the concept of a position on the game board.

Defining a class in Scala requires very little boilerplate. In fact, a minimal definition of a Scala class looks like this:

```
class Position
```

Let's put this into a file called `Position.scala` in the `src/main/scala` directory. Having defined the class, we can create an instance of it. To try this out, we are going to start a Scala REPL from our interactive sbt shell, using sbt's `console` command:

```
scala> val position = new Position
val position: Position = Position@73ee04c8
```

Admittedly, this class, and our new instance of it, are pretty useless at the moment. They have neither state nor behaviour. Nevertheless, there is a reason to start with such a minimalistic class definition, because there are already a few things we can learn from it:

1. Class definitions start with the `class` keyword followed by the name of the class.
2. The idiomatic way of naming classes is `UpperCamelCase`. However, this isn't enforced by the compiler.
3. You don't have to provide a class body or a constructor.
4. Classes without constructor arguments are instantiated with the `new` keyword followed by the name of the class.

Class parameters

A position has an x-coordinate and a y-coordinate. It would be nice if we could pass these as arguments to the constructor of the `Position` class. To allow that, we first need to add a list of *class parameters* to our `Position` class:

```
class Position(x: Int, y: Int)
```

The class parameter list is written directly after the name of the class. The Scala compiler will automatically create a constructor for us that takes the parameters from that list. However, the class parameters will not be visible outside this instance of the class.

To validate this, let's start a fresh REPL session and create an instance of the `Position` class:

```
scala> val position = new Position(1, 0)
val position: Position = Position@248c167b

scala> position.x
                ^
       error: value x is not a member of Position
```

We cannot access the x coordinate of the position. If we try, we will get a compile error. You'll see shortly how to make `x` visible to the outside.

The class body

Apart from a list of class parameters, a class can also have a body. A class body is a code block, which means it starts and ends with a pair of curly braces. You first learned about code blocks when we talked about multiline expressions in Section 2.2.

In a class body, you can:

1. Define methods
2. Define fields, also known as instance variables or values, using the `var` or `val` keyword, respectively.
3. Do anything else

So what can you do with class parameters? Every method, value, or variable defined in the class body has direct access to the class parameters. There is no need to assign them to instance variables or values beforehand. However, they cannot be reassigned, just like you cannot reassign function parameters.

Anything you do in the body of a class that is not a method definition will become part of the constructor, which means that it will be executed when an instance of the class is created.

For example, you could print the `x` and `y` class parameters to the console:

```scala
class Position(x: Int, y: Int) {
  println("New position: %d / %d".format(x, y))
}
```

If you create an instance of `Position` in a new REPL session, you will see that our log output gets printed to the standard output immediately:

```
scala> val pos = new Position(1, 4)
New position: 1 / 4
val pos: Position = Position@5a5b80f0
```

Enforcing invariants

A typical responsibility of a constructor in object-oriented programming is to enforce the invariants of a class. In Scala, you can do this in the class body. There's a helpful method called `require` that is automatically imported. It lets you pass in a `Boolean` condition, and if that condition is not satisfied, it will throw an `IllegalArgumentException`. Optionally, you can pass in an error message as a second argument. This is strongly recommended so that people understand what's going on.

Throwing an *exception* breaks the normal execution flow of your program. Somewhere in the call chain, the exception can be *caught* and handled. If that doesn't happen, the program will crash.

Before we use this method, let's specify our expected behaviour. Let's write a test to verify that we cannot create any instances that break our invariants. In the `src/test/scala` directory create a file called `PositionTest.scala` that looks like this:

```scala
import minitest._

object PositionTest extends SimpleTestSuite {
  test("Cannot create position with x < 0") {
    intercept[IllegalArgumentException] { new Position(-1, 0) }
    ()
  }
  test("Cannot create position with y < 0") {
    intercept[IllegalArgumentException] { new Position(0, -1) }
    ()
  }
  test("Cannot create position with x >= 20") {
    intercept[IllegalArgumentException] { new Position(20, 0) }
    ()
  }
  test("Cannot create position with y >= 20") {
    intercept[IllegalArgumentException] { new Position(0, 20) }
    ()
  }
}
```

Minitest lets you intercept any exceptions that you expect to be thrown. The type of the exception that is expected is a type parameter of the `intercept` method. You will learn more about type parameters in Chapter 7. For now, it's enough to know that you pass the concrete type in brackets. After that, we provide a code block that is supposed to throw the exception. The `intercept` method returns the intercepted exception type, but we need to return a value of type `Unit` in our tests. This is why we need to return the `()` value explicitly.

It's good practice to make sure that tests you have written actually fail, and only then start making them pass by implementing the function under test. If you run the `test` command in the interactive sbt shell, you should see something like this towards the end:

```
sbt:boardgame> test
...
[error] Failed: Total 4, Failed 4, Errors 0, Passed 0
[error] Failed tests:
[error]         PositionTest
[error] (Test / test) sbt.TestsFailedException: Tests unsuccessful
```

Now, let's finally use the `require` method to enforce our invariants and make our tests pass:

```
class Position(x: Int, y: Int) {
  require(x >= 0 && x < 20, "x was %d, expected >= 0 && < 20".format(x))
  require(y >= 0 && y < 20, "y was %d, expected >= 0 && < 20".format(y))
}
```

If you run the tests again, all of them should be green.

Please note that throwing exceptions is not a good practice in Scala, because it doesn't play well with purely functional programming. You will learn about an alternative way of enforcing your invariants at the end of Chapter 9. Until then, we are going to use exceptions in order to enforce the invariants of our classes and functions. However, we will restrict ourselves to using methods like `require` that are provided by the standard library and do the exception throwing for us.

Fields

For anyone to be able to do anything meaningful with a position, we need to provide fields that can be read from outside our class body. If we want to turn class parameters into fields, we just need to prefix our class parameters with the `val` keyword:

```
class Position(val x: Int, val y: Int) {
  require(x >= 0 && x < 20, "x was %d, expected >= 0 && < 20".format(x))
  require(y >= 0 && y < 20, "y was %d, expected >= 0 && < 20".format(y))
}
```

The default visibility of a `val` defined in a class is that it's *public*, which means that it can be read from any other class or package. You will learn more about visibility later in this book.

In a new REPL session, you will be able to access the `x` and `y` fields of a `Position` instance like so:

```
scala> val pos = new Position(1, 3)
val pos: Position = Position@2fafba77

scala> val x = pos.x
val x: Int = 1

scala> val y = pos.y
val y: Int = 3
```

Defining methods

In the previous chapter, I mentioned briefly that functions defined with the `def` syntax are quite different from function literals. While function literals get compiled to Java lambdas and can be passed around as values, the former ones get compiled to methods in the Java byte code — and you can't pass around methods the same way that you can pass around function values.

In fact, what we define with the `def` keyword is actually called a method in Scala. In the rest of the book, we will often refer to such functions as methods, especially when there is a need to distinguish them from function values.

What this means is that you already know how to define methods. When a method is defined in a class it has access to all the class parameters. For example, we may want to define a method called `steps` that expects another `Position` and returns the number of steps it takes to go there. Let's add such a method to our `Position` class:

```scala
class Position(val x: Int, val y: Int) {
  require(x >= 0 && x < 20, "x was %d, expected >= 0 && < 20".format(x))
  require(y >= 0 && y < 20, "y was %d, expected >= 0 && < 20".format(y))

  def steps(other: Position): Int = 0
}
```

Now, it's time to put our expectations into executable tests. In the `PositionTest` we will add the following test cases:

```scala
test("step from (0,0) to (10,5) == 15") {
  val pos1 = new Position(0, 0)
  val pos2 = new Position(10, 5)
  assertEquals(pos1.steps(pos2), 15)
}
test("step from (7,3) to (0, 0) == 10") {
  val pos1 = new Position(7, 3)
  val pos2 = new Position(0, 0)
  assertEquals(pos1.steps(pos2), 10)
}
```

As usual, these test cases should first fail when running the sbt `test` command. Here is an implementation of `steps` that will make our tests pass:

```scala
class Position(val x: Int, val y: Int) {
  require(x >= 0 && x < 20, "x was %d, expected >= 0 && < 20".format(x))
  require(y >= 0 && y < 20, "y was %d, expected >= 0 && < 20".format(y))

  def steps(other: Position): Int = {
    val xDiff = math.abs(other.x - x)
    val yDiff = math.abs(other.y - y)
    xDiff + yDiff
  }
}
```

Inheritance

In Scala, if a class does not extend another class explicitly, it will automatically extend `AnyRef`, which is an alias for Java's `java.lang.Object`, and is the super class at the top of the class hierarchy.

To extend another class explicitly, you need to use the `extends` keyword after the class parameter list. We haven't done that for our `Position` class, so it inherits from `AnyRef`. This is the same as if we wrote this:

```scala
class Position(val x: Int, y: Int) extends AnyRef
```

In general, inheritance is not that popular in Scala. Favouring composition over inheritance often leads to simpler solutions, especially when you take advantage of functions as values and the compositional nature of functions.

Overriding inherited methods or fields

Inheriting from a super class means that you have access to all its methods and fields, as long as they are not declared as `private` (we will discuss visibility modifiers a bit later in this chapter). It also means that you can override them.

To demonstrate this, let's override the `toString` method defined in `AnyRef`, or rather `java.lang.Object`.

To override a method defined in a parent class, you need to prefix it with the `override` modifier, like so:

```scala
class Position(val x: Int, val y: Int) {
  // rest of the class body omitted
  override def toString: String =
    "%s (x: %d, y: %d)".format(super.toString, x, y)
}
```

Here you can see how to access a field or method defined in the parent class. Just as in most object-oriented languages, you have to use the `super` keyword to reference the parent. In this example, we do this in order to make use of the already existing `toString` implementation in `AnyRef`.

With our custom `toString` implementation in place, when we create a `Position` instance in the Scala REPL it's now shown as something like this:

```
scala> val pos = new Position(3, 2)
val pos: Position = Position@28a2283d (x: 3, y: 2)
```

That is because the REPL calls `toString` on anything it needs to display. We can also call the `toString` method directly:

```
scala> val s = pos.toString
val s: String = Position@385d7101 (x: 3, y: 2)
```

The uniform access principle

Calling the `toString` method looks exactly the same as if our class had a field called `toString`. Instead of overriding `toString` as we did before, we can also do it like this:

```
class Position(val x: Int, val y: Int) {
  // rest of the class body omitted
  override def toString: String =
    "%s (x: %d, y: %d)".format(super.toString, x, y)
}
```

From the outside, this looks the same. To verify, please start a new REPL session after adjusting your `Position` class and try again:

```
scala> val pos = new Position(3, 2)
val pos: Position = Position@28a2283d (x: 3, y: 2)

scala> val s = pos.toString
val s: String = Position@385d7101 (x: 3, y: 2)
```

This is what we call the *uniform access principle*. Parameterless methods and values defined in a class are accessed in a uniform way, and you can consider both of them to be fields, or properties. The reason why this can work is that value definitions and method definitions in a class live in the same namespace.

Whether you want to use a `val` or a `def`, a value or a method, largely depends on how you expect that field to be used, and how expensive it is to compute the result. Implementing `toString` as a value means that the string formatting necessary to create the result of `toString` will happen immediately when an instance of the `Position` class is created. It will happen regardless of whether `toString` is ever used by anyone, on that `Position` instance. Implementing it as a `def` means that all this string concatenation will only happen if and when someone actually accesses `toString`. In this case, the resulting string will be recomputed every time someone accesses `toString`, even though the class is immutable — which means that the result of `toString` will always be the same for one instance of `Position`.

As always, it's a tradeoff. In a real program, you would usually not use a `val` to implement `toString`, but it is often a reasonable choice for other methods.

Abstract classes

Abstract classes are present in a lot of object-oriented languages. These cannot be instantiated but are meant to be inherited from by concrete classes. One use case for abstract classes in object-oriented programming is the template method pattern[2]. In a multiparadigm language like Scala, there is little need for this pattern. Nevertheless, since it might be familiar to you from other object-oriented languages, it serves us well as an example for learning about how Scala approaches abstract classes and a few other object-oriented language features later on.

Let's say our game needs various types of meeples, and each of them has different rules for how many steps they are allowed to move in a single turn. To support this, we want to define an abstract super class called `Meeple`. Let's create a new file called `Meeple.scala` next to the `Position.scala` file and add our `Meeple` class to it:

```scala
abstract class Meeple(val id: Long, val position: Position) {
  def move(newPosition: Position): Meeple = {
    val requiredSteps = position.steps(newPosition)
    require(isValidStepCount(requiredSteps))
    withNewPosition(newPosition)
  }
  def isValidStepCount(steps: Int): Boolean
  def withNewPosition(pos: Position): Meeple
}
```

This class is prefixed with the `abstract` keyword to indicate that this is an abstract class that cannot be instantiated. It declares three methods, of which it only implements one, the `move` method. This method does a bit of orchestration, delegating to the other methods in a certain order. The remaining two are *abstract methods* and need to be implemented by concrete subclasses. Moreover, our abstract super class

[2]https://en.wikipedia.org/wiki/Template_method_pattern

has a list of two class parameters. Let's see how a subclass can inherit from this abstract class by adding a `RegularMeeple` class to the same file:

```
class RegularMeeple(id: Long, position: Position)
    extends Meeple(id, position) {
  override def isValidStepCount(steps: Int): Boolean =
    steps > 0 && steps <= 5
  override def withNewPosition(pos: Position): Meeple =
    new RegularMeeple(id, pos)
}
```

The example demonstrates that we can pass along parameters to the parent class we extend.

While not enforced by the compiler, it is strongly recommended to use the `override` keyword not only when overriding a method that is already implemented by the parent class, but also when implementing an abstract method. The reason why this is better is that the compiler can make sure that you actually implement an existing abstract method. If you accidentally introduce a typo into your method name or use a different method signature, the compiler will be your friend and inform you about this problem — if you have used the `override` keyword.

Final classes, fields, and methods

There is still one problem with our abstract `Meeple` class. While only the two abstract methods are meant to be implemented by subclasses, nothing prevents subclasses from also overriding its `move` method. Allowing for inheritance requires that we also design for it[3]. We want to provide some well-defined hooks where subclasses can take control of the behaviour of our abstract class, but we don't want them to override our `move` method.

In order to prevent a field or method from being overridden by subclasses, we have to use the `final` modifier. Let's change our abstract `Meeple` accordingly:

[3] Bloch, J. (2017): Effective Java. Addison-Wesley Professional. See the item "Design and document for inheritance or else prohibit it".

```
abstract class Meeple(id: Long, position: Position) {
  final def move(newPosition: Position): Meeple = {
    val requiredSteps = position.steps(newPosition)
    require(isValidStepCount(requiredSteps))
    withNewPosition(newPosition)
  }
  def isValidStepCount(steps: Int): Boolean
  def withNewPosition(pos: Position): Meeple
}
```

Now, we can be sure that subclasses will only use the provided hooks to customise the behaviour of the `move` method.

If you want to prohibit inheritance, because your class is not designed for that, put the `final` modifier in front of the `class` keyword. For instance, if we want to prevent anyone from creating a subclass of `RegularMeeple`, we can write it like this:

```
final class RegularMeeple(id: Long, position: Position)
    extends Meeple(id, position) {
  override def isValidStepCount(steps: Int): Boolean =
    steps > 0 && steps <= 5
  override def withNewPosition(pos: Position): Meeple =
    new RegularMeeple(id, pos)
}
```

If anyone attempts to extend `RegularMeeple` now, this will result in a compile error.

It's generally a good idea to make all your concrete classes final. Designing for inheritance is really difficult for classes that are not abstract, so it's best to prevent people from inheriting from your concrete classes altogether.

Please note that, unlike in Java, there is no reason to annotate your class parameters, function parameters, or local values inside of a function body with the `final` keyword, and the compiler won't even allow it. In Java, the `final` keyword in this context prevents reassigning to the respective variable or parameter. In Scala, there is no way to reassign to class parameters, function arguments and locally defined values (only to variables, defined with the `var` keyword). Here, `final` only prevents subclassing or overriding a defined field or method.

Exercises

1. Try out what happens if you change the name of the `isValidStepCount` method in the `RegularMeeple` class. What happens if you remove the `override` keyword, and why?
2. Implement a second subclass of `Meeple`, the `King`. It can move zero or one steps.

3.2 Singleton objects

You have probably not noticed it yet, but Scala classes do not allow you to define static methods or fields (also known as class methods or fields in some languages). All the methods and fields we defined so far in our classes were instance methods or fields. Unlike in Java, for example, there is no keyword that makes a method or field static.

If you want to define something with similar semantics to those of a static method or field, you cannot put it into a class. Instead, Scala provides built-in support for singleton objects, which means that only a single instance of the respective type will exist. There are two main differences between singleton objects and classes with static members. Firstly, singleton objects can extend or override methods or fields from parent classes. Secondly, we can pass around the singleton object like any other value — which you can't do with a class.

In our board game, we want to have meeples with different colors. As a first step, we create a new file `Color.scala` in which we define a `Color` class:

```
class Color(val red: Int, val green: Int, val blue: Int) {
  require(red >= 0 && red < 256)
  require(green >= 0 && green < 256)
  require(blue >= 0 && blue < 256)
}
```

Let's say you want to define some static constants of common colors. You can define a singleton object by using the `object` keyword. Let's create a new file called `Colors.scala` containing a `Colors` singleton object:

```
object Colors {
  val White: Color = new Color(255, 255, 255)
  val Black: Color = new Color(0, 0, 0)
  val Red: Color   = new Color(255, 0, 0)
  val Green: Color = new Color(0, 255, 0)
  val Blue: Color  = new Color(0, 0, 255)
}
```

Please note that we our color value identifiers start with an uppercase letter. The convention in Scala is to use upper camel case for *constants*. A constant is an immutable value assigned to a `val` in a singleton object. If the `Color` class were mutable, `Colors` were a class, or `White` were a `def`, we would use lower camel case and define it as `white`.

Now, in a new REPL session, you can access the fields defined on the `Colors` singleton object like so:

```
scala> val blue = Colors.Blue
val blue: Color = Color@7d1fd65e
```

This looks pretty much the same as accessing a static field of a class in other languages.

Evaluation semantics

Scala's singleton objects are instantiated lazily. To illustrate that, let's print something to the standard output in the body of our `Colors` object:

```
object Colors {
  println("initialising common colors")
  val White: Color = new Color(255, 255, 255)
  val Black: Color = new Color(0, 0, 0)
  val Red: Color   = new Color(255, 0, 0)
  val Green: Color = new Color(0, 255, 0)
  val Blue: Color  = new Color(0, 0, 255)
}
```

If you re-start the REPL, you won't see anything printed to the standard output just yet. As soon as you access the `Colors` object for the first time, though, it will be initialised and you will see something printed to the standard output. Accessing the object after that will not print anything again, because the object has already been initialised. Let's try this out in the REPL:

```
scala> val colors = Colors
initialising common colors
val colors: Colors.type = Colors$@4e060c41

scala> val blue = colors.Blue
val blue: Color = Color@61da01e6
```

This lazy initialization is pretty similar to how the singleton pattern is often implemented in Java.

The type of a singleton object

As we have seen in the REPL output above, the type of the expression `Colors` is not `Colors`, but `Colors.type`, and the `toString` value in that REPL session was `Colors$@4e060c41`. That last part is the object id and will be a different one every time you start a new Scala REPL.

What can we learn from this? For one, `Colors` itself is not a type, or a class, it is an instance of a type. Also, while there can be an arbitrary number of values of type `Meeple`, there can only be exactly one value of type `Colors.type`, and that value is bound to the name `Colors`. Because of this, `Colors.type` is called a *singleton type*.

Since `Colors` itself is not a type, you cannot define a function with the following signature:

```
def pickColor(colors: Colors): Color
```

You *can* define a function with this signature, though:

```
def pickColor(colors: Colors.type): Color
```

Companion objects

It’s nice to know that we can get something resembling static methods and fields by putting them into a singleton object. However, what if we want to have both instance methods and static methods or fields for a given class? Since we cannot put static methods into our class, how can we achieve that?

The answer in Scala is called *companion objects*. If you have a class and a singleton object of the same name in the same source file, then the object is the companion object of the class, and the class is the *companion class* of the object.

One common use case for static methods is factory methods, which can provide a nicer API to create instances of your class. To see what this looks like in practice, let's create a companion object for the `Color` class. Here is a first version of our companion object, which you should add to the `Color.scala` source file:

```scala
object Color {
  def fromRGB(red: Int, green: Int, blue: Int): Color =
    new Color(red, green, blue)
}
```

When you start a new REPL session, you will be able to create a new `Color` instance like this:

```
scala> val gray = Color.fromRGB(50, 50, 50)
val gray: Color = Color@41269401
```

There is a special relationship between a class and its companion object: A class can access the private fields and methods of its companion object, and vice versa. We will see what that means when we discuss visibility modifiers in Section 3.3.

The apply method

When it comes to factory methods in companion objects, in cases where you only have one such method for a class and it's clear what it's doing, it's common practice to name that function `apply`. This allows you to create an instance of the respective class and make it look like applying the companion object as a function.

Why is that the case? In Scala, there is some syntactic sugar for calling a method if its name is `apply`, regardless of the number of parameters. For example, if there is a class `MyClass` that has a method `def apply(s: String): Int`, and a value `myInstance` of type `MyClass`, the method can be called not only as `myInstance.apply("hi")`, but also as `myInstance("hi")`. Essentially, you can leave out the `.apply` part. The idea is to make calling this method look more like function application. Idiomatic Scala code almost always uses the short form, leaving out the `.apply` part.

Let's rename the `fromRGB` method in our `Color` companion object:

```
object Color {
  def apply(red: Int, green: Int, blue: Int): Color =
    new Color(red, green, blue)
}
```

This is the most straightforward factory method you can imagine — all it does is delegate to the constructor of the companion class. What we gain from this is that we can create a new `Color` instance like this:

```
scala> val gray = Color(50, 50, 50)
val gray: Color = Color@3551e190
```

This will be expanded to writing `Color.apply(50, 50, 50)`.

One object to boot them all

We have been writing quite a bit of code now, experimenting with it in the Scala REPL and by writing tests. It's time to finally tell you how to create an entry point for an executable Scala application.

On the JVM, this entry point is always a static method called `main` that takes a `String` array, which will contain the arguments passed to the application. Since Scala runs on the JVM, this is exactly what we have to do as well. Let's create a file `Main.scala` next to the other source files in our `boardgame` sbt project, and create a singleton object `Main` inside like this:

```
object Main {
  def main(args: Array[String]): Unit = {
    println("Hello world!")
  }
}
```

Hooray, we are already in the third chapter and have finally seen a *Hello World* program. In order to execute it, type `run` in your sbt shell:

```
sbt:boardgame> run
[info] running Main
Hello world!
[success] Total time: 0 s, completed 4 Dec 2019, 09:23:43
```

Note that if your project contains more than one singleton object with a `main` method, sbt's `run` command will ask you to specify which one to launch.

Exercises

1. Move the color constants to the `Color` companion object and get rid of the `Colors` object.
2. Add a `Position` companion object with an `apply` factory method.
3. Add a `zero` value as a constant to the `Position` companion object.
4. Define a singleton object `MountDoom` that extends `Meeple`. This should be a meeple that has ID `1` and a constant position of `10, 5`, i.e. it cannot move.
5. Try to access the singleton object you defined from the REPL and assign it to a `val` of type `Meeple`.

3.3 Visibility modifiers

So far, you already know that fields of classes or objects are public by default and that class parameters are only visible within the body of the respective class. It's time to take a first look at Scala's visibility modifiers, as they are more powerful than what you may know from Java or Ruby, for example.

Please note that all the visibility modifiers you are about to see are applicable to classes, singleton objects, methods, fields, or constructors, even though you will not see all modifiers used in all of these contexts in this section.

Public by default

In Scala, everything is public by default: This is true for classes, objects, fields, methods as well as constructors. This also means that there is no such thing as a `public` modifier you may know from Java, for example.

What does it mean that something is public, though? For classes, this means, that anyone can reference instances of the respective class, as function parameters or fields, for example. For constructors, it means that anyone can create instances of the respective class. Public fields can be accessed from everywhere, and methods called.

While public by default has its advantages, there is also the danger of accidentally exposing implementation details to the outside and making them part of your API: You have to explicitly hide your implementation details. That's why it's important to always keep in mind that everything you write can used from everywhere else in your program, unless you take care of hiding it. In the following sections, you will see how to achieve that.

Designing for inheritance, revisited

Earlier in this chapter, when introducing the `final` modifier, we already touched the topic of designing for inheritance by making final all fields or methods in a class that are not supposed to be overridden by subclasses. Another aspect of designing for inheritance is to reduce the visibility of those fields and methods that subclasses are supposed to interact with, but that are not meant to be part of the public API.

Just like many other languages, Scala has the notion of `protected` fields and methods. Let's change our previous `Meeple` class by adding the `protected` modifier to those methods that are meant as hooks to be overridden by subclasses:

```scala
abstract class Meeple(id: Long, position: Position) {
  final def move(newPosition: Position): Meeple = {
    val requiredSteps = position.steps(newPosition)
    require(isValidStepCount(requiredSteps))
    withNewPosition(newPosition)
  }
  protected def isValidStepCount(steps: Int): Boolean
  protected def withNewPosition(pos: Position): Meeple
}
```

The effect of this is that the methods marked as `protected` can only be accessed by subclasses of `Meeple`. Please note that this gives you more protection than the `protected` modifier in Java, for example, where these methods can be accessed from other classes in the same package as well.

In our subclass, `RegularMeeple`, we should use the `protected` modifier as well when implementing these methods:

```
final class RegularMeeple(id: Long, position: Position)
    extends Meeple(id, position) {
  override protected def isValidStepCount(steps: Int): Boolean =
    steps > 0 && steps <= 5
  override protected def withNewPosition(pos: Position): Meeple =
    new RegularMeeple(id, pos)
}
```

The reason is that if we leave out the `protected` modifier when implementing or overriding a protected method, we are widening its visibility to `public`. Unless that is what you want, make sure to repeat the `protected` modifier here.

Saving Private This

Sometimes, you need to hide implementation details of a class or object. That is what the `private` modifier is for. Let's say we have a little helper function in our `Position` class to make the assertion of our invariants less repetitive. To prevent this implementation detail from becoming part of the API of that class, we'll mark it is private:

```
class Position(val x: Int, val y: Int) {
  assertInRange(x, "x")
  assertInRange(y, "y")

  def steps(other: Position): Int = {
    val xDiff = math.abs(other.x - x)
    val yDiff = math.abs(other.y - y)
    xDiff + yDiff
  }

  override def toString: String =
    "%s (x: %d, y: %d)".format(super.toString, x, y)

  private def assertInRange(value: Int, varName: String): Unit =
    require(
      value >= 0 && value < 20,
      "%s was %d, expected it to be >= 0 && < 20".format(varName, value)
    )
}
```

The `private` modifier only allows access to the respective method or field from instances of the class it is declared in. However, in Scala, we can go one step further and prevent access to fields or methods from anywhere but the instance to which the field or method belongs. This is done by means of the `private[this]` visibility modifier. Let's use this visibility modifier for the `assertInRange` method of our `Position` class:

```
private[this] def assertInRange(value: Int, varName: String): Unit =
  require(
    value >= 0 && value < 20,
    "%s was %d, expected it to be >= 0 && < 20".format(varName, value)
  )
```

Privacy and companion objects

As already hinted at when we introduced the concept of companion objects earlier in this chapter, there is a special relationship between a class and its companion object that allows both to access private fields or methods.

One example where you may want to access a private method defined in a singleton object from its companion class is to implement something similar to private static methods in Java. For instance, the `assertInRange` method we defined above as a private helper method is currently an instance method, as it is defined in the `Position` class, but it turns out that it is entirely stateless — there is no need for it to be an instance method. Instead, we could also define it in the `Position` companion object. Let's remove it from the `Position` class and put it into the companion object:

```
class Position(val x: Int, val y: Int) {
  Position.assertInRange(x, "x")
  Position.assertInRange(y, "y")

  def steps(other: Position): Int = {
    val xDiff = math.abs(other.x - x)
    val yDiff = math.abs(other.y - y)
    xDiff + yDiff
  }

  override def toString: String =
    "%s (x: %d, y: %d)".format(super.toString, x, y)
}

object Position {
  private def assertInRange(value: Int, varName: String): Unit =
    require(
      value >= 0 && value < 20,
      "%s was %d, expected it to be >= 0 && < 20".format(varName, value)
    )
}
```

Now, even though the `assertInRange` method is visible to the `Position` class, it is not automatically in scope, so we need to use the qualified identifier `Position.assertInRange`. Alternatively, we could import the method. While we have used the `import` keyword before, I haven't explained it yet, so we use the qualified identifier in this example.

3.4 Traits

No discussion of Scala's object-oriented side can be complete without introducing its concept of *traits*. In their simplest form, traits are purely abstract interfaces, and if a concrete class or object extends them, it needs to implement the fields or methods of that trait. In this book, we will restrict ourselves to using traits as interfaces. In the upcoming book, we'll also look at other ways of using traits and the pitfalls associated with them.

Traits as interfaces

Let's assume that we have the game board represented by a class `Board`. Please define the following class in a file `Board.scala` next to the other source files:

```
final class Board(
    val playerMeeples: Seq[Meeple],
    val opponentMeeples: Seq[Meeple]
) {
  require(
    playerMeeples.nonEmpty,
    "board must have at least 1 player meeple"
  )
  require(
    opponentMeeples.nonEmpty,
    "board must have at least 1 opponent meeple"
  )

  def firstPlayerMeeple: Meeple   = playerMeeples.head
  def firstOpponentMeeple: Meeple = opponentMeeples.head
}
```

A board contains two fields: one is a collection of all the player's meeples, the other a collection of the opponent's. A `Seq[A]` is a generic collection type from the standard library. It's similar to an array, but immutable[4]. Like `Array[A]`, it has a type parameter, but we are using a concrete type of sequence, a sequence of meeples, or `Seq[Meeple]`.

There are two invariants that we enforce using the `require` methods you have seen before: The two sequences of meeples must not be empty. This is because we define two methods that return the first player and opponent meeple, respectively. If `playerMeeples` was empty, `firstPlayerMeeple` would fail at runtime, crashing our program. The `head` method defined on `Seq` returns the first element of the sequence — or throws a `NoSuchElementException` if the sequence is empty. It's not safe to call this method without checking that the sequence is non-empty first. In Chapter 9 you will learn about a more elegant and safer way to access the head of a sequence.

We also want to represent a potential move of a meeple on the game board as data. Hence, we have a class `Move`. Please add a file `Move.scala` with the following content:

[4]There is also a mutable variant of `Seq`, but the `Seq` type that's automatically available is immutable. This was different in Scala versions before 2.13.

```scala
final class Move(val meepleId: Long, val newPosition: Position)
```

As mentioned, in its most simple form, traits are purely abstract interfaces. Let's say our game has a feature that allows players to ask for a recommended move when they are stuck. It's possible that we want to have different implementations of such a recommender engine. This is why we define an interface for it, a trait called `MoveRecommender`. Let's add a file `MoveRecommender.scala` that looks like this:

```scala
trait MoveRecommender {
  def recommendMove(board: Board): Move
}
```

We define a trait by prepending its name with the keyword `trait`. Like classes and objects, traits can have a body, in which you can define methods or fields. Just as with abstract classes, those methods or fields don't have to be implemented. A concrete class that extends this trait, however, needs to implement all fields or methods that are still abstract. Let's illustrate how this is done by implementing a `ReallySimpleMoveRecommender` in the same source file:

```scala
object ReallySimpleMoveRecommender extends MoveRecommender {
  override def recommendMove(board: Board): Move = new Move(
    meepleId = board.firstPlayerMeeple.id,
    newPosition = new Position(0, 0)
  )
}
```

This implementation of the `MoveRecommender` always suggests to move the first of the player's meeples to position `0, 0`.

The example shows that there is no difference between extending a class and extending a trait — we use the `extends` keyword in both cases.

Traits can be extended not only by concrete classes, but also by singleton objects, abstract classes or other traits. An abstract class or trait extending another trait doesn't have to implement any of the abstract methods of that trait, but it's free to do so.

Traits can also be used as *mix-ins* that interact with the class mixing in the trait. While that's a really powerful mechanism, it can also lead to a lot of trouble. Thus, for the time being, we'll stick to traits as interfaces.

Function values revisited

Let's go back to what we learned in the previous chapter about function values. Just to recap, we can define a function value like so:

```
scala> val isEven: Int => Boolean = x => x % 2 == 0
val isEven: Int => Boolean = $$Lambda$1023/582705581@569ed38f
```

What you may not know is that the type of this function literal, `Int => Boolean`, is just syntactic sugar for the type `Function1[Int, Boolean]`. You can actually try this in the Scala REPL:

```
scala> val isEven: Function1[Int, Boolean] = x => x % 2 == 0
val isEven: Int => Boolean = $$Lambda$1163/1398767440@3bcacbdf
```

Let's look at how this `Function1` thing is defined:

```
trait Function1[T1, R] {
  def apply(v1: T1): R
}
```

This is a somewhat simplified version of `Function1`[5]. Nevertheless, it still contains some new concepts — type parameters, which you have already seen briefly when we used arrays and sequences.

`Function1` is a generic trait with two type parameters (the things in square brackets), one for the input parameter and one for the return type. A concrete function type needs to fill in these type parameters. In our example, a function from `Int` to `Boolean` is of type `Function1[Int, Boolean]`. Type parameters will be explained in detail in Chapter 7.

The important thing to understand for now is that this is an ordinary trait, and `Int => Boolean` is just syntactic sugar for it that is baked into the language. This means that we can extend this trait to implement the functionality of `isEven`, like so:

[5]If you decide to look at the actual Scala source code of `Function1` you will notice a few differences. However, those are not important in the scope of this book.

```
object IsEven extends Function1[Int, Boolean] {
  override def apply(v1: Int) = v1 % 2 == 0
}
```

Let's put that into a source file `IsEven.scala` in the `repl-yell` sbt project and start a new REPL session, so that we can use it:

```
scala> val no = IsEven(3)
val no: Boolean = false

scala> val yes = IsEven(4)
val yes: Boolean = true
```

Here, you can also see the the apply method in action again (see Section 3.2). The reason we can apply function values by passing the arguments to it in parentheses is that each of the `Function` traits has an `apply` method. So we could just as well have written it like this:

```
scala> val no = IsEven.apply(3)
val no: Boolean = false
```

The same is true for the function literal we assigned to the `isEven` identifier in the `Predicates` singleton object. It's customary to leave out the `.apply`, but it's totally possible to write it explicitly:

```
scala> val no = Predicates.isEven(3)
val no: Boolean = false

scala> val yes = Predicates.isEven.apply(3)
val yes: Boolean = false
```

If you have worked in a more functional way in Java, the `Function1` trait may look familiar to you. Prior to Java 8, which introduced lambdas to the language, you used these interfaces by creating *anonymous classes*. Let's see what that would look like in Scala, by defining such a function value in the REPL:

```
scala> val isEven = new Function1[Int, Boolean] {
     | override def apply(x: Int) = x % 2 == 0
     | }
val isEven: Int => Boolean = <function1>
```

The REPL represents this function value as `<function1>`. That is the `toString` value of the `Function1` trait. Implementing functions like this leads to a different byte code representation than using the function literal syntax, `(x: Int) => x % 2 == 0`.

Usually, you would always use the functional literal syntax, but sometimes it can be easier or necessary to extend the `Function1` trait instead[6]. It's good to know that functions using literal syntax and functions implemented by extending one of the function traits are interchangeable, even though they have a completely different byte code representation — both approaches create functions that are values which can be passed around.

3.5 Everything is an object

You may know that in Java, there is a distinction between primitive types like `int`, `double`, or `boolean`, and objects that are instances of classes, like `String`, `HashMap` or any user-defined class. Other languages, for example Ruby, take their object-oriented nature more seriously and represent everything as objects, even those traditionally primitive types.

In this respect, Scala is more object-oriented than Java, because it, too, has the notion that everything is an object. Let's have a look at some mathematical calculations in the Scala REPL:

```
scala> val x = 5 + 3 * 2
val x: Int = 11
```

When we write `5 + 3`, we are actually calling a method called `+` defined in the `Int` class. Here is a tiny extract of the `Int` class:

[6]Or `Function2`, `Function3`, etc. Traits for functions with up to 22 parameters are available.

```
final abstract class Int private extends AnyVal {
  def +(x: Int): Int
}
```

There are two peculiarities here, which are not terribly important right now for the problem at hand, but still worth mentioning. First of all, `Int` is both abstract and final, so you cannot create instances of it using a constructor or subclass it. Secondly, this class extends a type called `AnyVal`. You will learn more about `AnyVal` in Chapter 6. For now, it suffices to know that this is the base type for all built-in value types, whereas normal classes, as you have seen, extend the base type `AnyRef`. Both `AnyRef` and `AnyVal` are subtypes of `Any`, which is at the top of the type hierarchy in Scala. In other words, it's the *top type* in Scala's type system.

Since `+` is just another method, we can re-write our previous piece of code like this:

```
scala> val x = 5.+(3.*(2))
val x: Int = 11
```

Of course, this is not how you should write it. Clearly, the previous notation is much easier to read. Why, however, is it possible to write `5 + 3 * 2` in the first place? The answer to that consists of two parts, infix operator notation and precedence rules.

Infix operator notation

First of all, there is some syntactic sugar for calling a method that takes a single parameter. For example, we can write this:

```
scala> val x = "My String" endsWith "b"
val x: Boolean = false
```

This is equivalent to the following:

```
scala> val x = "My String".endsWith("b")
val x: Boolean = false
```

This is also called *infix operator notation*, and, as already mentioned, you can use it for any method taking a single parameter. It is even available for methods taking multiple parameters, but then you will still have to encapsulate them in parentheses. For instance, here is the normal way of calling the `substring` method on a `String` value:

```
scala> val x = "My String".substring(3, 5)
val x: String = St
```

Using infix operator notation, it can be written like this:

```
scala> val x = "My String" substring (3, 5)
val x: String = St
```

However, most people agree that infix operator notation should be used sparingly for methods that don't act as operators. You usually wouldn't call the `endsWith` or `substring` methods like that, for example. Infix operator notation is also important as an enabler of internal DSLs[7].

Precedence rules

Since there are no operators as such in Scala, but only operator notation for what's actually a method call, precedence rules in Scala are just rules about the order in which methods in infix operator notation get called. That order, on the other hand, depends solely on the first character of the name of the method, according to the following figure.

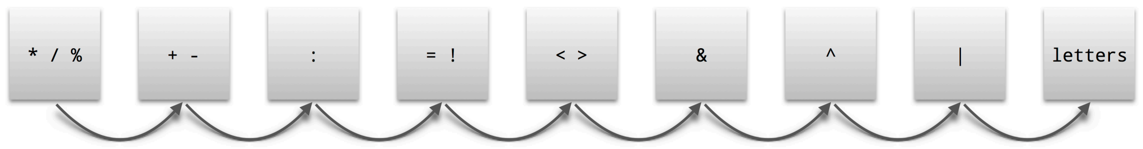


Figure 3.1: Precedence rules for infix operator notation by method name starting character

As Figure 3.1 shows, methods whose name starts with a `*`, `/`, or `%`, have higher precedence than methods whose name starts with a `+` or a `-`, and so on. Hence, by following these rules, the Scala compiler will be able to translate `5 + 3 * 2` to `5.+(3.*(2))`.

You can change precedence by inserting parentheses around an expression:

[7]Domain-specific languages. Internal DSLs are those that are expressed using the host language, in this case Scala, instead of having to be parsed and interpreted.

```
scala> val x = (5 + 3) * 2
val x: Int = 16
```

This is equivalent to the following:

```
scala> val x = 5.+(3).*(2)
val x: Int = 16
```

What's important to take away here is that there is nothing special about methods like `+` or `*`, as there are no built-in operators. Everyone can build APIs that benefit from the precedence rules, which are based on the starting character of the method name.

Unary prefix operators

You've just learned that Scala doesn't have any operators, and expressions `3 + 5` get translated to method calls. The same is true for unary prefix operators like the logical not. Consider the following expression:

```
scala> val x = !true
val x: Boolean = false
```

Even here, `!` is not actually an operator. The compiler has some knowledge baked in about specific unary operators and how to translate them to method calls. If it finds the operator symbol `!` in front of a value, it tries to translate that into a call to the `unary_!` method, which happens to be defined on the `Boolean` type:

```
scala> val x = true.unary_!
val x: Boolean = false
```

The same happens when you negate a number, for example. Table 3.1 summarises all unary prefix operators.

Table 3.1: Unary operators

Types	operator	method	description
`Boolean`	`!`	`unary_!`	logical *NOT*
all numeric types	`-`	`unary_-`	negation
all numeric types	`+`	`unary_+`	no effect
all numeric types	`~`	`unary_~`	bitwise negation

Basic types at runtime

Integers represented as objects? Isn't that expensive? Luckily, that is not the case. When compiled to Java byte code, types like `Int`, `Double`, `Boolean`, and so on are represented the same way as the corresponding Java primitive types, so there is nothing to be concerned about.

3.6 Am I equal? Yes I am!

Having talked about how from the Scala compiler's perspective everything is an object, it's time to take a closer look at what it means for two values to be equal in Scala.

Value equality versus reference equality

In most languages, there are two different ways of determining equality. On the one hand, there is *value equality* or *structural equality*, on the other, there is *reference equality*.

Value equality or structural equality refers to a strategy where values are considered equal if they are semantically equivalent. For example, two 10 EUR bills are considered equal when using this strategy, because both bills have the same value. It doesn't matter at all whether these are two physically different bills, or, in other words, whether these are the *same* bill.

In contrast, reference equality refers to a strategy where two values are only considered equal if they are actually the same object. Staying with the example above, if I have two 10 EUR bills in my purse, they are not considered to be equal using reference equality. In the world of programming, two value or variable identifiers are only considered to be equal if they point to the same object, i.e. to the same memory address.

It's easier to understand the difference between these two concepts of equality by looking at concrete code examples. Let's create a new sbt project `am-i-equal`, using sbt's `new` command:

```
$ sbt new dwestheide/minimal-scala-project.g8
[info] Loading global plugins from /home/daniel/.sbt/1.0/plugins
[info] Set current project to private (in build file:/home/daniel/)
[info] Set current project to private (in build file:/home/daniel/)
name [my-awesome-scala-project]: am-i-equal

Template applied in /home/daniel/./am-i-equal

$ cd am-i-equal
```

Now, let's start with an example that demonstrates value equality. In the project's root directory we're going to start the interactive sbt shell and, from there, a new REPL session using the `console` command. In the Scala REPL, let's use the `==` operator to check if two `Int` values `a` and `b` are equal:

```
scala> val a = 10 + 5
val a: Int = 15

scala> val b = 15
val b: Int = 15

scala> val amIEqual = a == b
amIEqual: Boolean = true
```

As expected, `a` is considered to be equal to `b`, because `10 + 5` is equal to `15`. `a` and `b` have the same value.

Now, let's add a file called `Money.scala` in the `src/main/scala` directory of our sbt project. In this file we're going to define a class `Money` that looks like this:

```
final class Money(val value: Int, val currency: String) {
  override def toString: String = "%s %d".format(currency, value)
}
```

Now, if we start a new REPL session, we can check if two 10 EUR bills are considered to be equal:

```
scala> val myBill = new Money(10, "EUR")
val myBill: Money = EUR 10

scala> val yourBill = new Money(10, "EUR")
val yourBill: Money = EUR 10

scala> val amIEqual = myBill == yourBill
val amIEqual: Boolean = false
```

Again, we are using the `==` operator. Apparently, `yourBill` and `myBill` are not considered to be equal. What we have here is a case of reference equality, just as described in the example earlier.

Why is it that `Int` values use value equality, while `Money` values use reference equality?

The == method

When we use the `==` operator in Scala, what we actually do is call the `==` method using infix notation. If we wanted to, we could also use the normal method invocation syntax, writing `x.==(y)` instead of `x == y`.

The behaviour of the `==` method depends on the type of the value it is called on. If it's a basic type (`Int`, `Long`, `Float`, `Double`, `Char`, `Boolean`, `Byte`), it will perform a value equality check. Remember that while values of these types are objects in Scala, they get compiled to primitive Java types, like `int`, `long`, `float`, and so on. So, a call to the `==` method on values of these types gets compiled to Java's `==` operator. For primitive types that operator performs a value equality check. It helps to remember that the parent class of all these built-in types is `AnyVal`.

If the type of the value is not a primitive type, things are a bit more complicated. In this case, the `==` method will delegate to a method called `equals`, which has the following signature:

```scala
def equals(other: Any): Boolean
```

The `equals` method is defined on `Any`, the parent type of all other types in Scala. It helps to remember that for any type extending from `AnyRef`, the parent type of all classes and singleton objects, the `==` method will perform a reference equality check — unless you decide to override the `equals` method.

Overriding `equals`

In most software, our model of money is such that we don't care at all about physical representations of bills or coins. When we check if two monetary amounts are equal, we want to check for structural equality instead of reference equality. Two monetary amounts are equal if their value and currency are equal.

To achieve that, we need to override the default behaviour of the `equals` method. The `equals` method on `AnyRef` is really the `equals` method from `java.lang.Object`. Usually, when you override it to implement structural equality, you check if the two values have the same type, and if they do, you check for equality of each of the relevant fields. For our `Money` class, it could look like this:

```
override def equals(other: Any): Boolean =
  if (other.isInstanceOf[Money]) {
    val that = other.asInstanceOf[Money]
    value == that.value && currency == that.currency
  } else false
```

The other value is equal to ours if it is also of type `Money` and if both its `value` and `currency` field are equal to ours. Here we rely on the natural value equality of `Int` (the type of the `value` field) and on the fact that `String` (the type of the `currency` field) overrides `equals` as well to provide for value equality.

Please note that implementing `equals` correctly on the JVM can be quite difficult. Our approach works fine because the `Money` class is final, which means there can't be any subclasses. If you allow subclassing, implementing `equals` becomes more difficult — one more reason to avoid it if possible.

If you override `equals`, it needs to have the following properties[8]:

- **Reflexivity**: A value must be equal to itself.
- **Symmetry**: If `a == b` returns `true` then`b == a` must also return `true`.
- **Transitivity**: If `a == b` returns true *and* `b == c` returns `true` then `a == c` must also return `true`.
- **Consistency**: Evaluating `a == b` repeatedly must return the same result, unless one of the values has changed its internal state. This shouldn't happen if you only use immutable data structures.

Now you can check for value equality of two instances of the `Money` class. Let's do that in a new REPL session:

[8]Bloch, J. (2017): Effective Java. Addison-Wesley Professional

```
scala> val tenEuros = new Money(10, "EUR")
val tenEuros: Money = EUR 10

scala> val alsoTenEuros = new Money(10, "EUR")
val alsoTenEuros: Money = EUR 10

scala> val amIEqual = tenEuros == alsoTenEuros
val amIEqual: Boolean = true
```

Now, two different instance of `Money` are equal if they have the same value and currency.

What if you really need to check for reference equality of two instances of a class, though? In functional programming, where values are immutable, this should rarely be necessary. In almost all situations value equality or structural equality is what matters. Nevertheless, if you happen to run into a situation where it does matter, you can always use a method called `eq` that's defined on `AnyRef`. Here's an example:

```
scala> val tenEuros = new Money(10, "EUR")
val tenEuros: Money = EUR 10

scala> val alsoTenEuros = new Money(10, "EUR")
val alsoTenEuros: Money = EUR 10

scala> val notTheSame = tenEuros.eq(alsoTenEuros)
val notTheSame: Boolean = false

scala> val anotherTenEuros = tenEuros
val anotherTenEuros: Money = EUR 10

scala> val same = tenEuros.eq(anotherTenEuros)
val same: Boolean = true
```

By the way, while structural equality is almost always a good choice for data classes like `Money`, there is no point in aiming for structural equality checks for classes or traits that act as modules. For example, you'll never want to check whether two instances of the `MoveRecommender` type are structurally equal. Don't override `equals` for these kinds of classes.

Overriding hashCode

The contract of the `equals` method in `java.lang.Object` is such that you also need to override the `hashCode` method in your class in a way that is consistent with the `equals` method. In short, this means that if two objects are equal, they must also have the same hash code. If they are not equal, their hash codes don't *have* to be different, although that's strongly recommended for performance reasons. The `hashCode` method exists to support data structures like hash maps and hash sets. They are more efficient if unequal objects have different hash codes.

Luckily, overriding `hashCode` involves a lot less code than overriding `equals`, because we can delegate most of the work to a helper method called `hash`. This method is defined in a Java class called `java.util.Objects`. Here is our `hashCode` implementation for the `Money` class:

```
override def hashCode(): Int = java.util.Objects.hash(value, currency)
```

While there is a lot more that can be said about `equals` and `hashCode`, this is more than enough. In fact, in the next chapter, you will learn about a feature that eliminates the need for you to implement these two methods completely.

3.7 Summary

In this chapter, you got a thorough overview of Scala's object-oriented features, from classes to singleton objects and traits as mechanisms for modularising your functions. We also discussed how to design for inheritance and how to hide the internals of your modules from prying eyes.

The next chapter will be pretty packed. You will learn about some cool language features that are crucial in making Scala a very expressive language, allowing you to write code that is more concerned with *what* instead of the *how*.

4. Scala, the expressive language

In the last two chapters, you got an overview of what you need to know about both Scala's functional and object-oriented side in order to be prepared for what's to come. Now, we're going to look at a few other features, of both the language and its standard library, that make Scala a very expressive language: All of the features you learn about in this chapter play a part in empowering you to write code that clearly conveys its intent to the reader, instead of hiding it between layers of boilerplate code.

4.1 Named and default arguments

Most modern programming languages have a notion of named arguments as well as default arguments, and Scala is no exception. Using named arguments, the intent of your code is easier to grasp, and you may avoid bugs introduced by passing arguments of the same type in the wrong order[1].

Named arguments

Remember our `Color` class from the previous chapter? Its companion object has an `apply` factory method for creating new `Color` instances:

```
object Color {
  def apply(red: Int, green: Int, blue: Int): Color =
    new Color(red, green, blue)
}
```

We can create a new `Color` instance like this:

[1]Avoiding multiple arguments of the same type is an even safer way of avoiding these kinds of bugs, as the problem will already be caught by the compiler. You'll learn how to do that in Section 6.1.

```
scala> val color = Color(200, 40, 30)
val color: Color = Color@32fc68db
```

However, it's easy to get the order of the arguments you pass to a method, function, or constructor wrong. Also, a reader of your code might not even know that the `apply` method expects RGB values. Using named arguments can help to prevent errors resulting from wrong argument order. Moreover, it helps readers of your code to understand what the `apply` method expects without looking up its documentation. Here is what using named arguments looks like in Scala:

```
scala> val color = Color(red = 200, green = 40, blue = 30)
val color: Color = Color@2f6c4faa
```

We specify arguments by providing the name of each argument, followed by the `=` symbol and the argument value.

Using named arguments also allows you to deviate from the order you would have to pass in the arguments. For example, it's perfectly possible to create a new `Color` instance like this:

```
scala> val color = Color(green = 40, red = 200, blue = 30)
val color: Color = Color@7d30872a
```

There is usually no good reason to change the order of the arguments, but it's good to be aware that it's possible.

You may wonder whether it's possible to mix positional and named arguments. The answer is yes, as long as you don't put any positional arguments after named arguments that don't follow the order of the parameters. For example, this is allowed:

```
scala> val color = Color(200, blue = 30, green = 40)
val color: Color = Color@43796b48
```

This is also allowed:

```
scala> val color = Color(200, green = 40, 30)
val color: Color = Color@2e968c53
```

Even though you name the `green` argument, you haven't changed any positions, so that the `30` can be unambiguously assigned to the `blue` parameter.

As soon as you start actually changing the order in which you pass in the arguments, though, the compiler will not let you place a positional, unnamed argument after a named one.

Unfortunately, named arguments are only available when calling Scala methods or constructors. When you call methods or constructors from Java libraries, you cannot make use of this language feature.

Default arguments

Scala also allows you to specify default values for your parameters. Let's stick to our `Color` class and the factory method defined in its companion object, and assume we want to provide zeros as default values for every parameter. Here is what our `apply` method will have to look like:

```
object Color {
  def apply(red: Int = 0, green: Int = 0, blue: Int = 0): Color =
    new Color(red, green, blue)
}
```

Apparently, default values are specified by adding the `=` symbol and the default value after the name and type of the respective parameter. You can't leave out the type annotation here and let the Scala compiler infer the parameter type from the type of the default value.

Once default values are specified, we can choose to leave one, some, or all arguments unspecified, so that the respective default value will be used. Here is how we create a new `Color` instance using only default values that happens to represent the color black:

```
scala> val black = Color()
val black: Color = Color@5d275b33
```

Using positional arguments, you can only leave out arguments from right to left. If you want to leave out the second argument, you cannot provide the third one. For example, you can provide the first argument and leave out the rest:

```
scala> val red = Color(255)
val red: Color = Color@50612529
```

If you want to specify the third argument but leave out the second one, you have to use named arguments:

```
scala> val color = Color(50, blue = 100)
val color: Color = Color@1f9e3c54
```

Here, the first argument is positional, the second is unspecified, and the third one, `blue`, has to be specified as a named argument. Instead of mixing positional and named arguments, though, it may be more readable to use named arguments consistently.

Please note that default arguments are not without their own problems. They can be convenient for you as a developer, especially when creating instances of some kind of configuration class. However, they can also lead to subtle bugs — it's easy to forget to pass an argument to a method that has a default, and often, default values do not make a lot of sense when the method is called in production code. Moreover, methods with default arguments don't play very well with some of Scala's functional features, which we're going to cover in the second book. This is why some people prefer to use separate methods for each parameter list you want to support, with distinct names for each. Hence, make sure to use default arguments sparingly and with caution.

4.2 A first glance at pattern matching

Like many functional languages, Scala has a feature called *pattern matching*. It's one of the crucial pieces, because it makes it easy to separate data from behaviour, which is an important aspect of functional programming. Since it's such a convenient feature, and because some of the other features we introduce in this chapter play so well with it, we're going to cover the essentials of pattern matching in Scala in this chapter.

So what is pattern matching? Some people like to think of it as a switch statement on steroids, and in certain ways, you could describe it like that. Then again, this is already wrong because like almost everything in Scala, pattern matching is not implemented as a statement, but as an expression.

Pattern matching provides a declarative way of defining conditional execution, where the condition is not a simple boolean expression, but whether the input data matches a specific pattern. The action you want to execute in case a particular pattern matches can access values that the matching pattern has extracted from the input data. You'll see an example of that shortly.

To get a better idea of what pattern matching looks like in Scala, we create a new sbt project called `greeting`, like so:

```
$ sbt new dwestheide/minimal-scala-project.g8
[info] Loading global plugins from /home/daniel/.sbt/1.0/plugins
[info] Set current project to private (in build file:/home/daniel/)
[info] Set current project to private (in build file:/home/daniel/)
name [my-awesome-scala-project]: greeting

Template applied in /home/daniel/./greeting

$ cd greeting
```

As before, you will be asked to enter the name of the project, `greeting` in this case. Let's create a new file called `Greeting.scala` in the `src/main/scala` directory of the new project containing a `Greeting` singleton object:

```scala
object Greeting {
  // here we are going to put all our examples
}
```

Finally, here is a function using pattern matching, which we'll define in the `Greeting` object:

```scala
def greeting(name: String): String = name match {
  case "Daniel" => "Guten Morgen!"
  case "Sóley"  => "Góðan dag!"
}
```

A *pattern matching expression* consists of an input expression (in this case, `name`), followed by the `match` keyword and one or more cases, surrounded by curly braces. A case, in turn, consists of the `case` keyword followed by a pattern on the left side, the `=>` arrow symbol familiar from function literals, plus a block on the right side, to be executed if this pattern matches.

The block on the right side can be in a new line or in the same line as the pattern. Unlike the code blocks in all other places you have seen so far, it can consist of multiple lines without requiring enclosing curly braces. For example, we could modify the previous example like this:

```scala
def greeting(name: String): String = name match {
  case "Daniel" =>
    println("Determining how to greet Daniel")
    "Guten Morgen!"
  case "Sóley" =>
    println("Determining how to greet Sóley")
    "Góðan dag!"
}
```

Kinds of patterns

There are multiple kinds of patterns, so let's walk through some of the most important ones. A few others will pop up later in this chapter, once we introduce the necessary prerequisites.

Literal patterns

This is what we have been using in our first example pattern matching expression above, repeated here for convenience:

```scala
def greeting(name: String): String = name match {
  case "Daniel" => "Guten Morgen!"
  case "Sóley"  => "Góðan dag!"
}
```

Both of the two cases use a literal pattern, the first one matching against the `String` value `"Daniel"`, the second one against `"Sóley"`.

Let's use a unit test to see how the `greeting` function behaves. In the `src/test/scala` directory, add a file `GreetingTest.scala` that looks like this:

```
import minitest._

object GreetingTest extends SimpleTestSuite {

  test("Daniel is greeted in German") {
    assertEquals(Greeting.greeting("Daniel"), "Guten Morgen!")
  }
}
```

In the interactive sbt shell of our `greeting` project we can run the `test` command to execute this test, and everything should be fine:

```
sbt:greeting> test
```

The wildcard pattern

There's a problem in the pattern matching expression that we used for our `greeting` function. What happens if we apply the `greeting` function to a value for which there is no case in our pattern matching expression? Let's find out in the REPL:

```
scala> Greeting.greeting("Anna")
scala.MatchError: Anna (of class java.lang.String)
  at Greeting$.greeting(Greeting.scala:4)
  ... 39 elided
```

Ouch! We got a `MatchError`, which is a nasty runtime error that happens if a pattern matching expression does not have any matching case for the given input expression. To avoid it it's important to make sure that you always have a matching case. One common way of doing that is by means of a wildcard pattern, which uses the underscore symbol _, like so:

```
def greeting(name: String): String = name match {
  case "Daniel" => "Guten Morgen!"
  case "Sóley"  => "Góðan dag!"
  case _        => "Good morning!"
}
```

Let's add the following test to the `GreetingTest` object to make sure our function behaves as expected:

```
test("Unknown people get greeted in English") {
  assertEquals(Greeting.greeting("Anna"), "Good morning!")
}
```

Run the `test` command again, and you should see that this test succeeds.

Pattern alternatives

Wouldn't it be nice if we could combine multiple patterns with identical right hand sides, so that we don't have to write a separate case for each of them? In fact, this is possible with pattern alternatives. You can provide two or more patterns, separated by the `|` symbol, and the resulting composite pattern will match if at least one of the combined patterns matches.

For example, we can return an Icelandic greeting not only for Sóley, but also for Snorri:

```
def greeting(name: String): String = name match {
  case "Daniel"            => "Guten Morgen!"
  case "Sóley" | "Snorri" => "Góðan dag!"
  case _                   => "Good morning!"
}
```

There are some limitations to what patterns you can combine this way. Variable patterns and pattern binders, which we will cover below, cannot be used as pattern alternatives.

Variable patterns and typed patterns

In a statically typed functional programming language, we try to avoid method signatures that expect parameters of type `Any`. If a value is of type `Any`, we don't know anything about it at compile time. We also try to use methods with a return type of `Unit` sparingly, only in those parts where we need side effects.

Nevertheless, Akka, a popular toolkit implementing the actor model, had been based on the notion of a function `Any => Unit`[2] for a long time. To demonstrate

[2]This has been the topic of a few cruel jokes, given how much Scala emphasizes the advantages of static typing and a powerful type system. To be fair, there are good reasons for why Akka's actors had been built around the untyped `Any => Unit` function for such a long time. Typed actors are an incredibly difficult field of research. There have been multiple attempts to come up with a sound solution before the creation of *Akka Typed*, which was only declared stable in Akka 2.6.0, released in November 2019.

some of the remaining kinds of patterns, let's implement such an untyped function as well in our `Greeting` object.

Please note that is this solely for the purpose of demonstrating as many types of patterns as possible. Generally, it's bad style to write functions that expect or return parameters of type `Any`. Since we have a statically typed language, and a powerful type system, we should make use of type information at compile time as much as possible.

With that being said, here is our `handleMessage` function:

```scala
def handleMessage(message: Any): Unit = message match {
  case _: Int               => println("Got a number!")
  case "Daniel"             => println("Guten Morgen!")
  case "Sóley" | "Snorri" => println("Góðan dag!")
  case name: String         => println("Good morning, %s!".format(name))
}
```

Here, we introduce typed patterns and variable patterns. The first typed pattern provides a type to a wildcard pattern — it will match if `message` is of type `Int`, but since we use a wildcard pattern, we cannot use that `Int` value on the right side. The second usage of a typed pattern is in combination with a variable pattern. This pattern matches if `message` is of type `String`, and it binds that `String` value to the `name` identifier. We can use the value `name` which is known at compile time to be of type `String`, on the right side.

While this is not evident in the example, it's also possible to use variable patterns without making them typed.

Stable identifier patterns

Sometimes, you want to match against an existing value already assigned to some identifier. Consider the following change to the `handleMessage` method:

```scala
val aMessage = "Message"
def handleMessage(message: Any): Unit = message match {
  case aMessage     => println("We got a message!")
  case name: String => println("Good morning, %s!".format(name))
}
```

In the first case, we want to match against the value assigned to the `aMessage` identifier. But that's not what happens. Indeed, `aMessage` is a variable pattern, as described above. Since it is not a typed pattern, it will match anything. The compiler is helpful enough to warn you about this, but let's try it out in the REPL anyway to see for ourselves:

```
scala> Greeting.handleMessage("Daniel")
We got a message!
```

This is certainly not what we intended. To match against the value of an existing identifier, we need to place backticks around that identifier in the pattern:

```
val aMessage = "Message"
def handleMessage(message: Any): Unit = message match {
  case `aMessage`   => println("We got a message!")
  case name: String => println("Good morning, %s!".format(name))
}
```

Now the first case uses a stable identifier pattern. It will only match if we pass it a value that is equal to the `aMessage` value:

```
scala> Greeting.handleMessage("Daniel")
Good morning, Daniel!

scala> Greeting.handleMessage("Message")
We got a message!
```

Pattern binders

Sometimes, you cannot use variable patterns, but still want to bind the value matched by a pattern to an identifier, honouring the statically known type of that pattern. For example, we may want to greet our Icelandic friends by name. To achieve that, we are going to use a pattern binder, which is an identifier, followed by the `@` symbol and a pattern. Let's add these two methods to the `Greeting` object:

```
def greetByName(id: Int): String = nameForId(id) match {
  case "Daniel"                    => "Guten Morgen, Daniel!"
  case name @ ("Sóley" | "Snorri") => "Góðan dag, %s!".format(name)
  case _                           => "Good morning!"
}

def nameForId(id: Int): String = id match {
  case 1 => "Daniel"
  case 2 => "Sóley"
  case 3 => "Snorri"
  case _ => "Someone"
}
```

Here, we bind the value matched by the pattern alternatives to the identifier `name`. We have to put parentheses around the pattern alternatives so that the compiler understands we want to bind the whole composite pattern ("Sóley" or "Snorri"), not just the first pattern alternative (which would be illegal, as mentioned above when describing pattern alternatives).

As a result, we can greet not only me, but also Sóley and Snorri by name. We should add the following tests to the `GreetingTest` object:

```
test("Daniel gets greeted by name") {
  assertEquals(Greeting.greetByName(1), "Guten Morgen, Daniel!")
}

test("Icelandic people get greeted by name") {
  assertEquals(Greeting.greetByName(2), "Góðan dag, Sóley!")
  assertEquals(Greeting.greetByName(3), "Góðan dag, Snorri!")
}
```

Guard clauses

When I wrote that the left side of a case consists of the `case` keyword followed by a pattern, I didn't tell you the whole truth. The pattern can be followed by a *guard clause*. A guard clause is an `if` keyword followed by a boolean expression that has access to everything bound in the pattern. The case will only match if the pattern matches and the guard clause returns `true`.

To illustrate this, let's add a function called `englishGreeting` to the `Greeting` object. Unlike the previous `greeting` function, this is not a polyglot greeting function. Instead, if the given name is not too long, we will return a named greeting, otherwise the person has to live with an anonymous greeting.

```
def englishGreeting(name: String): String = name match {
  case shortName if shortName.length < 6 =>
    "Good morning, " + shortName + "!"
  case _ =>
    "Good morning, my friend!"
}
```

Let's see if it works by adding some tests to the `GreetingTest` object:

```
test("English greeting for short names uses the name") {
  assertEquals(Greeting.englishGreeting("Anna"), "Good morning, Anna!")
}
test("English greeting for long names is anonymous") {
  assertEquals(Greeting.englishGreeting("Daniel"), "Good morning, my friend!")
}
```

Poor me doesn't get a personalised greeting, but Anna does, because her parents knew that a short name would prove beneficial one day.

Exercises

1. In the `greeting` function, try changing the order of the cases. What happens if a wildcard pattern is used in the first case of a pattern matching expression?
2. In languages like Java, the `switch` statement falls through each case, unless you break out of it with the `break` keyword. Why is such a falling-through approach not possible for pattern matching cases in Scala?

4.3 Case classes

Let's get back to our `boardgame` sbt project and the `Color` class we defined in there. If you come from Java, you may already have noticed a few problems with it:

- no `equals` method: Since we haven't overridden the default implementation of the `equals` method from `AnyRef`, `==` is really checking reference equality right now, not value equality (see Section 3.6)
- no `hashCode` method: on the JVM, whenever you override `equals`, you also want to override the `hashCode` method, so that instances of your class can be used as elements of a hash set or keys of a hash map.

Since `Color` is supposed to be a pure data class, we definitely want to have a proper `equals` and `hashCode` implementation. These things are notoriously easy to get wrong, and are also quite a bit of boilerplate to write.

Luckily, Scala doesn't want you to suffer through having to override these methods for every single data class you write. To avoid that, it gives you *case classes*. Case classes are special boilerplate-free classes. If you define a case class, the Scala compiler generates a lot of the things a well-designed immutable data class is supposed to have for you — `equals` and `hashCode` implementations are only two of them.

Let's make our `Color` class a case class and then walk through the benefits we get for free. In the `Color.scala`, change the `Color` class so that it looks like this:

```scala
final case class Color(red: Int, green: Int, blue: Int) {
  require(red >= 0 && red < 256)
  require(green >= 0 && green < 256)
  require(blue >= 0 && blue < 256)
}
```

We define a case class by prefixing the `class` keyword with the `case` keyword. It's highly recommended to make case classes final as well, because they are not actually designed for inheritance. This is why we prefix our case class definition with the `final` keyword as well.

Generated `toString`

Let's see what we gain from case classes in a new REPL session in our `boardgame` sbt project:

```
scala> val red = Color(red = 255)
val red: Color = Color(255,0,0)
```

Here, we have created a new `Color` instance and assigned it to the `red` identifier. As always, the REPL prints out the result of the expression, calling `toString` on it. Even though we didn't override `toString`, we got back an informative `String` representation. That's because the Scala compiler generates a `toString` method for each case class. This is mainly useful for log outputs, and it's certainly more helpful than the default from `AnyRef`, which just returns the object id. And, although you get it generated for free, nothing prevents you from providing your own implementation of `toString` for your case class.

Generated factory method

Let's remove the companion object of our `Color` class, and add the default arguments to the case class parameter list:

```scala
final case class Color(red: Int = 0, green: Int = 0, blue: Int = 0) {
  require(red >= 0 && red < 256)
  require(green >= 0 && green < 256)
  require(blue >= 0 && blue < 256)
}
```

Since we have removed the companion object with its `apply` method, can we still create new instances of `Color` as we did before, or do now we have to call the constructor again using the `new` keyword?

If you try it out in a new REPL session, you'll see, that the following works:

```
scala> val red = Color(red = 255)
val red: Color = Color(255,0,0)
```

Why? The Scala compiler generates a companion object for your case class, if it doesn't already exist. It will automatically define an `apply` method in the companion object, based on your class parameter list. It's common practice in Scala to use those generated factory methods instead of the constructor, when creating new instances of case classes.

Please note that it is still possible to define the companion object for your case class yourself. In this case, the compiler will add the generated factory method to your hand-written companion object. If there is already an `apply` method in the companion object, the compiler won't generate one for you.

Generated public fields

In the previous form of the `Color` class, we had to define values for each class parameter, using the `val` keyword, in order to make them available as public fields — otherwise, class parameters are not visible outside of the class body. See how our case class version of `Color` doesn't have any of that? Instead, we merely provide a list of class parameters, and all of them are automatically turned into public, immutable fields, so that we can access them from the outside:

```
scala> val white = Color(255, 255, 255)
val white: Color = Color(255,255,255)

scala> val r = white.red
val r: Int = 255
```

It's still possible to make fields non-public for case classes, by providing the appropriate visibility modifiers in the class parameter list.

Generated `equals` and `hashCode`

Nobody likes implementing `equals` and `hashCode`, whether they are working with Scala, Java, or any other JVM language. With case classes, you don't have to worry about that at all, because those methods get generated for you as well. This means that we can check for value equality of two `Color` instances:

```
scala> val color1 = Color(red = 233, green  = 13, blue = 55)
val color1: Color = Color(233,13,55)

scala> val color2 = Color(red = 233, green  = 13, blue = 55)
val color2: Color = Color(233,13,55)

scala> val yes = color1 == color2
val yes: Boolean = true
```

We can also verify that there is a generated `hashCode` method by creating two equal instances of `Color` and comparing their respective hash codes, which should be the same:

```
scala> val blue = Color(blue = 255)
val blue: Color = Color(0,0,255)

scala> val anotherBlue = Color(blue = 255)
val anotherBlue: Color = Color(0,0,255)

scala> val hash = blue.hashCode
val hash: Int = 817177104

scala> val otherHash = anotherBlue.hashCode
val otherHash: Int = 817177104
```

Generated `copy` method

Since all data is immutable in functional programming, the way that we represent a change is by creating a new immutable value. This works fine for values of simple types like `Int`, `String`, or `Boolean`. For more complex classes, consisting of multiple fields, we would like to have an easy way of creating new instances of these classes, based on existing instances, replacing the values of one or more fields.

Since this is such a ubiquitous need, the Scala compiler generates a `copy` method for all case classes. This method uses named and default arguments, allowing you to only provide the arguments for the fields whose values you want to replace. Here is an example:

```
scala> val black = Color()
val black: Color = Color(0,0,0)

scala> val red = black.copy(red = 255)
val red: Color = Color(255,0,0)
```

Here, we have created a copy of the `Color` assigned to the `black` identifier. That copy has the same values for the `green` and `blue` fields as its source, but a different value for the `red` field.

Participation in pattern matching

All the generated code you have seen so far is really nice, as it allows you to define well-behaved data classes with minimal boilerplate. However, why are they actually called case classes? The reason is that the compiler generates some additional code that makes them work so nicely with pattern matching.

A *constructor pattern* lets you match an expression against a case class constructor, effectively destructuring it. Let's try this out in our `greeting` sbt project again. In the `Greeting.scala` file, we will add a `Person` case class that looks like this:

```
case class Person(name: String, preferredLanguage: String)
```

Now, we can add a method to the `Greeting` object that takes into account a person's language preference when greeting them:

```
def preferenceAwareGreeting(person: Person): String = person match {
  case Person(name, "de") => "Guten Morgen, %s!".format(name)
  case Person(name, "is") => "Góðan dag, %s!".format(name)
  case Person(name, _)    => "Good morning, %s!".format(name)
}
```

Pattern matching is nested — for each of the fields of `Person` you specify another pattern. In this case, `name` is a variable pattern. For the second field, we use literal patterns to match against all languages we support, and a wildcard pattern for the case where the preferred language is not supported. In the latter case we fall back to English.

Variable patterns nested in constructor patterns are useful for binding relevant values of a data structure to identifiers.

We can also use other patterns nested in constructor patterns, for example pattern alternatives.

4.4 Tuples

Most of the time, case classes, which can have a meaningful name from the language of your application's domain, are the way to go. However, there are situations where you don't want to come up with a new named class — you just want to temporarily group two or more values together. Some languages have no special support for this, so you end up implementing your own generic `Pair` class, or use one from some library you pull in. Other languages, like Python, have built-in support for this use case, in the form of tuples, and Scala is following suit.

A tuple is a data structure consisting of a fixed number of elements, where each element can have their own type. For example, here is how we create a tuple of two elements, a `String` and an `Int`, in the REPL:

```
scala> val nameAndAge = ("daniel", 37)
val nameAndAge: (String, Int) = (daniel,37)
```

We create a tuple by surrounding a comma-separated list of expressions with parentheses. It looks like instantiating a case class and leaving out its name in front of the argument list. The type of this tuple is `(String, Int)`.

We can create a tuple of three elements like this:

```
scala> val person = ("Daniel", "Westheide", 37)
val person: (String, String, Int) = (Daniel,Westheide,37)
```

This tuple has the type `(String, String, Int)`.

Beyond the sugar

The way we created our tuples, and the way the resulting tuple type is represented, for instance as `(String, Int)`, is actually just syntactic sugar. Behind the scenes, they are ordinary case classes with generic type parameters. For example, here is a simplified definition of `Tuple2` and `Tuple3`[3]:

```
final case class Tuple2[T1, T2](_1: T1, _2: T2)
final case class Tuple3[T1, T2, T3](_1: T1, _2: T2, _3: T3)
```

You have seen these type parameters in square brackets before in the previous chapters. You will learn more about them in Chapter 7. For now, remember that they need to be filled with concrete types, and in that, you can think of them as Scala's equivalent to Java's generics. So, here is how the tuple types from our examples get desugared:

- `(String, Int)` is equivalent to `Tuple2[String, Int]`
- `(String, String, Int)` is equivalent to `Tuple3[String, String, Int]`

[3]Just as for `Seq` and the function traits `Function1` to `Function22`, the actual definition in the Scala source code looks a bit different. The differences make use of advances aspects of the Scala type system that we can ignore for now.

There is one case class for each number of elements, from `Tuple1` to `Tuple22`.

Each element of a tuple is a field of the respective case class, so we can access the first element of a tuple by accessing the `_1` field, the second one by accessing the `_2`, and so on:

```
scala> val name = nameAndAge._1
val name: String = daniel
```

Tuples in pattern matching

Sometimes, accessing tuple elements using the case class fields, like `_1`, `_2`, makes our code a bit ugly to read. What we typically do is destructure our tuples in a pattern matching expression instead. This is why Scala has the notion of *tuple patterns*. They are similar to constructor patterns, which you saw earlier in this chapter, and are best illustrated by example. Let's add the following method to our `Greeting` singleton object:

```
def greetTraveller(name: String, distance: Int): String =
  (name, distance) match {
    case (_, 0) =>
      "Great decision, %s, not all meetings must be conducted face-to-face."
        .format(name)
    case ("Daniel", _) =>
      "Hi Daniel, glad you're taking the train again!"
    case _ if distance >= 1000 =>
      "I hope you can bear your flight shame somehow!"
    case _ =>
      "Hi %s, thanks for taking the train for such a short trip.".format(name)
  }
```

This method takes two parameters — a name and a travelling distance in kilometres. In the method body we create a tuple from them which we can conveniently process with a pattern matching expression. Like constructor patterns, tuple patterns contain nested patterns for the elements of the tuple.

In the first case, we use a literal pattern for the distance to match the case that the person doesn't travel at all. In the second case, we are using a literal pattern, `"Daniel"`, and the wildcard pattern because I always travel by train, regardless of the distance. The remaining two cases differentiate between those journeys that are shorter than 1000 kilometres and those with a distance of 1000 kilometres or longer, ignoring the name.

4.5 Processing sequences, the functional way

You have learned how to define functions, how to define your own data types as case classes, and how to modularise your code with classes, traits, and objects. And still, you probably feel like you're far away from writing any real world application in Scala.

One of the crucial missing pieces is that you still haven't learned how to work with collections. Almost every real-world application has to deal with collections of values. This is one reason why we are taking a first look at one Scala collection type here.

In this section, we will examine Scala's collection API as a declarative alternative to the traditionally imperative way of working with collections. The goal is not to walk through every method available, but to give you an impression of what it's like to work with Scala collections, and to whet your appetite. While we'll only work with sequences here, a lot of the methods you'll see are not exclusive to sequences but are also available on most other collection types.

Creating sequences

Before we start working with sequences, let's take a quick look at their definition. A simplified version looks like this[4]:

```scala
trait Seq[A]
```

As with tuples and other generic types, you need to fill in the type parameter in order to get a concrete `Seq` type. Examples of concrete types are `Seq[String]` or `Seq[Int]`.

The `Seq` trait has a companion object with an `apply` method, which has the following signature:

[4]The actual definition in the Scala source code of `Seq` is a bit different and contains concepts you don't need to be concerned with at this stage of learning Scala.

```
def apply[A](elems: A*): Seq[A]
```

This method has a type parameter as well, `A`. You will learn more about them in Chapter 7. The `elems` parameter is a *repeated parameter*, which means we can pass in any number of arguments of type `A`, even zero.

In Scala, you can define a repeated parameter by appending the parameter type with a `*` symbol. In the function body, that parameter will be accessible as a sequence of the specified parameter type. For example, let's say you have a method with the following signature:

```
def myMethod(s: String*): Int
```

In the body of that function, `s` has the type `Seq[String]`. If you already have a sequence and you want to pass all its elements to a function with repeated parameters, you can *expand* it. Say you have a `val xs: Seq[String]`. You can call `myMethod` like this:

```
myMethod(xs: _*)
```

With the `apply` method defined on the `Seq` object we can create any type of sequence. If we pass it a few `Int` values, we get a `Seq[Int]`. On the other hand, if we pass it one or more `String` values, we get a `Seq[String]`. We don't have to provide the type parameter to the `apply` method explicitly, as the compiler can infer it from the types of the values we pass in, as you will see in the example below.

Let's create a new sbt project called `music`, using the same project template as always:

```
$ sbt new dwestheide/minimal-scala-project.g8
[info] Loading global plugins from /home/daniel/.sbt/1.0/plugins
[info] Set current project to private (in build file:/home/daniel/)
[info] Set current project to private (in build file:/home/daniel/)
name [my-awesome-scala-project]: music

Template applied in /home/daniel/./music

$ cd music
```

In this project, we're going to start a new Scala REPL session in order to see how we can create new sequences using the `apply` factory method of the `Seq` companion object:

```
scala> val xs = Seq(1, 2, 3)
val xs: Seq[Int] = List(1, 2, 3)
```

This is interesting. The type of our sequence is `Seq[Int]`, that much is expected. However, the `toString` of the actual value is `List(1, 2, 3)`. Remember that `Seq` is a trait, and as such, abstract. The `Seq` companion object will have to choose a concrete implementation that extends the `Seq` trait, and apparently, it chooses `List`.

`List` is an immutable linked list, and it's the default `Seq` implementation chosen when creating a `Seq` using the `apply` factory method.

Creating sequences is all fine, but what we really want to do is process these sequences somehow. The Scala collections API provides a huge number of powerful methods that can broadly be categorized into transformations and aggregations. Transformations return a new, changed collection, while aggregations return a single value.

In our `music` sbt project, let's create a file `Track.scala` in the usual `src/main/scala` directory, containing a case class called `Track` that looks like this:

```
case class Track(
    title: String,
    artist: String,
    length: Int,
    genres: Seq[String]
)
```

A track has a a title, an artist, a length (in seconds), and a sequence of genres. So far so good. Now, let's create a file called `Music.scala` in which we define an object called `Music`. This object will contain a sequence of tracks that we will use to explore what the collections API has to offer. Here it is:

```
object Music {
  val tracks: Seq[Track] = Seq(
    Track("Leeds United", "Amanda Palmer", 286, Seq("Alternative Rock")),
    Track("The Sweetest Curse", "Baroness", 271, Seq("Hard Rock", "Metal")),
    Track("Blood To Gold", "Boy & Bear", 137, Seq("Alternative Rock")),
    Track("Laying Traps", "Crippled Black Phoenix", 285, Seq("Rock")),
    Track("Great White Bear", "Dear Reader", 240, Seq("Alternative Rock")),
    Track("Angry Planet", "New Model Army", 325, Seq("Alternative Rock")),
    Track("Happy Up Here", "Röyksopp", 164, Seq("Dance", "DJ")),
    Track("Náttmál", "Solstafir", 675, Seq("Metal")),
    Track("Within A Dream", "Tad Morose", 246, Seq("Hard Rock", "Metal")),
    Track("Hey Daydreamer", "Nick Halmstead", 204, Seq("Alternative Rock")),
    Track("Oasis", "Amanda Palmer", 127, Seq("Alternative Rock")),
    Track("The Legacy", "Iron Maiden", 563, Seq("Hard Rock", "Metal"))
  )
}
```

Transformations

Let's have a look at a number of transformations we may want to perform on our sequence of tracks.

Filtering tracks by Amanda Palmer

We have a sequence of tracks and want to get a sequence of only those tracks that are by Amanda Palmer. In imperative programming, a task like that involves a lot of code that is about iterating through a sequence, usually using a for loop of some kind. That code will always be the same, regardless of the specific filter criteria. In a functional language, we can use a method that abstracts over this common pattern. This is abstraction is possible because we can use functions as values. Here is the Scala solution:

```
scala> val palmerTracks = Music.tracks.filter(_.artist == "Amanda Palmer")
val palmerTracks: Seq[Track] = List(...)
```

The simplified signature of `filter` for a `Seq[A]` looks like this:

```
def filter(f: A => Boolean): Seq[A]
```

For our concrete type`Seq[Track]`, this means that the signature looks like this:

```
def filter(f: Track => Boolean): Seq[Track]
```

It expects a function that serves as the predicate to filter on — only the elements that satisfy this condition will be part of the resulting sequence.

If you need a refresher on functions as values that can be passed around, please have another look at Section 2.4 in Chapter 2 where we defined our own methods expecting functions to be passed in. The Scala collection library is making heavy use of this idea. Virtually every method expects a parameter that is a function.

Mapping to track titles

Mapping from a sequence of tracks to a sequence of track titles is not such an uncommon thing to do. You have probably been doing similar things in your applications many times. Again, when using an imperative approach, most of the code would not be about mapping from tracks to titles, but about iterating through a sequence using a for loop.

The functional alternative is the `map` method. Let's try this in the REPL:

```
scala> val titles = Music.tracks.map(_.title)
val titles: Seq[String] = List(Leeds United, ...)
```

That's much closer to what we intend to do, right? A simplified version of the `map` signature for `Seq[A]` is this:

```
def map[B](f: A => B): Seq[B]
```

We have a `Seq[A]` and expect a function of type `A => B`. The result is `Seq[B]`. The signature already tells you what the `map` method will do: apply `f` to each element in the source `Seq[A]`. There is no other meaningful implementation.

As with the `apply` method on the companion object, `map` has a type parameter: This is the `B` that shows up in the type of the function that `map` is expecting, and in the type of the value returned by `map`, a `Seq[B]`. As you can see from the example, we don't have to provide a concrete type here when calling `map`. It can be inferred from the function `f` we pass to `map`, whose return type is the type parameter `B`. We are going to discuss type parameters in detail in Chapter 7. For now, let's focus on processing sequences.

Mapping to flat sequence of genres

We want to transform our `Seq[Track]` into a `Seq[String]` containing all genres. Our first attempt is to use `map` again:

```
scala> val genres = Music.tracks.map(_.genres)
val genres: Seq[Seq[String]] = List(...)
```

This is not exactly what we want. Since the `genres` field itself is a `Seq[String]`, mapping a `Seq[Track]` like this yields a nested sequence, a `Seq[Seq[String]]`. To turn this into a flat sequence of genres, we can make use of the `flatten` method:

```
scala> val genres = Sequences.tracks.map(_.genres).flatten
val genres: Seq[String] = List(Alternative Rock, ...)
```

This method turns a nested `Seq[Seq[A]]` into a flat `Seq[A]`.

However, this can be simplified, by using the `flatMap` method instead. You will find this method to be ubiquitous in Scala, and functional programming in general, and part of a pattern that is far more generally applicable than just in the context of sequences. The simplified signature for a `Seq[A]` looks like this:

```
def flatMap[B](f: A => IterableOnce[B]): Seq[B]
```

This method will apply `f` to every element in the source `Seq[A]`. Unlike `map`, `flatMap` expects a function from `A` to `IterableOnce[B]`, instead of a function from `A` to `B`. `IterableOnce` is an even more general type of collection than `Seq`, further up in the type hierarchy, so every `Seq[B]` is also an `IterableOnce[B]`.

While `f` returns an `IterableOnce[B]` for every element, `flatMap` still returns a `Seq[B]` as the overall result.

Let's replace our previous solution with `flatMap` to get the result we want:

```
scala> val genres = Music.tracks.flatMap(_.genres)
val genres: Seq[String] = List(Alternative Rock, ...)
```

Sorting by track length

We want to sort our tracks by descending track length. To do that, we have a number of options, but we are going to go with the `sortWith` method. It expects a comparison function `(A, A) => Boolean` which decides whether its first argument should precede its second argument. If so, it returns `true`, otherwise `false`. Here is how we can use it:

```
scala> val byLength = Music.tracks.sortWith(_.length > _.length)
val byLength: Seq[Track] = List(Track(Náttmál,Solstafir,675,List(Metal)), ...)
```

Getting a distinct sequence of artists

Let’s say we need a sequence of artists based on the sequence of tracks, but each artist should only occur exactly once. Merely mapping the tracks to their artist is not enough. By calling the `distinct` method on the resulting sequence of artists, we make sure that each element in it is contained only once, based on value equality:

```
scala> val uniqueArtists = Music.tracks.map(_.artist).distinct
val uniqueArtists: Seq[String] = List(Amanda Palmer, Baroness,...)
```

Getting the two longest tracks

We have already seen how to sort tracks by descending length. Now we want to have a sequence of just the two longest tracks. This is what the `take` method will do, which has one parameter, the number of elements you want to take from the front of the sequence:

```
scala> val topTwo = byLength.take(2)
val topTwo: Seq[Track] = List(Track(Náttmál,Solstafir,675,List(Metal)), ...))
```

Adding an artist

Filtering, mapping etc. is nice, but it would be sad if we couldn’t add new elements to a sequence. Here is how we can create a new sequence by adding a new artist in front of an an existing sequence of artists. The existing sequence remains unchanged:

```
scala> val artists = Seq("Amanda Palmer", "Iron Maiden")
val artists: Seq[String] = List(Amanda Palmer, Iron Maiden)

scala> val moreArtists = "Solstafir" +: artists
val moreArtists: Seq[String] = List(Solstafir, Amanda Palmer, Iron Maiden)
```

What is this `+:` thing? It certainly looks like some special operator. However, as almost anything in Scala, it's a plain old method, defined on `Seq`, and we are using infix operator notation (see Section 3.5).

If this is a method defined on `Seq`, why is it that we write `"Solstafir" +: artists` and not `artists +: "Solstafir"`?

The thing is that because the name of our operator ends with a colon, it is right-associative. This means that `"Solstafir" +: artists` translates to `artists.+:("Solstafir")`.

In Scala, operators (methods used in infix notation) are *left-associative* by default. For example `3 - 1` translates to `3.-(1)`. There is only one exception: If an operator name ends with a colon (the `:` character), it is *right-associative* instead. This means that if we have a value `artists` of type `Seq[String]`, `"Solstafir" +: artists` translates to `artists.+:("Solstafir")`.

There are a few occasions where it makes more sense for an operator to be right-associative — like creating a new sequence by prepending a new element in front of an existing sequence. The main reason for this hard naming rule is to accommodate for operators like `+:` and `::` (which we don't use here), which are inspired by corresponding operators from other functional programming languages. If you need your own right-associative operator, having it end with `:` is the way to go.

You can also create a new sequence by appending a new element at the end of a sequence:

```
scala> val artists = Seq("Amanda Palmer", "Iron Maiden")
val artists: Seq[String] = List(Amanda Palmer, Iron Maiden)

scala> val moreArtists = artists :+ "Solstafir"
val moreArtists: Seq[String] = List(Amanda Palmer, Iron Maiden, Solstafir)
```

The `:+` operator is left-associative, because it does not end with a colon. We're not discussing complexity and O-notation in this chapter, but please be aware that appending at the end of a sequence can be expensive, depending on the concrete type of sequence.

Aggregations

There are plenty of other transformation functions available on sequences and the other collection types, but this should be enough to get you started, so let's explore some of the aggregation functions instead.

Does the sequence of artists contain Iron Maiden?

To check if a sequence contains a specific value, we can use the `contains` method:

```
scala> val containsMaiden = Music.tracks.map(_.artist).contains("Iron Maiden")
val containsMaiden: Boolean = true
```

This will do a value equality check against all elements in the sequence, until it finds one element that is equal to the specified value. Checking whether an element is contained in a collection is far more efficient for sets than for sequences. However, for now, we want to focus on a single type of collection.

Total number of tracks

To determine the total number of tracks, we use the `size` method:

```
scala> val numberOfTracks = Music.tracks.size
val numberOfTracks: Int = 12
```

Counting the number of metal tracks

We want to find out how many of the tracks in our sequence are tagged with the genre "Metal". We can use the `count` method for that, which expects a predicate `A => Boolean` and counts the number of elements for which the predicate returns `true`. Here is an example:

```
scala> val metalCount = Music.tracks.count(_.genres.contains("Metal"))
val metalCount: Int = 4
```

Do we have a track called "Leeds United"?

We want to know if there is at least one track in our sequence that is called "Leeds United". Here, the `exists` method comes in handy. Like `count`, it expects a predicate `A => Boolean`, but the result is a `Boolean`, which is `true` if the predicate we pass in is `true` for at least one element in our sequence:

```
scala> val yes = Music.tracks.exists(_.title == "Leeds United")
val yes: Boolean = true
```

Do all tracks have at least one genre?

The complementary function to `exists` is `forall`. It allows us to check if a predicate is `true` for *all* elements in the sequence, whereas the former checks if it is `true` for at least one element. We can use this to find out if all of our tracks have at least one genre. Like `exists`, `forall` takes one parameter, a function `A => Boolean`. Let's try it out:

```
scala> val yes = Music.tracks.forall(_.genres.nonEmpty)
val yes: Boolean = true
```

Here, we also make use of the `nonEmpty` method which checks if the sequence has at least one element. There is also `empty`, which checks if the sequence is, well, empty.

Side effects: Print all titles to the standard output

Let's say we want to print all track titles to the standard output, each in their own line. This means that we actually want to perform a side effect for each element in our sequence. This is where the `foreach` method comes in, which is defined as follows:

```
def foreach[U](f: A => U): Unit
```

The function `f` you pass in can return a value of any type, which the compiler will infer to be the type parameter `U`, but the overall return type of `foreach` is `Unit`, so any value that `f` may return will be discarded by `foreach`. In practice, you usually pass in functions of type `A => Unit` and can think of `foreach` as being defined like this:

```
def foreach(f: A => Unit): Unit
```

To print each element to a new line, we can first map our sequence of tracks to a sequence of titles (so its type is `Seq[String]`) and then pass a function literal to `foreach` in which we print a single title:

```
scala> Music.tracks.map(_.title).foreach(t => println(t))
Leeds United
The Sweetest Curse
Blood To Gold
Laying Traps
Great White Bear
Angry Planet
Happy Up Here
Náttmál
Within A Dream
Hey Daydreamer
Oasis
The Legacy
```

Exercises

1. Find out what the result is of calling `forall` on an empty sequence. Can you explain the reasoning behind this?
2. Find a way to split the sequence of tracks into those that have the genre "Metal" and those that don't, by calling a single method defined on `Seq[A]`. You're looking for a method that takes a predicate `A => Boolean` and returns a tuple of type `(Seq[A], Seq[A])`. Why is this function signature not good enough to find the correct method? Consult the API documentation[5] to find the correct one, and also try to see what other methods with the same signature do. By the way, you should totally bookmark the Scala API documentation[6]. It's really helpful, especially when you're still getting familiar with the Scala standard library.

4.6 More sugar with for comprehensions

In the last section you got your feet wet working with Scala collections, specifically with sequences. Let's stay with that for a while. Let's say you work at a startup company that tries to come up with ideas for interesting cover versions. Sounds like a promising business model, right?

[5]https://www.scala-lang.org/api/current/scala/collection/Seq.html
[6]https://www.scala-lang.org/api/2.13.3/

You want to find out all possible performances for a sequence of tracks, where a performance is the combination of a track and an artist performing that track. In other words, we are looking for the cartesian product[7] of artists and tracks.

Let's create a new source file called `Performance.scala` next to the `Music.scala` file in our `music` sbt project. In this file, we are going to define a case class called `Performance` and a companion object containing a function that computes all possible performances, given a sequence of tracks. This it what our initial version looks like:

```scala
case class Performance(title: String, artist: String)

object Performance {
  def performances(tracks: Seq[Track]): Seq[Performance] =
   artists(tracks).flatMap(
      artist => tracks.map(track => Performance(track.title, artist))
    )

  private def artists(tracks: Seq[Track]): Seq[String] =
    tracks.map(_.artist).distinct
}
```

Can you see why we are using `flatMap` here? The thing is that we have nested sequence transformations: first, we iterate through all artists. For each artist, we iterate through all tracks, in order to create a combination of the track title with the artist. Now, if we were using `map` on the `artists` sequence, we would end up with a `Seq[Seq[Performance]]`, which is not what we want. We are using `flatMap` for the transformation of the `artists` sequence, because the function we pass to `flatMap` returns a `Seq[Performance]`.

Now, you may think that this kind of nesting of `flatMap` and `map` calls is a bit difficult to read, and you are right. While the example above is still readable, it can get convoluted pretty quickly, especially in cases where you have to add one or two additional levels of nested `flatMap` calls. In this chapter we have already seen that Scala is not a sugar-free language, and now we are going to add some more sweetness to our linguistic diet in the form of *for comprehensions*.

[7] https://en.wikipedia.org/wiki/Cartesian_product

The simplest possible comprehension

Let's start with a small building block of what we want to achieve, getting the titles of all tracks:

```
for(track <- Music.tracks) yield track.title
```

For comprehensions start with the `for` keyword. In parentheses, we can bind each element of our sequence to an identifier, using the `<-` symbol. The expression `track <- tracks` is called a *generator*. Finally, we have to finish the for comprehension with the `yield` keyword, followed by an expression which has access to the bound identifier.

This comprehension is semantically equivalent to writing:

```
tracks.map(track => track.title)
```

As you can see, the expression you yield becomes a function passed to the `map` method.

We don't need to use parentheses though. The previous example can also be written like this:

```
for {
  track <- Music.tracks
} yield track.title
```

Here, we are using curly braces to create a block. Which of the two you choose is ultimately up to you, but as you will see now, the block version can be more readable for longer for comprehensions.

Multiple generators

To get to our cartesian products of artists and tracks we need a for comprehension with more than one generator — one for artists, another one for tracks. Let's see what this looks like by implementing our `performances` method with a for comprehension:

```
def performances(tracks: Seq[Track]): Seq[Performance] =
  for {
    artist <- artists(tracks)
    track  <- tracks
  } yield Performance(track.title, artist)
```

What does it mean to have two generators? The same as nesting a `map` call in a function passed to `flatMap`. We iterate through all artists, and for each artist, we iterate through all tracks. In the yield expression, which translates to a `map` call on the `tracks` sequence, we have access to a track (because we are mapping over the sequence of tracks) and to one `artist`, from the outer `flatMap` call on the `artists` sequence. Conceptually, this for comprehension is equivalent to the nested `flatMap` and `map` calls we initially used. I will repeat it here to make it easier to relate each element of the for comprehension to the corresponding element in the alternative notation:

```
def performances(tracks: Seq[Track]): Seq[Performance] =
  artists(tracks).flatMap(
    artist => tracks.map(track => Performance(track.title, artist))
  )
```

Guards

So now we got us a sequence of performances. But something is not quite right about our solution. We don't want to have all possible combinations of artists and tracks. All we want is a sequence of potentially interesting cover versions. This means that we need to remove those performances whose artist is the same as the artist on the `Track` value, which is the original performer of the track.

Like cases in pattern matching expressions, for comprehensions can have guards, and this is exactly what we need to solve our problem. Let's add a function called `coverPerformances` that looks like this:

```
def coverPerformances(tracks: Seq[Track]): Seq[Performance] =
  for {
    artist <- artists(tracks)
    track  <- tracks
    if track.artist != artist
  } yield Performance(track.title, artist)
```

This should look familiar — a guard is defined by the `if` keyword followed by a boolean expression. That boolean expression has access to all the identifiers bound in the left side of the generators that came before. In this case, we use both the `artist` and the `track` in our guard to specify that we are not interested in performances where the artist is the same as the one on the track. The guard can also be defined on the same line as a generator, so the following does exactly the same thing:

```
def coverPerformances(tracks: Seq[Track]): Seq[Performance] =
  for {
    artist <- artists(tracks)
    track  <- tracks if track.artist != artist
  } yield Performance(track.title, artist)
```

A guard gets translated to an additional call to the method `withFilter`, which has the same semantics as `filter`. The difference is that it is lazy instead of strict — it does not return a new sequence, but just a filtered view of the original sequence. Here is what it looks like:

```
artists(tracks).flatMap(
  artist =>
    tracks
      .withFilter(track => track.artist != artist)
      .map(track => (track.title, artist))
)
```

Let’s verify in the Scala REPL that our for comprehensions are actually doing what we think they do:

```
scala> val performanceCount = Performance.performances(Music.tracks).size
val performanceCount: Int = 132

scala> val coverCount = Performance.coverPerformances(Music.tracks).size
val coverCount: Int = 120
```

We have twelve songs by eleven different artists, so getting 132 performances sounds about right. If we want to exclude the performances by the original artist, we need to subtract one performance for each track, which sums up to twelve, leaving us with 120 cover performances.

For comprehensions can help a lot to make your functional code more readable. Whenever you start to nest multiple `flatMap` and `map` calls, you probably want to refactor to a for comprehension instead. Since for comprehensions are so ubiquitous in Scala, it's important to understand what is happening under the hood when you use them. As you will see later, processing collections is not the only use case of for comprehensions.

4.7 Type aliases

One of the most important aspects of writing code is making sure that its intent is clear — both to other developers on your team, and to your future self who has to come back to this particular piece of code in two months to fix a bug or add a new feature. Using meaningful names will certainly help with that. This is not just about naming your functions and fields, but also very much about giving meaningful names to your types.

Sometimes, there are existing types that you want to use, and there's not always a good reason to wrap them in a custom class just so you can give it a name that is more meaningful in the context of your application domain. This is why Scala allows you to define *type aliases*.

Let's go back to the `repl-yell` project and the `forBoth` function which we defined in the `Predicates.scala` file, back in Section 2.4:

```
def forBoth(x: Int, y: Int, cond: Int => Boolean): Boolean =
  cond(x) && cond(y)
```

As someone who wants to use this function and only sees its name and signature, you may be able to understand what this function is doing, and what the meaning of each of the parameters is. Sometimes, though, type aliases can make this easier, maybe even in this simple example. Here is how we define a type alias and use it:

```
object Predicates {
  type Condition = Int => Boolean
  def forBoth(x: Int, y: Int, cond: Condition): Boolean =
    cond(x) && cond(y)
}
```

You define a type alias with the `type` keyword, followed by a valid name of a type, and `=` symbol, and the type you want to define an alias for. In this case, we are defining an alias for a function type, but it can be for any other type. You need to define a type alias inside of a class, trait, or object, which is what we did by placing it in the `Predicates` object. Now, we can refer to `Condition` just like any other type:

```
scala> val small: Predicates.Condition = _ < 100
val small: Predicates.Condition = $$Lambda$4809/0x0000000801928840@43397388

scala> val yes = Predicates.forBoth(23, 42, small)
val yes: Boolean = true
```

A popular use of type aliases is to give a shorter, more succinct name to a long and complicated type. That being said, don't go overboard with type aliases. Also, simple wrapper types do have their own advantages and can be a better choice sometimes. When and how to use those, is one of the topics of Chapter 6.

4.8 No strings attached

Being an expressive language also means that it should be possible to build strings in a way that doesn't lead to an utterly unreadable mess. That's why I want to close this chapter by introducing a few features of the Scala language that help to make your string processing code less clunky: Multi-line strings and string interpolation.

Multi-line strings

Sometimes, you have longer dumps of text spanning multiple lines that you don't want to read from an external file, but instead put directly into your code. Instead of concatenating a bunch of strings, one for each line, and putting the `\n` escape character into them to mark a line break, it's much more convenient to make use of Scala's support for multi-line strings. Unlike normal strings, these are surrounded by triple-quotes.

Let's define a singleton object `MultilineStrings` in a new file `MultilineStrings.scala` in the `src/main/scala` directory of our `repl-yell` sbt project. In that object, we'll first define a string value created by traditional means, concatenating different strings:

```
object MultilineStrings {
  val concatenatedText = "Have you ever seen people use quotation marks " +
    "to emphasize a word?\n" +
    "This can be quite hilarious at times. Consider this example:\n\n" +
    "'We hope you \"enjoy\" your stay!'\n\n" +
    "Instead of quoting for emphasis, consider using italics."
}
```

This is not readable at all. We have to use to `+` method to concatenate a bunch of small strings, and in each of those strings, we must use the `\n` escape character in order to create a new line. In addition to that, any `"` characters in one of the strings need to be escaped, using the `\` symbol.

Let's add the following field to the `MultilineStrings` object, which aims to create the same string value, but using Scala's multi-line string feature instead:

```
val text =
  """Have you ever seen people use quotation marks to emphasize a word?
  This can be quite hilarious at times. Consider this example:

  'We hope you "enjoy" your stay!'

  Instead of quoting for emphasis, consider using italics."""
```

One of the nice things about triple-quoted strings is that inside of them you can use normal quotation marks without having to escape them. I really "enjoy" that! Moreover, we don't need to use escape characters to mark new lines. This means that the way we define the string looks pretty much the same as the result we are after — we describe what our result should be instead of writing some arcane instructions on how to assemble this result.

Now, printing this multi-line string in the REPL may not result in what you expect:

```
scala> println(MultilineStrings.text)
    Have you ever seen people use quotation marks to emphasize a word?
    This can be quite hilarious at times. Consider this example:

    'We hope you "enjoy" your stay!'

    Instead of quoting for emphasis, consider using italics.
```

Unfortunately, our text is indented, exactly the way we defined it, because everything after the opening triple-quotes is part of our string. To prevent that, there are two alternative solutions. First, you could unindent the text in your source code already:

```
object MultilineStrings {
  val text =
    """Have you ever seen people use quotation marks to emphasize a word?
This can be quite hilarious at times. Consider this example:

'We hope you "enjoy" your stay!'

Instead of quoting for emphasis, consider using italics."""
}
```

This works, but some people don't like this because we are kind of breaking out of the current level of indentation in our source code.

Alternatively, we can mark a margin on the left and then call the `stripMargin` method on our multi-line string, which will give us a string in which each line starts right after that margin marker:

```
object MultilineStrings {
  val text =
    """Have you ever seen people use quotation marks to emphasize a word?
      |This can be quite hilarious at times. Consider this example:
      |
      |'We hope you "enjoy" your stay!'
      |
      |Instead of quoting for emphasis, consider using italics.
    """.stripMargin
}
```

The symbol for marking the margin is the pipe character, `|`. Now, printing this multi-line string leads to the desired result:

```
scala> println(MultilineStrings.text)
Have you ever seen people use quotation marks to emphasize a word?
This can be quite hilarious at times. Consider this example:

'We hope you "enjoy" your stay!'

Instead of quoting for emphasis, consider using italics.
```

Having to add the margin symbol on each line can be a bit annoying, but if you choose to use an IDE, like IntelliJ IDEA[8], it will take care of that for you, so it's no big deal.

String interpolation

Throughout this book, we have created new `String` values a few times already. So far, we have been using the `format` method, which is slightly more readable than using simple string concatenation with the `+` method in cases where you need to create a `String` value from a mixture of static and variable content.

One of the occurrences of using string formatting was in the companion object of the `Position` class, in our `boardgame` sbt project:

```scala
object Position {
  private def assertInRange(value: Int, varName: String): Unit =
    require(
      value >= 0 && value < 20,
      "%s was %d, expected it to be >= 0 && < 20".format(varName, value)
    )
}
```

To be honest, I have had enough of it, and am happy to finally introduce string interpolations so that we can use them in the remainder of the book. You might know this concept from languages like Ruby or Kotlin. String interpolations use *processed strings* that are the result of processing Scala expressions inside of the string. Scala ships with a few different string interpolators. In this section, we're going to look at the simple interpolator, the raw interpolation, and the formatted interpolator.

[8] https://www.jetbrains.com/idea/

The simple interpolator

Here is how we can use the simple string interpolator to make our `String` construction code from above a little more readable:

```
object Position {
  private def assertInRange(value: Int, varName: String): Unit =
    require(
      value >= 0 && value < 20,
      s"$varName was $value, expected it to be >= 0 && < 20"
    )
}
```

Scala's simple, and standard, string interpolation is used by prefixing a string literal with an `s`. Inside of the string literal, you can then use an identifier prefixed by a `$`. The processed string will replace this with the result of calling `toString` on the value. It is also possible to put arbitrary expressions into an interpolated string, but these have to be surrounded by curly braces coming after the `$` symbol:

```
scala> val expression = s"23 + 42 = ${23 + 42}"
val expression: String = 23 + 42 = 65
```

So how does this string interpolation thing work? To understand, we need to look at a case class called `StringContext` which looks like this:

```
case class StringContext(parts: String*) {
  def s(args: Any*): String = ???
  def raw(args: Any*): String = ???
  def f[A >: Any](args: A*): String = ???
}
```

It's not the complete class definition, and I have left out the implementations of the three methods shown. What's relevant for now is the class parameter, `parts`, and the `s` method. The previous example actually gets expanded to this:

```
scala> val expression = StringContext("23 + 42 = ", "").s(23 + 42)
val expression: String = 23 + 42 = 65
```

A new `StringContext` is instantiated and it receives a sequence of parts. These are the parts before, in between, and after interpolated expressions. The `s` method, in turn, receives all the interpolated expressions. The simple interpolator will then interleave the static parts with the results of the expressions in order to concatenate the resulting processed string.

The raw interpolator

The simple interpolator described above supports the standard escape sequences that you can use in plain, unprocessed strings as well. For example we can put a new line into our string:

```
scala> val withNewLine = s"Hey\n${5 + 4}"
val withNewLine: String =
Hey
9
```

Sometimes, this is not what you want, but you actually want to represent the escape sequence itself in your string. The `raw` interpolator does not interpret escape sequences and should be used in these situations. Here is the same example, but using the `raw` interpolator:

```
scala> val rawString = raw"Hey\n${5 + 4}"
val rawString: String = Hey\n9
```

The formatted interpolator

Finally, `StringContext` provides a formatted interpolator. This one is like the simple interpolator, but additionally it supports Java formatting specifiers[9] like `%s`, `%f`, and so on — the same ones we have been using with the `format` method. You can put such a formatting specifier directly after an interpolated expression, and it will be used to format the result of that expression. Here is an example:

```
scala> val percentage = 103/7d
val percentage: Double = 14.714285714285714

scala> val successRate = f"Success rate: $percentage%2.3f percent"
val successRate: String = Success rate: 14.714 percent
```

What's great about this is that the formatted interpolator is able to do some compile-time checks to make sure that the formatting specifier and the interpolated expression have a matching type. For instance, we might accidentally pass in a `Boolean` value instead of a `Double` value into the previous string. This would not cause a runtime error, but fail to compile:

[9] If you are like me and cannot remember how the string formatting specifiers work, or you are not familiar with them to begin with, I highly recommend to look at Alvin Alexander's excellent cheat sheet: https://alvinalexander.com/programming/printf-format-cheat-sheet

```
scala> val percentage = true
val percentage: Boolean = true

scala> val successRate = f"Success rate: $percentage%2.3f percent"
<console>:13: error: type mismatch;
 found   : Boolean
 required: Double
       f"Success rate: $percentage%2.3f percent"
```

Remember that in the REPL, it's sometimes easy to think that a compile error is a runtime error, because both happen directly after another for each expression you type in. Here, however, we have a compile-time error. The reason that this is possible is that the formatted interpolator is implemented as a Scalo macro. Macros allow you to get access to the abstract syntax tree of your Scala source code and perform assertions on it or transform it at compile time. However, they are quite an advanced topic, and we're not going to look at how macros work in this book.

4.9 Summary

In this chapter, you were introduced to some of the Scala features that are crucial elements of what makes working with this language so enjoyable. From pattern matching, case classes, and tuples to declarative sequence processing, for comprehensions, and type aliases — all of these features help a lot when it comes to writing expressive, intention-revealing code.

Even string interpolation, which doesn't look too exciting at first, can do a great deal to make your code more readable. Apart from the benefit provided by the standard interpolators, there are numerous very exciting ways that Scala libraries and frameworks have been using their own custom interpolators, and the most powerful ones of those are also using macros to perform compile-time checks that a given literal string matches a certain format. Usage in the wild ranges from SQL interpolators[10] that provide a typesafe way of building SQL queries to a URL path and query parameter interpolators[11] used in a REST API routing component.

In the next chapter, we will look at the tools we have available for designing our program as a collection of cleanly separated modules that don't leak any of their implementation details.

[10] https://scala-slick.org/doc/3.3.1/sql.html
[11] https://www.playframework.com/documentation/2.7.x/ScalaSirdRouter

5. Tools for modular code

Divide and conquer is a crucial strategy for human beings. As programmers, this means that we don't put all our application logic into one big function. Instead, we extract parts of our logic into smaller functions and give them a meaningful name. Following that, we put those functions that belong together into a class, trait, or object. We do this because we are incredibly bad at keeping more than a few things in our working memory at the same time.

But this is not the end of the story when it comes to modularising your Scala code. In this chapter, we'll look at additional tools Scala offers you for structuring your program into cleanly separated modules with clear boundaries. We're going to look at inner classes, packages, imports and how to use package-level visibility modifiers to hide the internals of your modules. Finally, you'll learn about a concept called *package objects*.

5.1 Inner classes, objects, and traits

We have already learned that classes, objects, and traits can serve to modularize, organise, and namespace functions. In addition to that, singleton objects allow you to do the same for classes, traits, and other singleton objects — by defining certain classes, traits, or singleton objects inside of another singleton object, you effectively organise your classes and objects into namespaces. To illustrate this, we create a new sbt project and call it `fancymail`, based on the usual project template:

```
$ sbt new dwestheide/minimal-scala-project.g8
[info] Loading global plugins from /home/daniel/.sbt/1.0/plugins
[info] Set current project to private (in build file:/home/daniel/)
[info] Set current project to private (in build file:/home/daniel/)
name [my-awesome-scala-project]: fancymail

Template applied in /home/daniel/./fancymail

$ cd fancymail
```

In the `src/main/scala` directory, we'll create a file `SpamClassifier.scala` that looks like this:

```
trait SpamClassifier {
  def isSpam(mailText: String): Boolean
}
object SpamClassifier {
  object SpamWords {
    val Casino = "Casino"
    val Rolex  = "R0leX"
    val Poker  = "POkER"
  }
  class SimpleSpamClassifier(spamWord: String) extends SpamClassifier {
    override def isSpam(mailText: String): Boolean =
      mailText.contains(spamWord)
  }
}
```

Here, we define a trait `SpamClassifier` that acts as an interface for spam classification. In the companion object, we define an inner singleton object that contains a few constants for popular spam words. In addition, the companion object contains an inner class, `SimpleSpamClassifier`, containing a crude implementation of the `SpamClassifier` trait. We could also define inner traits, but in this example, we stick to inner objects and classes.

Now, in a new REPL session, we can access those inner types by prefixing their names with the outer type, separated by a dot:

```
scala> val casino = SpamClassifier.SpamWords.Casino
val casino: String = Casino

scala> val casinoClassifier = new SpamClassifier.SimpleSpamClassifier(casino)
val casinoClassifier: SpamClassifier.SimpleSpamClassifier =
  SpamClassifier$SimpleSpamClassifier@5bdaaa6e
```

All in all, defining inner classes, objects, or traits looks similar to defining type aliases in a singleton object, and in all these cases, a new inner type comes into existence.

Hiding implementation details

You already know about visibility modifiers for classes, traits, and objects. We can put these to use here, too. If we consider `SpamClassifier` to be the interface for our little module, we might want to hide the concrete implementation of our classifier. We can do this by making the inner class `SimpleSpamClassifier` private. This means that it can only be seen inside of the `SpamClassifier` trait and its companion object. If we consider the `SpamWords` object to be an implementation detail as well, we can mark it as `private`, too, and provide a few methods returning spam classifiers in different configurations:

```scala
trait SpamClassifier {
  def isSpam(mailText: String): Boolean
}
object SpamClassifier {
  def forCasino: SpamClassifier = new SimpleSpamClassifier(SpamWords.Casino)
  def forRolex: SpamClassifier  = new SimpleSpamClassifier(SpamWords.Rolex)
  def forPoker: SpamClassifier  = new SimpleSpamClassifier(SpamWords.Poker)
  private object SpamWords {
    val Casino = "Casino"
    val Rolex  = "R0leX"
    val Poker  = "POkER"
  }
  private class SimpleSpamClassifier(spamWord: String)
      extends SpamClassifier {
    override def isSpam(mailText: String): Boolean =
      mailText.contains(spamWord)
  }
}
```

Note that inside the `SpamClassifier` trait or object, we can access the `SpamWords` object without prefixing it with the full path. Instead of `SpamClassifier.SpamWords`, we can refer to it by a simple `SpamWords` — because `SpamWords` is defined in the `SpamClassifier` singleton object.

Now, it's no longer possible to access the `SpamWord` object or create an instance of the `SimpleSpamClassifier` in the REPL, as we used to before. Instead, we need to use one of the provided methods for getting a concrete `SpamClassifier` implementation:

```
scala> val classifier = SpamClassifier.forRolex
val classifier: SpamClassifier = SpamClassifier$SimpleSpamClassifier@6cee6627
```

By the way, it's totally possible for an inner singleton object to have inner classes, traits, objects, and type aliases as well. These can be as deeply nested as you like.

5.2 Packages

So far, you have seen how to organise your functions and types into namespaces or modules by putting them into classes or objects. Usually, however, using the approach of inner classes, objects, and traits is not enough and you want to organise these into somewhat larger units.

Like many object-oriented languages, Scala has a notion of *packages*. While packages look similar to their counterpart in Java, there are some noticeable differences.

Simple package declarations

However, let's start by looking at a simple `package` clause. Create a directory called `fancymail` in the `src/main/scala` directory of your sbt project. Inside, please add a file called `fancymail.scala` with the following content:

```
package fancymail.spam

object MoreSpamWords {
  val Casino = "Casino"
  val Rolex  = "R0leX"
  val Poker  = "POkER"
}
```

This syntax may look familiar to you. At the top of the source file, we use the `package` keyword, followed by the name of the package. As a result, all classes or objects in that source file belong to that package. We use dots to signify a hierarchy of packages. If you start a new REPL session, you will be able to access the `MoreSpamWords` object by prefixing it with its package:

```
scala> val casino = fancymail.spam.MoreSpamWords.Casino
val casino: String = Casino
```

Block syntax

The top-level `package` declaration is actually syntactic sugar for putting the things you want to belong to the respective package into a code block. So what we wrote above can also be written as:

```
package fancymail.spam {
  object MoreSpamWords {
    val Casino = "Casino"
    val Rolex  = "R0leX"
    val Poker  = "POkER"
  }
}
```

When using this block syntax, it's perfectly possible to put multiple packages into one source file:

```
package fancymail.spam {
  object MoreSpamWords {
    val Casino = "Casino"
    val Rolex  = "R0leX"
    val Poker  = "POkER"
  }
}
package fancymail.malware {
  object MalwareDetector
}
```

Nested packages

Unlike Java packages, Scala packages indeed form a hierarchy, the dots are not just for show. In fact, we can rewrite our previous example like this:

```
package fancymail {
  package spam {
    object MoreSpamWords {
      val Casino = "Casino"
      val Rolex  = "R0leX"
      val Poker  = "POkER"
    }
  }
  package malware {
    object MalwareDetector
  }
}
```

Just like before, we have a package `fancymail` containing two subpackages, `spam` and `malware`, which can both be accessed from the outside by their full name, `fancymail.spam` and `fancymail.malware`, respectively.

When nesting packages like this, however, you can access classes and objects from sibling packages by their relative package name, just like this:

```
package fancymail {
  package spam {
    object MoreSpamWords {
      val Casino = "Casino"
      val Rolex  = "R0leX"
      val Poker  = "POkER"
    }
  }
  package malware {
    object MalwareDetector {
      println(spam.MoreSpamWords.Casino)
    }
  }
}
```

Any classes or objects that are in a package further up in the hierarchy can be accessed without a qualifier:

```
package fancymail {
  object Logger {
    def info(msg: String): Unit = println(msg)
  }
  package spam {
    object MoreSpamWords {
      val Casino = "Casino"
      val Rolex  = "R0leX"
      val Poker  = "POkER"
      Logger.info("MoreSpamWords initialised")
    }
  }
}
```

Please note that packages are not closed in the sense that everything that you want to be in a certain package has to be defined in the same source file. For example, `fancymail.Logger` can be defined in a separate file, but you would still be able to access it unqualified if you declare the `fancymail.spam` package in the nested way you have seen in this section.

Let's demonstrate this, by first showing the content of a new file `Logger.scala`, which should be located next to the `fancymail.scala` directory.

```
package fancymail

object Logger {
  def info(msg: String): Unit = println(msg)
}
```

Even though this is in a separate source file, we can still access it in `MoreSpamWords`, because it is in a parent package of the one to which `MoreSpamWords` belongs. The `fancymail.scala` file should now look like this:

```
package fancymail {
  package spam {
    object MoreSpamWords {
      val Casino = "Casino"
      val Rolex  = "R0leX"
      val Poker  = "P0kER"
      Logger.info("MoreSpamWords initialised")
    }
  }
}
```

If you have multiple nested packages like this, and only the one furthest down in the package hierarchy actually contains any class or object definitions, you can use some syntactic sugar again to remove some visual noise. The example above is equivalent to writing this:

```
package fancymail
package spam

object MoreSpamWords {
  val Casino = "Casino"
  Logger.info("MoreSpamWords initialised")
}
```

What if you use the dot syntax for declaring the `fancymail.spam` package? From the outside, it doesn't matter whether you use block syntax, the slightly cleaner syntax above, or dot syntax. The `MoreSpamWords` object is accessible via the path `fancymail.spam.MoreSpamWords` in all cases. However, when using the dot syntax, you can't access types defined in the top-level `fancymail` package any more without a package qualifier. This means that we have to refer to `Logger` as `fancymail.Logger` now:

```
package fancymail.spam

object MoreSpamWords {
  val Casino = "Casino"
  val Rolex  = "R0leX"
  val Poker  = "POkER"
  fancymail.Logger.info("MoreSpamWords initialised")
}
```

Unlike some other languages, Scala does not require you to put your source files into specific directories that mirror the package structure — this wouldn't be possible, because one source file in Scala can contain multiple packages.

Nevertheless, when a file contains only classes, objects, and traits from single package, declared at the top of the file, it is common practice to put it into a a corresponding directory. For example, if we have a file `MyClass.scala` that contains a top-level package declaration `package myapp.domain` and a class `MyClass`, you would put that file into the directory `myapp/domain` inside of the `src/main/scala` directory.

5.3 Imports

So far, we have accessed classes, objects, or traits by their fully qualified package name or by a relative package name in case of nested packages. However, this can be a bit tedious. Like most languages, Scala also allows you to import classes, objects, traits, or packages, so that they can be used without providing the full package name all the time. In fact, you have already been using imports when we wrote tests using the `minitest` library.

This is usually not an exciting topic, but since Scala's import facilities are a bit more sophisticated than those in many other languages, it is worth discussing them briefly anyway.

We will start by making use of the `import` statement hands-on in the REPL. Let's import a package:

```
scala> import fancymail.spam
import fancymail.spam
```

After doing so, we are able to access classes, objects or subpackages without using the full name, but by prefixing them only with the name of the imported package:

```
scala> val words = spam.MoreSpamWords
val words: fancymail.spam.MoreSpamWords.type =
  fancymail.spam.MoreSpamWords$@3f52d92
```

We can also import a class, object, or trait directly:

```
scala> import fancymail.spam.MoreSpamWords
import fancymail.spam.MoreSpamWords
```

This will allow us to access it without any qualifier:

```
scala> val words = MoreSpamWords.Casino
val words: String = Casino
```

It's even possible to import fields or methods defined in a singleton object. For example, we can import the `Casino` field like this:

```
scala> import fancymail.spam.MoreSpamWords.Casino
import fancymail.spam.MoreSpamWords.Casino
```

Now, we we can access the `Casino` field without any package prefixes:

```
scala> val word = Casino
val word: String = Casino
```

Imports in Scala source files

Most of time, you don't write your code in the REPL, though. When you are working in Scala source files, you often have only a single package declaration at the top of a file, and you put your import statements directly below that.

Let's look at an example of that. We're going to add an `Email` case class to the `fancymail` package. We'll do that in a file `Email.scala` in the `fancymail` directory, next to the `Logger.scala` and `fancymail.scala` files:

```
package fancymail

import java.time.ZonedDateTime

case class Email(
    sender: String,
    subject: String,
    text: String,
    sentAt: ZonedDateTime
)
```

We put the package declaration at the top. This is followed by one or more import statements — in this example, we import the `ZonedDateTime` package from the Java standard library, which represents a date and time, together with a timezone. Next, we define one or more classes. Here, we just define a single case class.

Wildcard imports

Wildcard imports of everything contained in a package or object are also possible by using an underscore:

```
scala> import fancymail.spam.MoreSpamWords._
import fancymail.spam.MoreSpamWords._
```

Now we can access everything inside of `MoreSpamWords` without a qualifier:

```
scala> val word = Casino
val word: String = Casino
```

Wildcard imports can be convenient, but sometimes they bring types into scope that you don't want, because they clash with others. Use them with care!

Relative imports

Since we have previously imported the `fancymail.spam` package, importing all the fields of the `MoreSpamWords` object could have been done more succinctly, without using the full qualifier:

```
scala> import fancymail.spam._
import fancymail.spam._

scala> import MoreSpamWords._
import MoreSpamWords._
```

Multiple imports

It's also possible to import multiple classes, traits, objects, or packages from the same parent package without having to import all of them or having to put each of them into a separate `import` statement. To do that, provide a comma-separated list enclosed by curly braces:

```
scala> import java.util.{Currency, UUID}
import java.util.{Currency, UUID}
```

Here we are importing both `java.util.Currency` and `java.util.UUID` from the Java standard library. This can come in handy for big packages that include a lot of things you don't need, especially if one of those classes or objects uses a name that conflicts with one of your own types.

Multiple imports also work for fields or methods. For example, we could import both the `Casino` field and the `Rolex` field of our `SpamWords` object, but not the `Poker` field:

```
scala> import fancymail.spam.MoreSpamWords.{Casino, Rolex}
import fancymail.spam.MoreSpamWords.{Casino, Rolex}
```

Aliasing imports

Sometimes, you do need to import a type whose name conflicts with another one. For example, Scala has a `List` type, and the Java standard library has one as well. The Scala `List` type is imported automatically, but if you import Java's `List`, the former one will be shadowed and needs to be accessed by its full name. To avoid that, you can use Scala's aliasing feature during the import:

```
scala> import java.util.{ArrayList, List=>JList}
import java.util.{ArrayList, List=>JList}

scala> val javaList: JList[String] = new ArrayList()
val javaList: java.util.List[String] = []
```

Assigning a new name for an imported type is done using the arrow symbol `=>`. Now you can only use Java's `List` as `JList`, while Scala's `List` can still be used as usual.

The `_root_` package

Shadowing can occur in other ways as well. Let's create a directory `util` in the `src/main/scala` directory, and add a file `Fancifier.scala` to it:

```
package util

object Fancifier {
  def fancify(): Unit = println("Fancify")
}
```

We want to make use of our fancy `Fancifier` in our `mailer.Demon` object, but unfortunately, `mailer` also contains a `util` subpackage. Let's add a directory `mailer` in `src/main/scala`, and put a file called `mailer.scala` inside:

```
package mailer

import util.Fancifier.fancify

package util {
  object Purgatory
}
object Demon {
  fancify()
}
```

Unfortunately, this file doesn't compile. The compiler will complain that `Fancifier` is not defined in the package `mailer.util` — the top-level `util` package is shadowed.

Luckily, there is a way out: Every top-level package is implicitly a subpackage of the `_root_` package. This means that we can access the top-level `util` package as `_root_.util`, like this:

```
package mailer

import _root_.util.Fancifier.fancify

package util {
  object Purgatory
}
object Demon {
  fancify()
}
```

Now, our `mailer.scala` file compiles just fine.

Locally-scoped imports

One thing about imports in Scala that is different from most other languages is that you don't have to declare imports on a file level, as we did in the `mailer.scala` example above, and in the `Email.scala` file. Rather, imports can be declared in any code block as well (remember, code blocks are the chunks of one or more lines within curly braces). In the example, for instance, we can move the import into the body of the `Demon` object:

```
package mailer

package util {
  object Purgatory
}
object Demon {
  import _root_.util.Fancifier.fancify
  fancify()
}
```

Any other code block can have import statements, though, for example a multi-line function definition. The advantage of importing in code blocks is that the import is only valid locally, inside of that code block. If some package is only used in a small code block but would cause problems in the rest of the source file, for example due to name clashes, locally scoped imports are a good practice.

5.4 Package-level visibility modifiers

You already learned how to use visibility modifiers to hide the implementation details of your classes, traits, or objects. In this section, we'll look at package-level visibility modifiers. Together with Scala's support for nested packages as opposed to flat package structures, this allows for quite sophisticated ways of designing public APIs and hiding the internals of your module, making it easier to change the latter without breaking the expectations of client code.

Simple package privacy

It's often a good idea to hide implementation details from the clients of your API. One mechanism to prevent such details from leaking is to make your implementation classes invisible outside their package. Consider the spam classifier example, where we did exactly that by using private inner classes. Now, we'll look at how to do the same, but relying on package privacy instead.

As a first step, we want to put everything defined in the `SpamClassifier.scala` file into the same package. Currently, they are all part of the global root package because we haven't declared any package at all in this file. That's bad practice, so let's remedy that by adding a package declaration as the very first line to the `SpamClassifier.scala` file:

```
package fancymail.spam
```

We also want to mirror the package hierarchy in our directory structure, so let's make sure we have a directory `fancymail/spam` in `src/main/scala` and move the `SpamClassifier.scala` file into that directory.

For now, we're also going to remove the whole companion object, so that the `SpamClassifier.scala` file looks like this:

```
package fancymail.spam

trait SpamClassifier {
  def isSpam(mailText: String): Boolean
}
```

Now, we're going to add the `SimpleSpamClassifier` to the same file, but not nested inside a companion object. In order to hide it from users of our module, we'll make it visible only within the `fancymail.spam` package, using the `private` modifier:

```
package fancymail.spam

trait SpamClassifier {
  def isSpam(mailText: String): Boolean
}

private class SimpleSpamClassifier(spamWord: String) extends SpamClassifier {
  override def isSpam(mailText: String): Boolean =
    mailText.contains(spamWord)
}
```

Using the `private` modifier on a top-level class, trait, or object restricts its visibility to the package in which it's defined. In this case, to the `fancymail.spam` package.

Package privacy with nested packages

Sometimes you want to have the public API of your module and its implementation in separate packages. This is especially useful if your module is a bit bigger. In this case, people often like to have an `impl` package containing the internals of their module.

In most languages, having a package `fancymail.spam` and a package `fancymail.spam.impl` means that you have two packages that sit next to each other, because packages are flat. In Scala, packages are actually hierarchical, so if we have a package `fancymail.spam.impl`, it's a sub package of the `fancymail.spam` package.

Can we put this hierarchical structure to use to hide our spam module's internals? Let's add a `SpamClassifier` companion object again with a factory method, and let's move the `SimpleSpamClassifier` into an `impl` sub package:

```
package fancymail.spam

trait SpamClassifier {
  def isSpam(mailText: String): Boolean
}
object SpamClassifier {
  def wordBased(word: String): SpamClassifier =
    new impl.SimpleSpamClassifier(word)
}

package impl {
  private class SimpleSpamClassifier(spamWord: String)
      extends SpamClassifier {
    override def isSpam(mailText: String): Boolean =
      mailText.contains(spamWord)
  }
}
```

Unfortunately, this doesn't work, because our `SimpleSpamClassifier` is private to the `fancymail.spam.impl` package — it's not even visible in the `fancymail.spam` package. However, there is a way out of this. Whenever we use the `private` modifier, we can specify an upper package boundary in which the respective type, field, or method should be visible. For example, we can say that our `SimpleSpamClassifier` class should be visible in the `fancymail.spam` package and all its sub packages. To do that, all we need to do is put the desired package name in brackets after the `private` keyword:

```
package impl {
  private[spam] class SimpleSpamClassifier(spamWord: String)
      extends SpamClassifier {
    override def isSpam(mailText: String): Boolean =
      mailText.contains(spamWord)
  }
}
```

This looks very similar to the `private[this]` modifier you can use for fields and methods of classes, traits, and objects (see Section 3.3).

The package name you specify here cannot be just any package name, it has to be a package that the type, field, or method whose visibility you want to restrict is part of. Saying that this class is private to the `spam` package is possible because `fancymail.spam` is indeed a parent of our `fancymail.spam.impl` package.

Now, our implementation class is factored out into an implementation package. It's visible inside of that implementation package and up to the `fancymail.spam` package, where the `SpamClassifier` companion object needs to be able to instantiate it in the `wordBased` factory method.

5.5 Package objects

You know about packages as a way of namespacing and you know about singleton objects. While packages can contain classes, traits, and singleton objects, the latter can contain a few other things, like functions or type aliases. Sometimes, you want to make certain functions, type aliases, or implicit classes available everywhere inside of a package, and to everyone doing a wildcard import of things defined in that package.

Scala supports this by means of *package objects*. Every Scala package can have one such package object, which is a singleton object that can contain anything that any other singleton object can contain.

Let's go back to our `repl-yell` sbt project and create a new directory `predicates` in its `src/main/scala` directory. The `predicates` directory is supposed to mirror the package structure of our project. Let's put a file `package.scala` into that new directory that looks like this:

```scala
package object predicates {
  type IntCondition = Int => Boolean
}
```

We define a package object like any other singleton object. The only difference is that we need to prefix it with the `package` keyword. It is common practice to name the file containing the package object `package.scala` and put it into a directory corresponding to the package name.

Now that we have defined this package object, everything else defined in the package has direct access to the type alias we put into the package object. Let's put a file `Integers.scala` into the `predicates` directory. It will contain an `Integers` singleton object that is part of the `predicates` package:

```
package predicates

object Integers {
  def forBoth(x: Int, y: Int, cond: IntCondition): Boolean =
    cond(x) && cond(y)
}
```

Moreover, when we do a wildcard import of the package content, we have access to everything defined in there. In this case, it means that we can refer to the `IntCondition` type alias:

```
scala> import predicates._
import predicates._

scala> val even: IntCondition = _ % 2 == 0
val even: predicates.IntCondition = $$Lambda$4811/0x000000080194a840@3f49ee07
```

Please note that it is not a good strategy to put everything into a package object that's possible to put there. Most importantly, putting all of your regular classes, traits, and singleton objects into package objects is not a good idea, as it makes navigating your code quite difficult. Defining certain top-level functions and type aliases in a package object is usually fine.

Scala's predefined types

Now that you know about package objects, you can get a better understanding of where Scala's built-in types are coming from. One source is the package object of the `scala` package. It contains a ton of type aliases for common types from the Java and Scala standard libraries, and is automatically imported, so anything defined or aliased in there is automatically available.

For example, it contains a type alias for `Seq[A]`, pointing to the fully-qualified `Seq[A]` in the Scala collections package[1]:

[1]The actual definition of this type alias in the Scala source code looks slightly different. It makes use of advanced concepts of the type system that we can ignore for now.

```scala
package object scala {
  type Seq[A] = scala.collection.immutable.Seq[A]
}
```

This is a common trick in the `scala` package: Using type aliases not to invent new names for complex types, but in order to make commonly used types available without having to use the full package prefix or having to import them all the time.

A second source that's also always imported automatically is Scala's `Predef` object. It contains additional type aliases, and helper methods, which we're not going to look at in detail at this time.

Moreover, just like in Java, all types defined in the `java.lang` package are always available without requiring any imports.

5.6 Summary

In this chapter, we have looked at various tools the Scala language has available to help you organise your code into nicely separated modules, most notably nested packages combined with fine-grained visibility modifiers at the package level. We've also seen what Scala's import mechanism looks like and how it differs from those provided by many other languages — especially due to its support for locally-scoped imports and for aliasing imports.

In the next chapter, we'll move back from the topic of modular design to programming with classes in the small. We're going to look at how to make your programs more typesafe, with mostly zero runtime costs, and how to enrich existing types with new methods.

6. Bringing value to your classes

So far, when it comes to classes, you have learned about regular classes and about case classes. In this chapter, we're going to look at two more types of classes that can come in very handy in many situations. *Value classes* provide a mechanism to define wrapper types for existing data types with minimal cost at runtime. *Implicit classes* allow you to extend existing types with extra functionality. This feature is sometimes known as *extension methods* in other languages.

6.1 Value classes

If you look back at the parameters of the classes and methods we have written so far, you will see the type `String` pop up many times. You could also say that we have been pursuing a style of programming that is often called *stringly-typed programming*. It's a style that people look down to for good reasons.

One of the big issues with this style of programming is that the type `String` usually doesn't have any meaning in our domain. For instance, we named a field of the `Track` class `title`, but its type is `String`. For a reader of this code, it would be nicer if the domain concept of a `SongTitle` would also be reflected in the types.

One solution to that would be to define type aliases (see Section 4.7). This may help a human reader of the code, but the compiler will still treat any `String` as a `SongTitle`. As the name suggests, it is just an *alias*.

Many errors happen because we pass in parameters in the wrong order. For example, we may accidentally switch the `title` and `artist` arguments. This will lead to subtle bugs that may only be caught much later. Somebody in your organization will have to deal with this *bad data* mess and curse you. You can mitigate the problem to a certain extent with named arguments (see Section 4.1).

Yet, you can achieve an even stronger protection by having your own type for each concept from your application's domain. Sticking with the example, this is especially true if a song title has its own business invariants that need to be asserted even when it's used outside of the context of a track.

To illustrate how this can be achieved, let's start with a simple wrapper type called `SongTitle`, which we put into its own file called `SongTitle.scala` in the `music` sbt project:

```
case class SongTitle(value: String)
```

In a first attempt, we make this a case class. Now, we can change the type of the `title` field in the `Track` class as well:

```
case class Track(
    title: SongTitle,
    artist: String,
    length: Int,
    genres: Seq[String]
)
```

Please make sure to change all places where you create a `Track` accordingly, to make the compiler happy, for instance the place where the `tracks` sequence is defined, in the `Music` object.

Here is an example of how we create a `Track` instance now:

```
Track(
  SongTitle("Leeds United"),
  "Amanda Palmer",
  286,
  Seq("Alternative Rock")
)
```

Unfortunately, wrapping the title in a `SongTitle` class means that we are creating a lot of objects. At some point the garbage collector of the JVM needs to clean them up. Is this overhead necessary? Is the benefit worth it?

The good news is that we can have the best of both worlds. We can prevent unnecessary object allocations to some extent and enjoy domain-specific value types. To do so, we need to create our own *value classes*. Let's replace the `SongTitle` case class with a value class:

```
class SongTitle(val value: String) extends AnyVal
object SongTitle {
  def apply(title: String): SongTitle = new SongTitle(title)
}
```

Here are a few things to be aware of:

1. We define our own value class by extending the `AnyVal` type. This is the base type for all value types — both your own value classes and Scala's built-in value classes like `Int`, `Boolean` etc.
2. A value class must have exactly one class parameter, and that must be an immutable field, which is why we made the class parameter a `val`. The `val` can be made `private`, but not `private[this]`.
3. In the example, we created a companion object with an `apply` method. This is good style, and it will give you the chance to validate the argument before wrapping it, but it is not required, if you prefer to use the `new` everywhere. We'll look into validation a bit later in this section.
4. You cannot override the `equals` and `hashCode` methods in value classes — these will always delegate to the `equals` and `hashCode` methods of the underlying value (in our example, the wrapped `String` value). Nevertheless, if you compare an instance of a value class with a value of the unwrapped type using the `==` operator, the result will always be `false`. After all, a `SongTitle` cannot be equal to a `String`.
5. The body of a value class can only contain method definitions.

Why the restrictions of one parameter and no other value definitions? What you gain from value classes is that under many circumstances, instances of your wrapper class will not be created at runtime. You gain the compile-time safety of your new type. Yet, instances of this type will be represented as values of the wrapped type — `String`, in our example. Only in specific situations, for example when used in generic code, the wrapper classes will have to be instantiated, but this boxing and unboxing happens automatically and transparently.

A value class can contain methods, but not other value definitions. For example, we can define a method `show` like so:

```scala
class SongTitle(val value: String) extends AnyVal {
  def show: String = s"SongTitle[$value]"
}
```

How is it possible that we can define methods in a value class and call them, and at runtime, there will still not be any instance of our wrapper value class? Each method you define on a value class gets compiled to a method on the companion object of the value class. If no companion object exists, the compiler will create one. This method on the companion object has a slightly different signature. The first parameter will always be called `$this` and have the type wrapped by the value class — in our example, `String`. Method calls on an instance of a value class get translated to method calls on its companion object. The value wrapped by the value class is passed in as the first parameter, `$this`.

This means that in our Scala code, we can define a value of type `SongTitle` and call the `show` method on it, like this:

```scala
object SongTitleExperiment {
  val title = SongTitle("Daniel")
  val value = title.show
}
```

No `SongTitle` object will be actually be created, and the `show` call will be translated to something like this:

```scala
SongTitle.show$extension("Daniel")
```

Here, `show$extension` is a static method defined in the `SongTitle` class, expecting a `String`, which is the value wrapped by this class, from a Scala perspective. It's what our non-static, parameterless method in our Scala class `SongTitle` gets translated to by the compiler.

Built-in value classes

We have already seen in the previous chapter that even values of Java's primitive types are objects in Scala, and you learned that they are still represented as primitive values at runtime, unless they need to be boxed, for example when you put them into a collection. Scala's built-in types can be divided into numeric types and non-numeric types.

- Numeric types: `Double`, `Float`, `Long`, `Int`, `Char`, `Short`, `Byte`
- Non-numeric types: `Boolean` and `Unit`

We have already been using many of these types in this book. Unlike your own custom value classes, these built-in value classes cannot be instantiated with the `new` keyword — you have to use literals instead.

The `Unit` is an even more special snowflake than the other built-in value classes because only a single value of this type exists: `()`. It's not often that you have to use this value in your code, but still good to keep in mind that this exists.

Enforcing business invariants

One reason to introduce types for concepts from your application's domain is that it helps you avoid bugs. For example, you're less likely to mix up the order of arguments, using an artist name where the method expects a song title.

Another benefit is that you can put all the business invariants for a specific domain concept into one place: the constructor or factory method for your type. If you know that a value of type `SongTitle` always represents a valid song title, according to the invariants imposed by your domain experts, there is no need to be all defensive about it in a gazillion of places. You no longer need to check whether that `String`-typed parameter representing a song title violates your invariants for a song title everywhere it is used. Instead, you can rest assured that it has already been validated.

Let's see how we can do this with our `SongTitle` value class. We are going to put the code enforcing our business invariants for `SongTitle` into the factory method in the companion object:

```
object SongTitle {
  def apply(title: String): SongTitle = {
    require(title.nonEmpty, "title must not be empty")
    require(title.size < 255, "title must be shorter than 255 characters")
    new SongTitle(title)
  }
}
```

You have already seen the `require` method in previous chapters. If the `requirement` is `true`, it will do nothing. Otherwise, it will throw an `IllegalArgumentException` with the given message. As already mentioned, throwing an exception is not the best way for asserting business invariants, but for now, it's the only way we have.

Now, our `apply` method will throw an exception if we try to create invalid `SongTitle` instances:

```
scala> SongTitle("")
java.lang.IllegalArgumentException: requirement failed: title must not be empty
  at scala.Predef$.require(Predef.scala:277)
  at SongTitle$.apply(Sequences.scala:12)
  ... 36 elided
```

Why put the `require` statements in the factory method instead of the constructor? As already mentioned, you can't put anything into the class body of a value class apart from method definitions.

Private constructors

Note, however, that there is still a loophole: It's still possible to use the constructor of the class directly:

```
scala> val title = new SongTitle("")
val title: SongTitle = SongTitle@0
```

Fortunately, we can hide the constructor so that it's only visible to the `SongTitle` companion object. All we have to do is mark the constructor as private:

```
class SongTitle private(val value: String) extends AnyVal
```

Now the `SongTitle` companion object is the only one that can create new instances of our `SongTitle` value class. Every instantiation needs to go through our `apply` factory method, which will not allow you to create invalid values of this type. In Chapter 9 we'll examine how to achieve the same effect without throwing exceptions.

Exercises

1. Improve `Track` by introducing the value classes `Artist` and `Genre`.
2. Introduce another value class `Length` and a factory method `fromSeconds` in the companion object. Make sure that it's impossible to create `Length` values where the value in seconds is lower than 1.
3. When creating `Track` instances, why will there still be object allocations for the `Genre` value class?

6.2 Extension methods

Sometimes, you run into situations where you would like to have additional methods available on existing types that you don't control. However, since you can't change the source code, you cannot go ahead and add them. What is a programmer supposed to do?

In some dynamically typed programming languages, the solution is a technique called monkey patching[1], which allows you to re-open existing classes and change them — by adding methods or replacing existing ones so that they behave differently. This is a rather dangerous approach and needs to be followed with great care. A common pitfall is that it's difficult to understand where the patched behaviour is coming from as well as to control which patches you'd like to be in effect.

Other languages, like C# or Kotlin, follow a more principled approach with a technique called extension methods[2]. Essentially, these are static methods that can be called like instance methods. The technique is similar to how methods defined on custom value classes get compiled to methods on its companion object in Scala, with a reference to the object on which the method is called being passed in as an additional argument (see Section 6.1 above). These extension methods are allocation-free, which means that it does not result in the creation of any wrapper objects.

While Scala does not have built-in language support for extension methods, it does allow you to express this concept by means of other language features.

[1]https://en.wikipedia.org/wiki/Monkey_patch
[2]https://en.wikipedia.org/wiki/Extension_method

Implicit classes

Let's say we want to represent the length of a track with a more meaningful type than `Int`, and we are going to use the `Duration` class from the `java.time` package that was introduced in Java 8. However, we want to have some syntactic sugar for creating instances of these classes, so that we can write the following:

```
val trackLength: java.time.Duration = 283.seconds
val anotherTrackLength: java.time.Duration = 3.minutes
```

There is already a `Duration` type in the Scala standard library with support for creating instances of it using exactly this syntax. However, we don't want to use that type. Firstly, it's defined in the `scala.concurrent` package, and our `Track` class is part of our domain model, where we don't want to deal with types that represent technical concepts like concurrency. The `Duration` type from the `java.time` package is a better fit for our purpose. Secondly, the learning effect is even better if we build this on our own instead of just looking at what the Scala standard library has done in the `scala.concurrent` package.

We are going to approach this problem by first creating a wrapper for `Int`. Please create a new file `DurationSyntax.scala` in your `music` sbt project, and add a singleton object called `DurationSyntax` to that file, like this:

```
object DurationSyntax {
  // add wrapper type and conversion methods here
}
```

Inside our `DurationSyntax` object, we are going to define a wrapper class for `Int`, like this:

```
import java.time.Duration

class ExtendedInt(x: Int) {
  def seconds: Duration = Duration.ofSeconds(x)
  def minutes: Duration = Duration.ofMinutes(x)
}
```

We can already use this wrapper class, as demonstrated in this new REPL session:

```
scala> import DurationSyntax._
import DurationSyntax._

scala> val trackLength = new ExtendedInt(283).seconds
val trackLength: java.time.Duration = PT4M43S
```

We first need to import our `ExtendedInt` type, and we are doing this by simple importing everything from the `DurationSyntax` object. Unfortunately, we don't get any advantage yet over calling the factory method defined on `java.time.Duration`. How can we improve?

We are going to use a language feature called *implicit classes*. An implicit class is a class that is prefixed with the `implicit` keyword. In our case it looks like this:

```
object DurationSyntax {
  import java.time.Duration

  implicit class ExtendedInt(x: Int) {
    def seconds: Duration = Duration.ofSeconds(x)
    def minutes: Duration = Duration.ofMinutes(x)
  }
}
```

Since an implicit class is designed to wrap a value of another type, it can only have a single class parameter, which is the value we want to wrap — in our example, a value `x` of type `Int`. If you try to define it with multiple class parameters, or none at all, you will get a compile error.

Whenever a method that is not defined on type `A` is called, the Scala compiler will look for an implicit class that fulfils these two conditions:

1. It wraps a value of type `A`
2. A method with the expected name and signature is defined on this implicit class

If the compiler finds such a class, it will create an instance of it and pass the value to be wrapped to its constructor. Hence, `283.seconds` gets expanded to:

```
new ExtendedInt(283).seconds
```

This is because the `seconds` method doesn't exist on `Int`, but there is an implicit class `ExtendedInt` that acts as a wrapper for an `Int` value, and this class happens to have a `seconds` method.

The good thing is that this conversion will not just magically be applied anywhere, unlike extensions done via monkey patching, for example. In order for an implicit class to be taken into account by the compiler, it has to be in scope. A more detailed explanation of what that means, and how the Scala compiler looks for implicits will be in the second book. If you're curious, you can also find more information in the FAQ on finding implicits[3] in the official Scala documentation.

For now, though, it's enough to know that you can put an implicit class into scope by importing it. Hence, in order to get the previous code snippet to compile, we need to modify it so that it looks like this:

```
scala> import DurationSyntax._
import DurationSyntax._

scala> val trackLength = 283.seconds
val trackLength: java.time.Duration = PT4M43S
```

Now, `283.seconds` gets expanded, as previously described.

This brings us to one additional restriction of implicit classes: They can't be defined at the top-level, like normal classes or case classes. Instead, they always need to be defined inside of another singleton object, class, or trait, so that they can be brought into scope by importing them. This is what we did when we placed it in the `DurationSyntax` singleton object.

While implicit classes can be a powerful technique to extend existing types, it's important to keep in mind to use them sparingly. Even though they are more principled than monkey patching, having lots of implicit classes in scope can cause confusion and slow down the compiler.

Implicit value classes

There is still one problem with the technique for defining extension methods we have explored above. Implicit classes give us a convenient, relatively boilerplate-free syntax for it. However, under the hood, this is still creating a new instance of `ExtendedInt`, our wrapper class, every single time we call one of the extension

[3]https://docs.scala-lang.org/tutorials/FAQ/finding-implicits.html

methods we defined. This is not the case with extension methods in C# or Kotlin. Didn't I promise you we could achieve the same thing in Scala? The good news is that we can and that we are almost there.

Remember how value classes work? They wrap a single value of another type. In the Java byte code, they will usually be represented as the plain, unwrapped type, so no instance of your value class gets created. You can also define methods on them, which get translated to methods on the companion object taking the wrapped value as their first parameter — much like extension methods in other languages. Even though no instance of your value class gets created in the resulting byte code, in your Scala code you still need to create it. We had to do that for our `SongTitle` value class, like so:

```scala
val title = new SongTitle("Oasis")
```

Implicit classes also take exactly one class parameter. What happens if you make an implicit class a value class as well? In the `DurationSyntax` singleton object, let's change our `ExtendedInt` definition so that it is also a value class:

```scala
import java.time.Duration

implicit class ExtendedInt(private val x: Int) extends AnyVal {
  def seconds: Duration = Duration.ofSeconds(x)
  def minutes: Duration = Duration.ofMinutes(x)
}
```

All we had to do was make our `ExtendedInt` class extend `AnyVal` and, because value classes require this, make the single class parameter a field by prefixing it with the `val` keyword.

Now, our extension methods are entirely allocation-free. Whenever we call one of them, for example by writing `283.seconds`, this does not create a new instance of `ExtendedInt`. Instead, it will translate to something like this:

```scala
ExtendedInt.seconds$extension(283)
```

In the second book, you will see that it's not always possible to turn your implicit classes into value classes. Nevertheless, in many cases it is, and then it's a great technique for defining allocation-free extension methods.

Implicit classes and package objects

You learned that implicit classes cannot be defined at the top-level, but only as nested classes inside of another class, trait, or singleton object. Since package objects are just a special kind of singleton object, this means that you can also put your implicit classes into a package object. For example, we can create a directory `duration` in our `src/main/scala` directory, containing a `package.scala` file that looks like this:

```scala
package object duration {

  import java.time.Duration

  implicit class ExtendedInt(private val x: Int) extends AnyVal {
    def seconds: Duration = Duration.ofSeconds(x)
    def minutes: Duration = Duration.ofMinutes(x)
  }
}
```

Now, the extension methods provided by this implicit class are available whenever you import this class from the `duration` package:

```
scala> import duration._
import duration._

scala> val songLength = 234.seconds
val songLength: java.time.Duration = PT3M54S
```

When it comes to defining implicit classes in package objects, you have to be a bit careful. If you have a package containing the classes, traits, and singleton objects that constitute the public API of your component or library, people will likely want to do a wildcard import of that package. If the package object for that package contains a lot of implicit classes, people doing the wildcard import have no way of opting out of those. If these implicit classes are a crucial part of your public API, that may be exactly what you want. But are they? If they are only nice to have, consider putting them into a separate package, or into a regular singleton object inside of the respective package. This way, consumers can opt-in to these implicit classes. If they are only meant to be used internally, by all means put them into a separate implementation package that consumers will never ever see.

Exercises

1. Create a new implicit value class `ExtendedDuration` that wraps the `Duration` class and add an extension method to it with the following signature: `def +(other: java.time.Duration): java.time.Duration` The goal is to return a `Duration` that is the sum of the wrapped duration and the one passed to the `+` method.
2. Try this extension method out in the REPL, adding two durations with the `+` method in infix notation.

6.3 Summary

In this chapter, you learned about value classes as a mechanism for making your code more type-safe, more readable, and less defensive, by putting the invariants for the respective domain concept into one place. In most cases, this happens without incurring any runtime costs in terms of additional object allocations. Moreover, we introduced a principled approach for adding new methods to existing types whose source code you don't control, using implicit classes.

In the next chapter, we're going to take a first glimpse at Scala's powerful type system. You'll learn what things like `Seq[A]` or `Array[A]` are about, and you'll learn how to write methods that abstract over the concrete types of their parameters. Moreover, we'll take the concept of implicit classes one step further and look at how to write extension methods not to a single concrete type, but to arbitrary types.

7. Scala type system 101

Using a statically typed language can have a lot of benefits. They all boil down to the fact that you get a lot of tests for free, and those tests happen at compile time. Yet, to have any of these benefits, you also need a powerful type system.

Scala happens to be such a language, and in this chapter, we will take a first glance at its type system. There may be some new terminology in here, but it's less frightening than it may seem. In particular, you are going to learn about *type constructors* and *polymorphic methods*. This is not all there is to know about Scala's type system. But it's enough to get you up and running. In the second book, we are going to examine the more advanced aspects of Scala's type system.

7.1 Type constructors

If you are coming from a language like Java, C++, or Kotlin, you are most likely familiar with generic classes — classes that take type parameters. In Scala, classes and traits can be parameterized with types as well. In fact, I haven't been able to hide this language feature from you entirely until now.

You have seen `Array[Char]` and `Array[Byte]` in Section 1.8, and `Array[String]` in Section 3.2, when we looked at the signature of the `main` method that serves as an entry point to a Scala application. You also came across tuples in Section 4.4, which have type parameters as well. For example, we created tuples of type `(String, Int)`, or rather `Tuple2[String, Int]`.

To understand generic classes and traits, we are going to look at the `Seq` trait again. A simplified version of it looks like this[1]:

```
trait Seq[A]
```

Like `Array`, `Seq` has one type parameter. What we can learn from this type signature is that type parameters follow the name of the trait or class, surrounded by

[1]As mentioned in Section 4.5, the actual definition in the Scala source code of `Seq` looks slightly different. However, in the context of this book, we can ignore those differences.

square brackets. Often, we use a single uppercase letter, but that is not required. Sometimes, a word is a better choice to convey the meaning of the type parameter. Then, it's best practice to use camel case notation.

A generic class or trait can have more than one type parameter. These are separated by comma. An example of that is `Map`. Like `Seq`, this is a collection data structure. A map associates keys to values. In some languages, you'll know it by the name hash, or hash table. Here is a simplified definition of `Map`[2]:

```
trait Map[K, V]
```

`Map` has two type parameters. Often, we name type parameters starting with `A`, then `B`, and so on. `Map` is an example where more meaningful names were chosen, albeit still only using single letters. Since the first type parameter represents the key type of the map, and the second one its value type, they are named `K` and `V`, respectively.

The idea of generic classes and traits is that these do not have to care about the specific type they are parameterized with — they are implemented in a completely generic way. As such, type parameters are a great way of achieving reusability. When defining generic classes or traits with type parameters, we also talk about *parametric polymorphism*.

Before we continue, let's start with a new sbt project called `type-system-101`, which we create with the `sbt new` command:

```
$ sbt new dwestheide/minimal-scala-project.g8
[info] Loading global plugins from /home/daniel/.sbt/1.0/plugins
[info] Set current project to private (in build file:/home/daniel/)
[info] Set current project to private (in build file:/home/daniel/)
name [my-awesome-scala-project]: type-system-101

Template applied in /home/daniel/./type-system-101

$ cd type-system-101
```

Let's start a new REPL session from that project's interactive SBT shell.

When we create a sequence, a concrete type must be provided for each type parameter:

[2]For the same reasons as for `Seq[A]`, I have removed aspects of the signature of `Map` that require you to know about certain advanced aspects of the Scala type system.

```
scala> val xs = Seq(1, 2, 3)
val xs: Seq[Int] = List(1, 2, 3)
```

As demonstrated in the example, that type, `Int` in this case, is also placed in square brackets after the name of the trait or class. In this case, it could be inferred from the types of the values we passed to the `apply` factory method defined on the `Seq` companion object. We can also be explicit about the type of our `xs` identifier:

```
scala> val xs: Seq[Int] = Seq(1, 2, 3)
val xs: Seq[Int] = List(1, 2, 3)
```

But what has all of this got to do with type constructors? And what are type constructors supposed to be anyway?

We are getting there, trust me. To do that, we have to go back to our `SongTitle` class from `music` sbt project, which looks like this:

```
class SongTitle private(val value: String) extends AnyVal {
  def show: String = s"SongTitle[$value]"
}

object SongTitle {
  def apply(title: String): SongTitle = {
    require(title.nonEmpty, "title must not be empty")
    require(title.length < 255, "title must be shorter than 255 characters")
    new SongTitle(title)
  }
}
```

The `SongTitle` class has a constructor that takes one parameter of type `String`. Calling this constructor results in a new *value* of type `SongTitle`. We can also say that the `apply` method defined on the companion object is a factory method. If it were a function value, we would write its type like this:

```
 String => SongTitle
```

In other words, it takes one `String` parameter, and returns a value of type `SongTitle`. Constructor functions like this, or any Scala functions, operate on the value level — their parameters are values, and they return a value.

Now, let's move from the value level to the type level. Clearly, `SongTitle` is also a type. But what about `Seq`? The `Seq` trait has a type parameter, for which you need to pass in a concrete type, but it's not itself a type. Instead, you can think of it as a function on the type level. More specifically, think of it as a function that takes one parameter (which is a type) and returns a type — it's a type constructor. Type constructors declare their parameters in square brackets, and get applied by passing arguments, which are concrete types, in square brackets.

By passing the type `String` to the `Seq` type constructor, we get a `Seq[String]`, which is a concrete type. By passing the type `Int` to the `Seq` type constructor, we get a `Seq[Int]`, which, again, is a concrete type.

And that's all there is to know about type constructors, for now. You will be using them a lot — especially, but not only, when working with collection types.

Exercises

1. In the `type-system-101` sbt project, add a new case class `Entry` in a new Scala source file in `src/main/scala`. This class should have two type parameters and one field of each of the two types, named `key` and `value`. It'll be our own version of `Tuple2`, basically.
2. Add a method `swap` to that case class that returns a new instance of `Entry` with the keys and values swapped.
3. In the REPL, what happens if you create a new sequence `Seq(1, 2)` and assign it to an identifier of type `Seq[String]`?

7.2 Polymorphic methods

In the previous section, you already learned about parametric polymorphism, but only in the context of generic classes and traits. However, it is not limited to that. Methods can be polymorphic as well. This is the case if their signature contains type parameters, like with generic classes or traits. In fact, we have already seen and used polymorphic methods when we took a first glimpse at how to work with sequences in Scala (see Section 4.5). Back then we were kind of ignoring the details though of what these type parameters actually mean and how they work. In this section, we will make up for that and approach the concept from the beginning.

The merits of generic programming

Once again, let's go back to the `forBoth` method we originally implemented in Section 2.4, in the `repl-yell` project:

```
def forBoth(x: Int, y: Int, cond: Int => Boolean): Boolean =
  cond(x) && cond(y)
```

This method allows us to check that an arbitrary condition holds true for two values. This is already pretty powerful. It's only possible because we can make use of functions as values — the third parameter of the `forBoth` method is a function.

So what's the problem with this method? If you look at its signature, you will notice that it only works for values of type `Int`. That's too bad. It would be nice if we could have a `forBoth` method for `String`, `Char`, or any other type we don't even know about yet. For sure we don't want to implement it again and again. In a nutshell, we want to have a reusable version of this method.

Looking at the implementation of the `forBoth` method, we discover something interesting. There is nothing in it that makes use of the fact that `x` and `y` are of type `Int`. It doesn't care, because all it does is apply the passed in `cond` function to the two passed in values.

So how can we make our method more generic, and completely ignorant of the types of its parameters? The answer is: by introducing a type parameter and making it polymorphic. A method can have one or more type parameters, and we have to put them in square brackets, directly after the name of the method. If there is more than one type parameter, they have to be separated by commas. In this case, though, we only need one type parameter.

Let's add a new file `Generics.scala` in the `src/main/scala` directory of our `type-system-101` project. In that file, we're going to define a singleton object `Generics` containing our refined `forBoth` method:

```
object Generics {
  def forBoth[A](x: A, y: A, cond: A => Boolean): Boolean =
    cond(x) && cond(y)
}
```

As the example shows, we can refer to type parameters in the parameter list of our method. We can also refer to them in the result type of our method, but that's not necessary in this case. All we had to do was replace any occurrence of `Int` with the name of our type parameter, `A`.

Now, our `forBoth` method is usable with any type we can imagine, or not imagine yet — it's polymorphic. This is the power of *generic programming*, which describes programming without knowledge about concrete types. This can happen by means of polymorphic methods, like our revised `forBoth` method, and generic traits or classes.

Apart from increased reusability of our code, there is another fascinating property of generic programming: By abstracting over concrete types, we are constraining ourselves. These constraints are actually a good thing. If we don't know what concrete type `x` and `y` are in our `forBoth` method, the space of possible implementations of our method becomes much smaller. We can let the types guide our implementation, so that it becomes easier to reason about what the method is supposed to do, and less likely to implement it the wrong way.

Optimising for type inference

With the original version of `forBoth`, as defined in the `repl-yell` sbt project, we were able to do the following:

```
scala> val notReally = Predicates.forBoth(2, 3, _ > 3)
val notReally: Boolean = false
```

Remember, `_ > 3` is a function literal, and a shortcut for `x => x > 3`. If we try the same with our new version of `forBoth`, though, we get a rather cryptic compile error:

```
scala>  val notReally = Generics.forBoth(3, 5, _ > 3)
                                               ^

       error: missing parameter type for expanded function
       ((<x$1: error>) => x$1.$greater(3))
```

The reason for this error is that at the time when you pass the function literal to the `forBoth` method, the Scala compiler has not yet inferred the type of the method's type parameter `A`. Without knowing the concrete type for this type parameter, it doesn't know either what's the concrete type of the function literal passed to the `forBoth` method. This means that there is no way for it to verify that the input parameter of the function you pass to `forBoth` is of a type that has a `>` method defined. With the original version of `forBoth`, that was not a problem, because the type of the `cond` parameter was fixed to be `Int => Boolean`.

What can we do about this? We need to stop using the shortcut notation for function literals that leaves out the types of its parameters. Instead we have to use the most verbose notation possible, relying on explicit type annotations in the parameter list:

```
scala> val notReally = Generics.forBoth(3, 5, (x: Int) => x > 3)
val notReally: Boolean = false
```

However, this is not the only way we can make the compiler happy. Alternatively, we can explicitly provide a concrete type for the type parameter `A`, like so:

```
scala> val notReally = Generics.forBoth[Int](3, 5, _ > 3)
val notReally: Boolean = false
```

In this case, no type inference is necessary. Since we told the compiler that `A` is to be filled in with `Int`, it knows that the function literal passed to the method must be of type `Int => Boolean`.

Personally, I prefer this second version, because I like to write function literals as succinctly as possible and find the long form with explicit type annotations for input parameters rather clunky.

Apparently, Scala's type inference still has its limitations, especially if you compare it to some other languages like OCaml or Haskell.

Luckily, there is still something we can do to help the Scala compiler with type inference. A function in Scala can actually have more than one parameter list. There are a few reasons for this design decision, and you will learn a lot more about the ideas behind this in the second book.

For now, though, let's see how this seemingly weird feature can help us with our current problem. We're going to change our method signature so that the `cond` parameter is in its own parameter list, enclosed by its own pair of parentheses:

```
object Generics {
  def forBoth[A](x: A, y: A)(cond: A => Boolean): Boolean =
    cond(x) && cond(y)
}
```

Why does this make things better? The first parameter list contains two parameters of type A. At the time the Scala compiler looks at the second parameter list, containing the cond parameter of type A => Boolean, it has already inferred the concrete type of the method's type parameter A from the types of the two arguments you pass to the first parameter list. In any subsequent parameter list, this type information is already available. This means that by moving cond into a second parameter list, the compiler will know what concrete type the A in A => Boolean is without us having to explicitly specify any types.

Let's give it a try in a new REPL session:

```
scala> val notReally = Generics.forBoth(3, 5)(_ > 3)
val notReally: Boolean = false
```

When calling a method with multiple parameter lists, each of them is enclosed by its own pair of parentheses. This is exactly like the syntax for defining a method with multiple parameter lists.

More importantly, this little type inference optimization actually worked. Sweet! All that changes for client code using our method is that they need to pass in the cond function in a separate parameter list.

When designing APIs with polymorphic methods, keep the aspect of type inference in mind. It may not be the only thing to think about when you decide about the signatures of your polymorphic methods. Yet, it's something you should factor in and be aware of.

Having learned about polymorphic methods, you will have an easier time understanding the signatures of methods like map. Remember, a Seq[A] has a map method with the following signature:

```
def map[B](f: A => B): Seq[B]
```

When we call map, the concrete type that is filled in for the type parameter B can almost always be inferred from the function we pass in. For example, if we have a Seq[String], and we pass a function of type String => Int to the map method, then B is inferred to be Int. So, specifying B explicitly, as in the following example, is actually redundant:

```
scala> val words = Seq("him", "her")
val words: Seq[String] = List(him, her)

scala> val lengths = words.map[Int](_.length)
val lengths: Seq[Int] = List(3, 3)
```

Since `_.length` is a function of type `String => Int`, the explicit type annotation can be removed:

```
scala> val words = Seq("him", "her")
val words: Seq[String] = List(him, her)

scala> val lengths = words.map(_.length)
val lengths: Seq[Int] = List(3, 3)
```

Polymorphic function literals?

You may wonder if it's also possible to write function literals that are polymorphic. Consider the `forBoth` method again. The original version that only works on `Int` values can also be expressed as a function literal instead of a method:

```
val forBoth: (Int, Int, Int => Boolean) => Boolean =
  (x, y, cond) => cond(x) && cond(y)
```

Here, we define a function literal that takes two `Int` parameters and another function, and we assign it to a `val` with the identifier `forBoth`. Unfortunately, we cannot replace the `Int` with an `A` here, because there is no place to define a type parameter `A` anywhere — value identifiers cannot have type parameters.

The only way out is to create a parameterless method that returns the function:

```
def forBoth: (Int, Int, Int => Boolean) => Boolean =
  (x, y, cond) => cond(x) && cond(y)
```

Now, in a second step, we can introduce a type parameter `A` and make the function literal polymorphic by proxy, through a polymorphic, parameterless method that returns our function literal:

```
def forBoth[A]: (A, A, A => Boolean) => Boolean =
  (x, y, cond) => cond(x) && cond(y)
```

If you need type parameters in function literals, this is how you can get there.

Type erasure

It's important to understand that we only know at compile time that a `Seq[String]` is a `Seq[String]`. Due to what is known as *type erasure*, a concept you may know from Java, that information will be lost at runtime. At runtime, all type parameters of generic classes or traits will be `Any`.

Usually, that is not a problem because we can go a long way without looking at type parameters at runtime. However, there are cases where people fall into the trap of writing their code under the assumption that this type information is still available at runtime. The most prominent example of that is pattern matching using typed patterns.

To illustrate this, let's define the following singleton object in a file called `TypeErasure.scala` in the `src/main/scala` directory:

```
object TypeErasure {
  def write(xs: Seq[Any]): Unit = xs match {
    case strings: Seq[String] => println(strings.mkString(" "))
    case numbers: Seq[Int]    => println(numbers.sum.toString)
    case other                => println("Sequence has unknown type")
  }
}
```

What do you think will happen if we pass a `Seq[Int]` to the `write` method? Let's find out:

```
scala> TypeErasure.write(Seq(1, 2, 3))
1 2 3
```

Apparently, this does not match the second case, which would result in the value `"6"` being printed to the console. Instead, it already matches the first case, leading to a result of `"1 2 3"`. The reason is that at runtime there is no knowledge about `xs` being a `Seq[Int]`. The type parameters in our typed patterns are completely ignored, and the first of our patterns will always match!

The Scala compiler actually warns us about this. When you compile the `TypeErasure.scala` file, it will complain that the non-variable type argument `String` is unchecked, because it's eliminated by erasure (the same applies to the second case with the type argument `Int`).

The fact that the parameter of our `write` method has the type `Seq[Any]` should already be a warning sign. We try to avoid losing this much compile-time information about types whenever possible. We don't want to pass around values of type `Seq[Any]`. Instead, we want to use as much compile-time type information as possible.

In the example, `Seq[Any]` is also highly problematic because `write` actually expects sequences of specific types — a sequence containing only `Int` values or a sequence containing only `String` values. `Seq[Any]`, however, represents a sequence that can contain values of different types, whose common super type is `Any`. A sequence containing an `Int` value and a `String` value would be a valid instance of this type.

It would be better if our `write` method were polymorphic over a type parameter `A` and expected a `Seq[A]`. In addition, we'd have to pass in a strategy for displaying a `Seq[A]` — the `write` method itself doesn't know anything about `A`, so we need to tell it what to do. Here is one possible approach, which you can add in a new file `Typesafe.scala` next to the `TypeErasure.scala` file:

```
object Typesafe {
  type DisplayStrategy[A] = Seq[A] => String
  val displayStrings: DisplayStrategy[String] = _.mkString(" ")
  val displayInts: DisplayStrategy[Int]       = _.sum.toString
  def write[A](xs: Seq[A])(displayStrategy: DisplayStrategy[A]): Unit =
    println(displayStrategy(xs))
}
```

We define a type alias (see Section 4.7) for the display strategy, which is actually a function from `Seq[A]` to `String`. We also define two strategies, one for `String`, one for `Int`. Our `write` method expects a `Seq[A]` and a `DisplayStrategy[A]`. We use two parameter lists in order to help with type inference.

In the REPL, this will lead to the expected result:

```
scala> Typesafe.write(Seq(1, 2, 3))(Typesafe.displayInts)
6
```

Exercises

1. Implement a polymorphic version of the `timed` function we defined in Section 2.5 and which only worked for `Long` values.
2. What are the performance implications of using a function literal assigned to a `def` instead of a function literal assigned to a `val`, or a regular method like the `forBoth` we implemented in this section?
3. In the REPL, create a new sequence like this: `Seq(1, "hi")`. What is the type of the result, and why?
4. Why does the compiler complain if you accidentally pass `Seq(1, "hi")` to the `write` method of the `Typesafe` object? Try this out in the REPL, passing in one of the two display strategies we defined. Compare with what happens if you pass `Seq(1, "hi")` to the `write` method of the `TypeErasure` object.

7.3 Polymorphic extension methods

In Section 6.2 you learned how to define extension methods by means of implicit classes. However, so far, you only learned how to define such extension methods for concrete types. For example, we defined an implicit class `ExtendedInt` that added a few methods to the `Int` type, namely `seconds` and `minutes`.

We saw that we can abstract over concrete types for regular methods by making those methods polymorphic. For example, our `forBoth` method is polymorphic over a type parameter `A`. It works for any concrete type that is filled in for that type parameter.

What if we want to extend every single type with some extension method? Surely, we cannot define a separate implicit class for each type. That would not only be a lot of copy and paste, it would actually be impossible, because we don’t know all conceivable types in advance. What we need is a polymorphic implicit class — an implicit class that abstracts over a concrete type it extends, and is thus available for values of arbitrary types.

Let’s look at an example of that from the Scala standard library. Remember tuples? We discussed them briefly in Section 4.4. This is how you create a tuple of two elements:

```
scala> val entry = ("Daniel", 38)
val entry: (String, Int) = (Daniel,38)
```

However, there is an alternative syntax, provided by a polymorphic implicit class. Scala's `Predef` singleton object, the content of which is automatically imported, contains an implicit class called `ArrowAssoc`. This class looks like this:

```
implicit final class ArrowAssoc[A](private val self: A) extends AnyVal {
  def ->[B](y: B): (A, B) = (self, y)
}
```

`ArrowAssoc` is a polymorphic class, wrapping values of any type `A`. This means that the `->` method becomes available on values of any type. Moreover, the `->` method is polymorphic over a type parameter `B`. The parameter list contains one parameter of type `B`. The value of type `A` that you call `->` on will become the first element in the tuple, the value of type `B`, which is passed as an argument to the method, will become the second element of the resulting tuple.

Since this implicit class is defined in `Predef`, we don't need to import anything to make it available. Here is how we can use it as an alternative syntax for creating tuples of two elements:

```
scala> val entry = "Daniel" -> 38
val entry: (String, Int) = (Daniel,38)
```

This syntax adds somewhat less noise than the standard syntax, so you will often see it used when people create pairs. For example, the `Map` companion object contains an `apply` factory method expecting you to pass in an arbitrary number of entries from which a new map will be created. These entries are of type `(K, V)` — the two-element tuple. Here, The first element, of type `K`, denotes the key, and the second one, of type `V`, the value. Here is an example:

```
scala> val agesByName = Map("Daniel" -> 38, "Scala" -> 18)
val agesByName: scala.collection.immutable.Map[String,Int] =
  Map(Daniel -> 38, Scala -> 18)
```

Please note that `->` is used in infix operator notation here. Normally, the regular syntax of a dot, followed by a method name and the parameter list in parentheses is preferred. However, for method names consisting only of symbols, infix operator notation is usually the preferred choice. Nevertheless, it would totally be possible to write the previous example like this:

```
scala> val entry = "Daniel".->(38)
val entry: (String, Int) = (Daniel,38)
```

7.4 Summary

In this chapter, you learned about the crucial elements of Scala's type system. You were introduced to type constructors and polymorphic methods, and how to design your methods in a way that allows you to get the most out of Scala's type inference. Moreover, we looked at how to get even more out of implicit classes, allowing you to define extension methods that are made available not for a concrete type, but for arbitrary types, by making your implicit classes polymorphic.

After spending some time with Scala's type system, the next chapter will be all about values again. Specifically, we will look at different modes of evaluating expressions and how they are supported in Scala.

8. The strict brown fox jumps over the lazy dog

The question of strict versus lazy evaluation is an important one in the world of functional programming. If all expressions are evaluated lazily, this can have a positive effect on performance. After all, certain calculations that aren't actually necessary can be avoided. Another benefit is that you can create infinite sequences, since only those parts of the sequence that get requested will actually be evaluated. On the other hand, if all expressions are evaluated strictly, it's often easier to reason about what's happening at runtime in terms of memory consumption and performance.

A programming language whose very foundation is lazy evaluation is Haskell. Scala, however, defaults to strict evaluation for all expressions. This should be familiar to most developers coming from a language like Java, Ruby, or Python.

In this chapter, we want to explore the difference between strict and lazy evaluation. We'll introduce two mechanisms in Scala that allow for lazy evaluation: *by-name parameters* and *lazy vals*. These serve as building blocks from which you can build lazy data structures. This allows for things like the infinite sequences mentioned above.

8.1 By-name parameters

Usually, when you apply a function in Scala, the arguments passed to the function are evaluated strictly. Strict evaluation is also often known as *eager evaluation*. Consider the following scenario:

```
scala> def isEven(x: Int): Boolean = x % 2 == 0
def isEven(x: Int): Boolean

scala> val no = isEven(2 + 5)
val no: Boolean = false
```

What does it mean when we say that function arguments are evaluated strictly? Sticking to the example, it means that the expressions are evaluated in the following order:

1. The expression `2 + 5` is evaluated, the result is `7`.
2. `7` is passed to the `isEven` function.
3. The expression `7 % 2` is evaluated, the result is `1`.
4. The expression `1 == 0` is evaluated, the result is `false`.

Most of the times, this is perfectly fine. Sometimes, strict evaluation would be utterly wrong, though. Consider this side-effecting code:

```
def printAssessment(x: Int): Unit =
  if (x > 10) println("big") else println("small")
```

Here, if we call `printAssessment(4)`, the expressions `println("big")` and `println("small")` are evaluated lazily — and this is exactly what we would expect from an if expression. Otherwise, the whole idea of an if expression would be pointless and both branches would always be executed.

Let's have a look at what would happen in the case of if expressions being evaluated strictly. We'll assume that `printAssessment` gets called with an argument `x = 5`.

1. The expression `5 > 10` is evaluated, the result is `false`.
2. `if (false)` is evaluated
3. The side-effecting expression `println("big")` is evaluated, leading to `big` being printed to standard out.
4. The side-effecting expression `println("small")` is evaluated, leading to `small` being printed to standard out.

This is not what an if expression should do. Only the second side-effecting expression should have been evaluated, because the condition passed to the if expression evaluated to `false`.

In general, control structures always need lazy evaluation. If a language doesn't use lazy evaluation by default, then the built-in control structures need to get some special treatment.

In Scala, we have a way to make method parameters lazy. Parameters that are evaluated lazily instead of strictly are called *by-name parameters*.

How exactly do by-name parameters work? They are evaluated anew every time you access them in the respective method body. If you decide that not to access a by-name parameter at all, for example because of some branching logic, that parameter is never evaluated.

This is in contrast to *by-value parameters*, the default for method parameters in Scala. These are evaluated exactly once, before they are passed to the method.

Looping in German

Effectively, by-name parameters allow you to write methods that behave like and look similar to built-in control structures.

To demonstrate this, let's define a German version of the while loop. While you will notice that you will make use of loop constructs less and less the more familiar you get with the concepts of functional programming, this is a fun exercise helping to clarify the idea of by-name parameters.

We'll create a new sbt project called `lazy-dog` using sbt's `new` command:

```
$ sbt new dwestheide/minimal-scala-project.g8
[info] Loading global plugins from /home/daniel/.sbt/1.0/plugins
[info] Set current project to private (in build file:/home/daniel/)
[info] Set current project to private (in build file:/home/daniel/)
name [my-awesome-scala-project]: lazy-dog

Template applied in /home/daniel/./lazy-dog

$ cd lazy-dog
```

In the `src/main/scala` directory, we're going to add a file `German.scala` in which we define the `German` singleton object. In this object, we'll add a method that re-implements the logic of a while loop, giving it a German name. Here is what a first version of our `German.scala` file looks like:

```scala
import scala.annotation.tailrec

object German {
  @tailrec
  def solangeDieBedingungErfülltIst(cond: => Boolean, exec: => Unit): Unit =
    if (cond) {
      exec
      solangeDieBedingungErfülltIst(cond, exec)
    }
}
```

Our method name translates to something like "as long as the condition is fulfilled". It takes two parameters: The first one is a condition that we need to check. If it is false, the looping must come to an end. The second parameter is the code that needs to be executed as long as the condition is true.

There is some unfamiliar syntax in here: The types of the two parameters are prefixed with a `=>` symbol. This is Scala's syntax for by-name parameters.

The `cond` parameter is by-name because the idea behind a while loop is that the condition to be checked will change from evaluating to `true` to evaluating to `false` at some point. Typically, some mutable variable is involved in the condition. By making `cond` a by-name parameter, we make sure that the expression passed to `cond` is evaluated anew each time it is accessed.

The `exec` parameter is by-name because the expression passed in here needs to be executed conditionally - only if evaluating `cond` leads to a result of `true`.

In our implementation, we make use of tail recursion, which you learned about in Section 2.5. If the passed in condition is fulfilled, we evaluate the `exec` parameter and call ourselves again. In doing so, we pass `cond` and `exec` to ourselves, but since these are by-name parameters, they only get evaluated again lazily. This is crucial for our German looping method to work correctly. For example, if `cond` were evaluated strictly, we could be stuck in an endless loop.

What does it look like to use our German looping method? Let's find out by starting a new REPL session using sbt's `console` command:

```
scala> import German._
import German._

scala> var i = 1
val i: Int = 1

scala> def logAndIncrement(): Unit  = { println(i); i = i + 1 }
def logAndIncrement(): Unit

scala> solangeDieBedingungErfülltIst(i < 5, logAndIncrement())
1
2
3
4
```

We make our looping method available by importing everything defined in the `German` singleton object. Next, we define a mutable variable and a method that logs the current value of that variable and increments it by one. Finally, we can make use of our German looping method, passing in a condition based on the state of the mutable variable `i`, and an expression to be evaluated repeatedly, until the looping condition is no longer fulfilled.

While this works, it doesn't quite look like the built-in `while` loop syntax yet. However, there is something we can do about it. Let's change our looping method such that each of the two parameters is defined in its own parameter list. Remember, Scala methods can have more than one parameter list. If we apply this change, our method should look like this:

```
import scala.annotation.tailrec

object German {
  @tailrec
  def solangeDieBedingungErfülltIst(cond: => Boolean)(exec: => Unit): Unit =
    if (cond) {
      exec
      solangeDieBedingungErfülltIst(cond)(exec)
    }
}
```

What have we gained from this though? It seems that all that's changed is that in the example REPL session above, we would now have to call our looping method like this:

```
scala> solangeDieBedingungErfülltIst(i < 5)(logAndIncrement())
1
2
3
4
```

This still doesn't look exactly like the syntax of the built-in `while` loop. Here is something interesting about calling Scala methods, though. If a parameter list of a method has only one parameter, you can replace its parentheses with curly braces. Since code blocks of multiple lines are started and ended with curly braces, this allows us to not just pass in simple one-line expressions, but also entire code blocks.

Let's put a method called `count` into the `German` object that demonstrates how to put this to use:

```
def count(start: Int, end: Int): Unit = {
  var i = start
  solangeDieBedingungErfülltIst(i <= end) {
    println(i)
    i = i + 1
  }
}
```

Now using our German looping method looks quite similar to using the built-in `while` loop. As a second argument, we pass in a code block containing two lines.

Last time we made use of multiple parameter lists, it was to improve type inference for one of our methods (see Section 7.2). Here, you have seen another use case: Being able to use a code block as the last argument of a method, with syntax that is pretty much the same as for built-in control structures.

Mind you, defining our own replacement of the `while` loop is not a reasonable thing to do in real application or library code. However, it serves as a nice and small example of what you can do with the combination of by-name parameters and the curly braces syntax for the last parameter list. It's a great tool for building domain-specific languages in Scala.

In general, what we have done here is use by-name parameters for reasons of correctness. Our method wouldn't behave correctly if its parameters were by-value parameters. Another common use case for by-name parameters is to improve performance: Sometimes, there are situations where a method may or may not need the value of a specific parameter passed to it. If that value is expensive to compute, using by-name parameters means that it will only have to be computed if it's needed. Using by-value parameters, it would be computed regardless of whether it's needed, because that computation already happens before the body of the method is evaluated. There are quite a few examples of methods using by-name parameters for this very reason in the Scala standard library. You'll see an example of that in the next chapter.

By-name versus parameterless methods

By-name parameters can be a bit confusing. From the perspective of someone calling a method with by-name parameters, there is only a tiny bit of syntax telling you whether a parameter is by-name or not, whether it will be evaluated lazily or strictly. For example, `x: => String` tells you that `x` is a by-name parameter, while `x: String` tells you it's a by-value parameter. However, it's just a different syntax of defining those parameters. For the caller, there is no difference at all. This means that when you read some code calling some method, you don't know how the arguments passed to the method will be evaluated. For the compiler, `String` and `=> String` have the same type.

At the same time, when looking at the body of a method that takes by-name parameters, it can be confusing to reason about what's exactly going on. After all, accessing a parameter that is by-name looks exactly the same as accessing a parameter that is by-value. Let's look at our German looping method again:

```
import scala.annotation.tailrec

object German {
  @tailrec
  def solangeDieBedingungErfülltIst(cond: => Boolean)(exec: => Unit): Unit =
    if (cond) {
      exec
      solangeDieBedingungErfülltIst(cond)(exec)
    }
}
```

In the method body, the first time we access `cond`, it gets evaluated, and the same is true for `exec`. The second time they appear though, they are arguments to a method that takes them as by-name parameters. This means they will not be evaluated until that method's body accesses them.

Both problems can be avoided if instead of a by-name parameter, we choose to use parameterless functions. Semantically, `=> String` is exactly the same as `() => String`. However, they are two distinct types — one is a `String`, the other a function that takes zero arguments and returns a `String`.

If we use parameterless functions in our looping method, it will look like this:

```scala
import scala.annotation.tailrec

object German {
  def solangeDieBedingungErfülltIst(
      cond: () => Boolean
  )(exec: () => Unit): Unit =
    if (cond()) {
      exec()
      solangeDieBedingungErfülltIst(cond)(exec)
    }
}
```

The implementation of this method is now easier to understand: The first time, the `cond` and `exec` functions are applied — as can be observed from the parentheses following their names. When we call our looping method recursively, though, we pass the `cond` and `exec` functions as values. It's clear they are not being applied here. Not only is this easier to read, it's also safer: If we forget to apply the `cond` function in the if expression, for example, the compiler will complain, because a function returning a `Boolean` is not the same as the expected `Boolean` value.

For a user of our method, the evaluation semantics are also clearer than before. This comes at the price of code using our method not looking exactly like the built-in control structures any more. The `count` method defined in the `German` singleton object will have to be revised so that it looks like this:

```scala
def count(start: Int, end: Int): Unit = {
  var i = start
  solangeDieBedingungErfülltIst(() => i <= end) { () =>
    println(i)
    i = i + 1
  }
}
```

Now, we have to pass functions to our method, which means the evaluation semantics are more explicit and can be seen without having to look at our looping method's signature. On the downside, there is somewhat more visual noise.

Please also note that we still use curly braces for our last parameter list. The only difference is that this time, we pass in a function literal as the argument. If the body of a function literal consists of multiple lines, this is an elegant solution, because those multiple lines must be defined in a code block — and the curly braces provide the boundaries of such a code block.

Even though parameterless methods solve a lot of the problems of by-name parameters, the latter do have their place, for cases where being transparent about the evaluation semantics matters, or when minimising visual clutter matters a lot — for example, when building domain-specific languages.

That being said, there is one other limitation of by-name parameters: You can only use them in regular methods, not in function literals. For example, if you have a function literal that takes a `String` and returns a `Boolean`, its type is always `String => Boolean`. There is no way to specify that the `String` parameter should be a by-name parameter. If you need something like that, you will have to change the type of your parameter from `String` to `() => String`.

Revisiting Minitest

In Chapter 2, we started to use Minitest for writing our tests, and I promised to explain what actually happens when you write a test like this:

```
import minitest._

object StringsTest extends SimpleTestSuite {

  test("pluralise one comment") {
    assertEquals(Strings.pluralise("new comment", 1), "1 new comment")
  }

  test("pluralise two comments") {
    assertEquals(Strings.pluralise("new comment", 2), "2 new comments")
  }
}
```

The `test` method is defined in `SimpleTestSuite`, and it has two parameter lists. The first parameter list has only one parameter, the name of the test. The second parameter list contains one parameter as well, which is a by-name parameter. This means that we can pass in a code block, making the test more readable. In principle, we could use parentheses instead of curly braces for the second parameter list:

```
test("pluralise one comment")(assertEquals(3 + 5, 8))
```

However, this would only be readable for short tests like the one in this example. Often, you want a code block in your test, so using the curly braces syntax is very common when using Minitest.

Here is the signature of the `test` method:

```
def test(name: String)(f: => Void): Unit
```

Here, `f` is the by-name parameter that represents the actual test. Its return type is `Void`, which is a custom type defined by Minitest. You can think of it as semantically equivalent to `Unit`, and all expressions returning `Unit` conform to this signature.

Exercises

1. Change our German looping method so that `cond` is a by-value parameter. What is the effect on the behaviour of the method, as used in the `count` method?
2. Instead of making `cond` a by-value parameter, change the signature of the method so that `exec` is a by-value parameter. What happens now when you call `count`, which uses our looping method?
3. Use our German looping method to implement a new version of the imperative fibonacci function that you encountered in Chapter 2.
4. In Chapter 2, we defined a method called `timed` as a wrapper around functions of type `Long => Long`. Can you define a more general `timed` method that can time any expression and return its value? Use a by-name parameter and make this `timed` method polymorphic, so that it works for expressions of any type, not just expressions of type `Long`.

8.2 Lazy values

You saw that by-name parameters are evaluated repeatedly, every time they are used, while by-value parameters are only evaluated once, before they are passed to the respective method. If we move away from parameters passed to methods, we see that there are analogous evaluation semantics for expressions assigned to `def`s versus `val`s.

To demonstrate, let's define a method `repeatedlyExpensive` that takes no parameters and returns a `Long` value. The implementation is a code block in which we first print something to standard out, sleep and then return the value in question. We do this to simulate a computation that takes a lot of time — a computation we would like to avoid if it's not actually needed.

Please remember not to copy the `|` symbols. Instead, create a new line, and the REPL will display the `|` symbol until you have finished your expression:

```
scala> def repeatedlyExpensive: Long = {
     | println("hey")
     | Thread.sleep(500)
     | 1843
     | }
def repeatedlyExpensive: Long

scala> val x = repeatedlyExpensive
hey
val x: Long = 1843

scala> val y = repeatedlyExpensive
hey
val y: Long = 1843
```

Defining this method returns immediately, because we haven't called it yet. If we call this method twice, we'll see that each time, the method call takes about 500 milliseconds, and each time, our log statement appears in the REPL. That's because the code block on the right side of the method definition is evaluated anew every time. Just like parameterless functions, parameterless methods have the same evaluation semantics as by-name parameters.

In contrast, the evaluation semantics for value definitions are equivalent to by-value parameters. Let's assign the same code block to a value:

```
scala> val expensiveOnce: Long = { println("hey"); Thread.sleep(500); 1843 }
hey
val expensiveOnce: Long = 1843

scala> val z = expensiveOnce
val z: Long = 1843
```

This time, the value definition itself takes abouut 500 milliseconds, and our log statement appears immediately when the value is defined. This is because an expression assigned to a value or variable is evaluated strictly, which means, immediately. Accessing the `expensiveOnce` value shows indeed that no repeated evaluation takes place.

Call-by-need semantics

Scala provides a third way in between immediate evaluation and repeated evaluation: *call-by-need*, which means the respective expression is evaluated at most once, namely the first time the value is used. If it is never used, it will never be evaluated, whereas repeated use does not result in repeated evaluation because the value has already been cached. This kind of caching is also known as *memoisation*.

In Scala, call-by-need is supported by means of the `lazy` modifier for the `val` keyword. The following example will illustrate this:

```
scala> lazy val expensiveAtMostOnce: Long = {
     | println("hey")
     | Thread.sleep(500)
     | 1843
     | }
lazy val expensiveAtMostOnce: Long // unevaluated

scala> val x = expensiveAtMostOnce
hey
val x: Long = 1843

scala> val y = expensiveAtMostOnce
val y: Long = 1843
```

As the REPL output shows, the expression assigned to `expensiveAtMostOnce` is lazy and has not been evaluated yet. When we access the value for the first time, we see that the evaluation takes place. Our log statement is printed out, and we have to wait for about 500 milliseconds. A subsequent use of the `expensiveAtMostOnce` value does not require a new evaluation of the expression, so the second time we access it in the example, the value `1843` is returned immediately.

You can use the `lazy` modifier for any `val`, whether is is a local one defined only inside of a method or a field of a class or object.

You should be aware that `lazy` is not for free. Unlike a strictly evaluated `val` or a parameterless method defined with `def`, a `lazy val` requires synchronization, because the semantics of evaluating the right-side expression at most once must also be guaranteed if multiple threads try to use that value concurrently. Hence, you need to consider whether you really need the call-by-need semantics, or whether by-name or by-value evaluation is fine as well in a particular case.

Uniform access principle revisited

In Section 3.1, you learned about the uniform access principle. Methods and values share the same namespace, so it doesn't make a difference for a user of a class whether a field of that class is defined as a value or as a parameterless method. Now that you have learned about lazy values, you should be aware that it also doesn't matter whether a field is a strict value or a lazy value — the uniform access principle means that all three cases are the same from a consumer's point of view.

Going back to our `boardgame` sbt project from Chapter 3, you may choose to override the `toString` method not as a regular, strict `val`, but as a `lazy val`, like this:

```scala
class Position(val x: Int, val y: Int) {
  // rest of the class body omitted
  override lazy val toString: String =
    "%s [x: %d, y: %d]".format(super.toString, x, y)
}
```

This can make sense if the computation of the `String` is quite expensive, and it's unclear whether anyone will never call `toString` on instances of the respective class. The same is true for any other fields in classes you define: You can be flexible about whether to use a `val`, `lazy val`, or a `def`. Which of the three makes most sense depends a lot on your specific use case.

The other side of the coin is that, just as with by-name parameters, when looking at some code accessing a field, there is no way of knowing the evaluation strategy for that field without navigating to the source code in which the field is defined.

8.3 Summary

In this chapter you learned about three different evaluation strategies: Call-by-value, call-by-name, and call-by-need, and about Scala's built-in mechanisms for supporting these three strategies. You also got an idea of when call-by-name or call-by-need can make sense. You will see that call-by-name plays an important part in the Scala standard library. In the next chapter, you will learn about a useful generic data type that makes working with potentially absent values safe and elegant — and that happens to provide a few methods that make use of by-name parameters.

9. Know your options

In many languages, there is the concept that variable identifiers don't necessarily have to point to a valid value of the specified type. Instead, their value can be undefined, or `null`.[1] Tony Hoare, who first introduced null references in *ALGOL W*, later called this decision his "billion dollar mistake"[2].

While it's possible to use `null` in Scala, it's strongly discouraged and would take everyone who uses your code by surprise, because the language provides a much safer alternative — and apart from being safer, it's also much more pleasant to use: `Option`, a generic data type that serves as a wrapper for possibly absent values.

In this chapter, you'll learn everything there is to know about it: Why it's useful, how to work with it, what practices to avoid, and finally how you can use it to enforce invariants of a data type in a purely functional way.

9.1 The basic idea

You might wonder what's so bad about the concept of null references. Well, the big problem is that if a value identifier can point to something undefined, or `null`, you need to check whether a value is `null` or not — every single time you read a value. The reason you have to do that is because from the compiler's perspective, a value that can possibly be `null` has the same type as a value that can definitely not be `null`. For example, if you see that a value `x` is of type `String`, you don't know if it's a valid `String` value or whether `x` refers to `null`.

Programmers are human beings, and as such, we make mistakes. If we forget to check whether `x` is `null`, and call a method on it, this will lead to a runtime error, and if we don't catch the error, our whole program will crash. If you have worked with Java at all in the past, it's likely that you have come across a `NullPointerException` at some time[3]. Usually this happens because some method returns `null` when you were not expecting it to, and were thus not dealing with that possibility in your client code.

[1] In some languages, these values are also referred to as `nil` or `undefined`.
[2] https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/
[3] Other languages will throw similarly named errors in such a case.

Some languages treat `null` values in a special way or allow you to work safely with values that might be `null`. For instance, Kotlin has the null-safe `?` operator for accessing properties. In Kotlin `a?.b?.c` will not throw an exception if either `a` or its `b` property is `null`. Instead the whole expression will return `null`. Moreover, in Kotlin, values or variables that might be `null` have a different type from those that can never be `null`. This means that the compiler can give you some more confidence that you are not accidentally dereferencing a `null` value. People often decide to return `null` instead of a valid value from a method in order to represent an optional value that is absent.

Clojure, being a dynamically typed language, treats its `nil` value like an empty thing that behaves correctly depending on its context: It behaves like an empty list if accessed like a list, or like an empty map if accessed like a map, for example. This means that the `nil` value is sneaking its way up the call hierarchy. Often this is okay, but sometimes this just leads to an exception very far away from where the `nil` value was introduced because some piece of code isn't that nil-friendly after all.

In Scala, null references do exist, but that's because Scala runs on the JVM, and the language interoperability between Scala and Java. Assigning `null` to a value identifier is perfectly legal in Scala. It's also legal to pretend that a value is definitely not `null`. Just as in Java, when you are mistaken, this leads to a `NullPointerException` at runtime:

```
scala> val x: String = null
val x: String = null

scala> val length = x.length
java.lang.NullPointerException
  ... 36 elided
```

While this is legal Scala code, it's far from idiomatic. Scala tries to solve the problem by getting rid of `null` values altogether. It captures the possibility of an absent value in a container object, which means that it's also reflected in the type of a value. This is why a `NullPointerException` is a rare sight in Scala applications.

`Option[A]` is a generic data type with a type parameter `A`, and serves as a container for an optional value of type `A`. If you are familiar with Java, you'll notice a similarity to its `Optional` type[4]. `Option[A]` has two subtypes:

[4]The `Optional` type in Java was only introduced in Java 8, but you'll find that it's not quite as convenient to use as Scala's `Option` type.

- `Some[A]` — If the value of type `A` is present, the `Option[A]` is an instance of `Some[A]`, containing the present value of type `A`.
- `None` — If the value is absent, the `Option[A]` is the singleton object `None`.

By stating that a value may or may not be present *on the type level*, you and any other developers who work with your code are forced by the compiler to deal with this possibility. There is no way you may rely on the presence of a value that is optional.

`Option` is mandatory! Do not use `null` to denote that an optional value is absent.

Let's have a look at how `Option` is defined:

```
sealed abstract class Option[+A] extends IterableOnce[A]
final case class Some[+A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

There are a few new things in here: The `sealed` keyword, `case` objects, the `+` symbol in front of the type parameter, and the `Nothing` type. Let's have a short interlude to give you a quick explanation.

Case objects

A case object is similar to a case class. The compiler generates `equals` and `hashCode`, and it can participate in pattern matching. At the same time, it's a singleton object, just like other objects defined with the `object` keyword. `None` is a case object instead of a case class because, representing the absence of a value, it doesn't require any class parameters, and there is no need to create multiple instances of it.

Sealed types

The `sealed` keyword can be used for abstract classes and traits. You declare that an abstract class or trait is sealed by putting the keyword in front of the `trait` or `abstract` keyword.

But what does it mean for a type to be sealed? Essentially, it gives you control over sub typing: All the types extending a sealed type need to be defined in the same Scala source file as the sealed type. For instance, `Some` and `None` must be defined in the same file as `Option`, which happens to be `Option.scala`. This means that you can be sure that nobody is adding any other subtypes. Often, when you design a data type like `Option[A]`, you want to have full control over which subtypes exist — you define a language with a finite and well-known vocabulary.

An additional benefit of the fact that the compiler knows the exact set of subtypes of a sealed type is that it can tell you if you have covered all the cases in a pattern matching expression. You will see how to use pattern matching with `Option` values in Section 9.3.

It's possible to define your own sealed data types, which is great for modelling your domain. For example, we can define the result of a computation with integer values like this:

```scala
sealed abstract class Result
final case class Success(value: Int) extends Result
final case class Error(message: String) extends Result
```

This represents the result of an integer computation that can either succeed or go wrong in some way. We can use pattern matching to process values of type `Result`, and the compiler will tell us if we missed a case. You will learn a lot more about this technique of modelling your domain in the second book.

The `Nothing` type

While the top type in a type system is the type that is the super type of all other types, the *bottom type* is the type that is a subtype of all other types. `Nothing` is the bottom type of the Scala language. You have already seen Scala's top type, `Any`.

The reasons why `None` extends `Option[Nothing]` are quite subtle and related to the `+` symbol in front of the type parameter `A`. For now it's enough to understand that if you have an `Option[A]`, the `None` case object is a subtype, whether it's an `Option[String]`, an `Option[Int]`, or an `Option` of an arbitrary other type `A`.

The reason is that `Option` is *covariant* in its type parameter `A`, which is what the `+A` notation tells us. In a nutshell, this means that if `Apple` is a subtype of `Fruit`, then `Option[Apple]` is a subtype of `Option[Fruit]`. Since `Nothing` is a subtype of every other type, `None`, which is an `Option[Nothing]` is a subtype of any `Option[A]`.

You don't need to understand more than this about covariance at this point and can ignore the `+` symbol in front of the type parameter. If you are interested in this topic, this article[5] is a good starting point. In addition, the second book will cover

[5]https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)

the more advanced aspects of the Scala type system and explain covariance and related concepts in detail.

9.2 Creating an option

Usually, you can create an `Option[A]` for a present value by directly instantiating the `Some` case class:

```
scala> val greeting: Option[String] = Some("Hello world")
val greeting: Option[String] = Some(Hello world)
```

Or, if you know that the value is absent, you assign or return the `None` object:

```
scala> val greeting: Option[String] = None
val greeting: Option[String] = None
```

However, time and again you will need to interoperate with Java libraries or code in other JVM languages that happily make use of `null` to denote absent values. For this reason, the `Option` companion object provides a factory method that creates `None` if the given parameter is `null`, otherwise a `Some` wrapping the parameter:

```
scala> val absentGreeting: Option[String] = Option(null)
val absentGreeting: Option[String] = None

scala> val presentGreeting: Option[String] = Option("Hello!")
val presentGreeting: Option[String] = Some(Hello!)
```

In Scala code, you will hardly ever find a defensive style of programming in which you constantly perform null checks. This defensiveness can only be found at the boundary to Java libraries, where you can expect methods to return `null`. This is where the null-safe `Option` factory method is most useful.

You may also want to prefer the factory method on the `Option` companion object due to the way types get inferred otherwise. In the following example, we leave out the explicit type annotation on our value identifiers:

```
scala> val someGreeting = Some("Hello world")
val someGreeting: Some[String] = Some(Hello world)

scala> val noGreeting = None
val noGreeting: None.type = None

scala> val optGreeting = Option("Hello!")
val optGreeting: Option[String] = Some(Hello!)
```

The example shows that the `Option` factory method always returns an `Option[A]`, but when you use the `Some` case class directly, the type is inferred to be `Some[A]`. Sometimes, this can be annoying. If that is the case, the `Option` factory method can help you out.

Similarly, `noGreeting` is inferred to be of type `None.type` if we don't provide an explicit type annotation for the value identifier. Often, that's not what we want. If we want to have an `Option[String]` that is `None`, a better way to define it is this:

```
scala> val noGreeting = Option.empty[String]
val noGreeting: Option[String] = None
```

9.3 Working with optional values

Creating optional values is all pretty neat, but how do you actually work with them? It's time for an example. Let's do something boring, so we can focus on the important stuff.

Imagine you are working for one of those hipsterrific startups, and one of the first things you need to implement is a repository of users. We need to be able to find a user by their unique identifier. Sometimes, requests come in with bogus identifiers. This calls for a return type of `Option[User]` for our find method.

First, we need new sbt project called `know-your-options`, based on the usual project template:

```
$ sbt new dwestheide/minimal-scala-project.g8
[info] Loading global plugins from /home/daniel/.sbt/1.0/plugins
[info] Set current project to private (in build file:/home/daniel/)
[info] Set current project to private (in build file:/home/daniel/)
name [my-awesome-scala-project]: know-your-options

Template applied in /home/daniel/./know-your-options

$ cd know-your-options
```

Now we define the `User` case class in a file `User.scala` in the `src/main/scala` directory:

```scala
case class User(
    id: Int,
    firstName: String,
    lastName: String,
    age: Int,
    pronouns: Option[String]
)
```

A `User` has a few mandatory fields, like the `id` or their `age`. In addition to that, there is an optional field called `pronouns` where users can specify their preferred pronouns as free text, if they so choose. That's why the type of the `pronouns` field is not `String`, but `Option[String]`.

Next, we're going to add a dummy implementation of our user repository into a file `UserRepository.scala` in the same directory:

```scala
object UserRepository {
  private val users = Map(
    (1, User(1, "John", "Doe", 32, Some("he/him"))),
    (2, User(2, "Johanna", "Doe", 30, Some("she/her"))),
    (3, User(3, "Jackie", "Doe", 27, Some("they/them"))),
    (4, User(4, "Jay", "Doe", 24, None))
  )
  def findById(id: Int): Option[User] = users.get(id)
  def findAll: Seq[User]              = users.values.toSeq
}
```

In a real application, you would likely store your users in a database and allow to add new users. For our purposes, a dummy repository will do, in which all users are stored in-memory.

Our in-memory repository makes use of a data structure called `Map`, which you have first encountered in Section 7.1, when you learned about type constructors. A `Map` allows to store values associated to a unique key. In this case, we store users by their identifier. This allows us to implement a very efficient `findById` method. The `Map` companion object contains an `apply` factory method expecting an arbitrary number of tuples, each of them consisting of a key and a value.

Our `findById` method has a return type of `Option[User]` because it can be expected that people pass in identifiers for which no user exists. The implementation merely delegates to the `get` method defined on `Map`, which expects a key and returns an optional value for that key — in this example, it's an `Option[User]`, aligning perfectly with the return type of the `findById` method.

The `findAll` method returns all users, and we use the `values` method defined on the `Map` type, which returns a collection of all values stored in the map.

Now, if you receive an instance of `Option[User]` from the `UserRepository` and need to do something with it, how do you do that?

One way would be to check if a value is present by means of the `isDefined` method of your option, and, if that is the case, get that value via its `get` method:

```
scala> val user = UserRepository.findById(1)
val user: Option[User] = Some(User(1,John,Doe,32,Some(he/him)))

scala> if (user.isDefined) println(user.get.firstName)
John
```

If you think this is clunky and expect something more elegant from Scala, you're on the right track. More importantly, if you use `get`, you might forget about checking with `isDefined` before. This will lead to an exception at runtime, so you haven't gained anything over using `null`. You should stay away from this way of accessing options whenever possible. A lot of people think that the `get` method should never have been defined on `Option` in the first place.

There are a couple of safe ways of working with `Option` values, and we'll go through all of them now.

Providing a default value

One of the most straightforward ways of quickly escaping the `Option` wrapper is to access the value and fall back to a default value, in case the `Option` is `None`. This use case is covered by the `getOrElse` method:

```
scala> val user = User(4, "Jay", "Doe", 24, None)
val user: User = User(4,Jay,Doe,24,None)

scala> val pronouns = user.pronouns.getOrElse("not specified")
val pronouns: String = not specified
```

One detail to be aware of is that the default value expected by the `getOrElse` method is a by-name parameter, which means that it is only evaluated if the option on which you invoke `getOrElse` is indeed `None`. The reason is that the expression you pass to `getOrElse` as the default value might be costly to evaluate. See Section 8.1 for more details on this topic.

Pattern matching

`Option` is a sealed data type, and the two possible subtypes are a case class and a case object. As such, it lends itself perfectly to being processed using pattern matching. Let's add a file `UserRenderer.scala` to the `src/main/scala` directory and define a `UserRenderer` singleton object in it. This object will contain a few methods, the first one expecting a user and returning a label that can be used to represent the user in a user interface. Here is the skeleton, to be implemented in the next step:

```scala
object UserRenderer {
  def userLabel(user: User): String = ""
}
```

Before using pattern matching to implement this method, let's specify its expected behaviour in a test. We're going to add a file `UserRendererTest.scala` to the `src/test/scala` directory, containing two test cases:

```scala
import minitest._

object UserRendererTest extends SimpleTestSuite {
  test("Label for user with pronouns contains first name and pronouns") {
    val user = User(1, "Jimmy", "Doe", 25, Some("they/them"))
    assertEquals(UserRenderer.userLabel(user), "Jimmy (they/them)")
  }
  test("Label for user without pronouns contains first and last name") {
    val user = User(1, "Jane", "Doe", 31, None)
    assertEquals(UserRenderer.userLabel(user), "Jane Doe")
  }
}
```

If you run these tests using the `test` command in the interactive sbt shell, they should fail, for now. We're going to make them succeed by implementing the `userLabel` method with a pattern matching expression:

```scala
object UserRenderer {
  def userLabel(user: User): String = user.pronouns match {
    case Some(pronouns) => s"${user.firstName} ($pronouns)"
    case None           => s"${user.firstName} ${user.lastName}"
  }
}
```

An `Option` can only ever be either a `Some`, containing a value, or `None`, so these are the two cases we need to consider. In each case, we use string interpolation in order to assemble the resulting `String`.

Sometimes, pattern matching on an `Option` value can be rather verbose. So, even if you are all excited about pattern matching, it's a good idea to get comfortable with the alternatives you are about to learn.

Folding

Let's say our application includes a discussion board, and we want to show the user's first name if they exist, or `N/A` if they do not exist. Here is a simple function doing that with a pattern matching expression, which you can add to the `UserRenderer` object:

```
def firstName(optUser: Option[User]): String = optUser match {
  case Some(user) => user.firstName
  case None       => "N/A"
}
```

We should also add some test cases to our `UserRendererTest` that describe the expected behaviour:

```
test("firstName is returned if optUser is defined") {
  val optUser = Option(User(1, "Jane", "Doe", 31, None))
  assertEquals(UserRenderer.firstName(optUser), "Jane")
}
test("firstName returns N/A if optUser is empty") {
  assertEquals(UserRenderer.firstName(None), "N/A")
}
```

Executing the `test` command from the interactive sbt shell should show that all tests were successful.

A more succinct alternative to the pattern matching expression above is the `fold` method defined on `Option`. Its signature is as follows:

```
def fold[B](ifEmpty: => B)(f: A => B): B
```

This method is polymorphic, taking a type parameter `B`, and it has two parameter lists. The first one has a by-name parameter that will be evaluated and returned if the `Option` is empty, or, in other words, `None`. If it is defined, though, the function `A => B` will be applied to the value contained in the `Some`. Here, `A` is the type parameter of our `Option[A]`. In either case, a value of type `B` will be returned.

The reason that `ifEmpty` is a by-name parameter is the same as for the `getOrElse` method. The value to use if the `Option` is empty may be expensive to compute. When calling `fold`, there's no need to care about that.

Essentially, `fold` allows you to do the same thing we have been doing with our pattern matching expression in the `firstName` method. Let's see what it looks like if we replace that expression with a call of the `fold` method:

```scala
def firstName(optUser: Option[User]): String =
  optUser.fold("N/A")(_.firstName)
```

First, we pass in the empty value, `"N/A"`. Then, in the second parameter list, we provide a function literal from `User` to `String` that uses the placeholder syntax to access the `User` argument. This is certainly shorter than the previous version. You may still find pattern matching on the `Option` more readable, especially if you haven't internalised the concept of functions as values yet. However, personally I prefer to `fold` options instead of pattern matching on them.

Exercises

1. Verify that the version the `firstName` method that uses `fold` has the same behaviour as the previous implementation based on pattern matching.
2. Can you think of a good reason why `fold` uses two parameter lists instead of one?
3. Rewrite the `userLabel` method in terms of `fold`, replacing the existing pattern matching expression.

9.4 Options are collections

I already mentioned that `Option[A]` is a container for a value of type `A`. More precisely, it's a type of collection with an interesting constraint: Instances of this collection type contain either zero elements or exactly one element of type `A`. This is a powerful idea!

When we looked at how `Option` is defined, you saw that `Option[A]` extends `IterableOnce[A]`. The latter is a trait that is high up in the type hierarchy of the Scala collection library. When we looked at how to process sequences in a functional way in Section 4.5, we saw that `Seq[A]` extends `IterableOnce[A]` as well. Basically, `IterableOnce[A]` denotes something that can iterate at least once over elements of type `A`.

To reinforce the idea that `Option` is merely a special kind of collection with either one or zero elements, see what happens when you convert an `Option` value to a `Seq` value, by means of the `toSeq` method:

```
scala> val someInt: Option[Int] = Some(23)
val someInt: Option[Int] = Some(23)

scala> val noInt: Option[Int] = None
val noInt: Option[Int] = None

scala> val seq = someInt.toSeq
val seq: Seq[Int] = List(23)

scala> val emptySeq = noInt.toSeq
val emptySeq: Seq[Int] = List()
```

As you can see, an `Option` that is defined gets converted to a sequence with one element, while an undefined `Option` gets converted to an empty sequence.

Let's look at some of the things that we can do with `Option` values now.

Performing a side-effect if a value is present

If you need to perform some side-effect only if an optional value is present, the `foreach` method you know from sequences comes in handy:

```
scala> UserRepository.findById(2).foreach(user => println(user.firstName))
Johanna
```

The function passed to `foreach` will be called exactly once, if the `Option` is a `Some`, or never, if it is `None`.

Mapping an option

The good thing about options behaving like other collections is that you can work with them in a very functional way. This means you can draw on what you have already learned about processing sequences.

Just as you can map a `Seq[A]` to a `Seq[B]`, you can map an `Option[A]` to an `Option[B]`. If your instance of `Option[A]` is defined, meaning it is `Some[A]`, the result is `Some[B]`, otherwise it is `None`.

If you compare `Option` to `Seq`, `None` is the equivalent of an empty sequence: when you map an empty `Seq[A]`, you get an empty `Seq[B]`, and when you map an `Option[A]` that is `None`, you get an `Option[B]` that is `None`.

To illustrate this, here is how we can get the age of an optional user:

```
scala> val age = UserRepository.findById(1).map(_.age)
val age: Option[Int] = Some(32)

scala> val age2 = UserRepository.findById(23).map(_.age)
val age2: Option[Int] = None
```

Here we are mapping the `Option[User]` we get from the repository to an `Option[Int]` of that user's age. In the first case, the user exists, so our resulting `Option[Int]` is defined. In the second case, the repository returns an empty `Option[User]`, so the resulting `Option[Int]` is empty as well.

Flat mapping an option

Let's do the same for the pronouns:

```
scala> val pronouns = UserRepository.findById(1).map(_.pronouns)
val pronouns: Option[Option[String]] = Some(Some(he/him))
```

The type of the resulting `pronouns` is `Option[Option[String]]`. Why is that?

Think of it like this: You have an `Option` container for a `User`, and *inside* that container you are mapping the `User` instance to an `Option[String]`, since that is the type of the `pronouns` field on our `User` class.

These nested options are a nuisance? No problem, `Option` provides a `flatMap` method as well. Just like you can flatten a `Seq[Seq[A]]`, you can do the same for an `Option[Option[A]]`. Here's an example:

```
scala> val pronouns1 = UserRepository.findById(1).flatMap(_.pronouns)
val pronouns1: Option[String] = Some(he/him)

scala> val pronouns2 = UserRepository.findById(4).flatMap(_.pronouns)
val pronouns2: Option[String] = None

scala> val pronouns3 = UserRepository.findById(42).flatMap(_.pronouns)
val pronouns3: Option[String] = None
```

The result type is now `Option[String]`. If the user *and* its pronouns are defined, we get the latter as a flattened `Some`. If either the user or its pronouns are empty, we get a `None`.

Maybe you remember that for `Seq[A]`, the signature of `flatMap` looks like this:

```
def flatMap[B](f: A => IterableOnce[B]): Seq[B]
```

You can pass in a function returning a collection of any subtype of `IterableOnce[B]`, and the result of calling `flatMap` will be coerced to `Seq[B]`.

The signature of `flatMap`, as defined for `Option[A]`, is a bit stricter. It requires you to pass in a function that returns an `Option[B]`. It wouldn't make sense to pass in a collection that might contain multiple elements. How would `flatMap` be able to return an `Option[B]`, which can contain at most one element? So here is what `flatMap` looks like:

```
def flatMap[B](f: A => Option[B]): Option[B]
```

To understand how this works, let's have another look at what happens when flat mapping a sequence of sequence of strings, always keeping in mind that an `Option` is a collection, too:

```
scala> val names = Seq(Seq("John", "Ada"), Seq(), Seq("Ida", "Paul"))
val names: Seq[Seq[String]] = List(List(John, Ada), List(), List(Ida, Paul))

scala> val nestedUppercase = names.map(_.map(_.toUpperCase))
val nestedUppercase: Seq[Seq[String]] =
  List(List(JOHN, ADA), List(), List(IDA, PAUL))

scala> val uppercase = names.flatMap(_.map(_.toUpperCase))
val uppercase: Seq[String] = List(JOHN, ADA, IDA, PAUL)
```

If we use `flatMap`, the mapped elements of the inner sequences are converted into a single flat sequence of strings. As a result, nothing will remain of any empty inner sequence.

To lead us back to the `Option` type, consider what happens if you map or flat map a sequence of options of strings.

```
scala> val names = Seq(Some("Ada"), None, Some("Daniel"))
val names: Seq[Option[String]] = List(Some(Ada), None, Some(Daniel))

scala> val nestedUppercase = names.map(_.map(_.toUpperCase))
val nestedUppercase: Seq[Option[String]] = List(Some(ADA), None, Some(DANIEL))

scala> val uppercase = names.flatMap(_.map(_.toUpperCase))
val uppercase: Seq[String] = List(ADA, DANIEL)
```

If you just map over the sequence of options, the result type remains `Seq[Option[String]]`. Using `flatMap`, all elements of the inner collections[6] are put into a flat sequence: The one element of any `Some[String]` in the original sequence is unwrapped and put into the result sequence, whereas any `None` value in the original sequence does not contain any element to be unwrapped. Hence, `None` values are effectively filtered out, like empty sequences in the example above.

This works because the signature of the function expected by the `flatMap` method on `Seq[A]` is not `A => Seq[B]`, but rather `A => IterableOnce[B]`. As we saw, both `Seq[A]` and `Option[A]` are subtypes of `IterableOnce[A]`.

With this in mind, have a look again at what `flatMap` does on the `Option` type.

Filtering an option

You can filter an option just like you can filter a sequence. If the instance of `Option[A]` is defined, i.e. it is a `Some[A]`, *and* the predicate passed to `filter` returns `true` for the wrapped value of type `A`, the `Some` instance is returned. If the `Option` instance is already `None` or the predicate returns `false` for the value inside the `Some`, the result is `None`. Let's try this out in the REPL:

[6]Remember, `Option[A]` is a collection type, a subtype of `IterableOnce[A]`.

```
scala> val optUser1 = UserRepository.findById(1).filter(_.age > 30)
val optUser1: Option[User] = Some(User(1,John,Doe,32,Some(male)))

scala> val optUser2 = UserRepository.findById(2).filter(_.age > 30)
val optUser2: Option[User] = None

scala> val optUser3 = UserRepository.findById(3).filter(_.age > 30)
val optUser3: Option[User] = None
```

Here, `optUser1` is defined because the user exists and their age is greater than 30. However, `optUser2` is `None` because even though the user exists, their age is not greater than 30, and `optUser3` is `None` because the user does not even exist — the filtering doesn't have any effect, as it happens on an `Option[User]` that's already `None`.

Being able to use operations like `map`, `flatMap`, or `filter` on options means that there is often no need to escape the option container early. Instead, we can choose to stay inside the container as long as possible, transforming it until we finally need to escape the container using `getOrElse`, `fold`, or a pattern matching expression.

Exercises

1. Add a file `Options.scala` to the `src/main/scala` directory of the `know-your-options` sbt project, containing a singleton object `Options`. Add a function with the following signature to it:
   `def map[A, B](opt: Option[A])(f: A => B): Option[B]`. Implement it without using the existing `map` method defined on `Option`. Instead, only use pattern matching. Add a file `OptionsTest.scala` in `src/test/scala` and add a few test cases describing the expected behaviour of `map`. As before in this book, use the `minitest` testing library to implement your tests.
2. Do the same for the following method:
   `def flatMap[A, B](opt: Option[A])(f: A => Option[B]): Option[B]`.
3. Repeat the exercise for a `filter` method with this signature:
   `def filter[A](opt: Option[A])(f: A => Boolean): Option[A]`.
4. When you learned about sequences, you learned about two complementary methods: `exists` and `forall`. Since `Option` is a collection type, it has these methods as well, with the same signature. Use the REPL to find out what the behaviour of these methods is for options, depending on whether the option is defined or empty.

9.5 For comprehensions

Now that you know that an `Option` is a collection and provides `map`, `flatMap`, `filter`, and other methods you know from sequences, you will probably already suspect that options can be used in *for comprehensions*. Often, this is the most readable way of working with options, especially if you have to chain a lot of `map`, `flatMap` and `filter` invocations.

Here is a function you can add to the `UserRenderer` singleton object, that gets the pronouns of a user, given its user id, using a for comprehension:

```scala
def pronounsForId(userId: Int): Option[String] =
  for {
    user     <- UserRepository.findById(userId)
    pronouns <- user.pronouns
  } yield pronouns
```

This is equivalent to nested invocations of `flatMap` and `map` (see Section 4.6). Here is what the Scala compiler translates the previous definition of `pronounsForId` to:

```scala
def pronounsForId(userId: Int): Option[String] =
  UserRepository
    .findById(userId)
    .flatMap(user => user.pronouns.map(pronouns => pronouns))
```

The call to `map` on the pronouns is redundant in our example and could be removed. However, the Scala compiler doesn't know that. The `yield` expression can be more complex than in our example. Thus, it always gets translated to a `map` call where the yielded expression becomes the right side of the function literal passed to `map`.

If the `UserRepository` already returns `None` or the `pronouns` field of the user is `None`, the result of the for comprehension is `None`.

For the user id in the following example, the `pronouns` field is defined, so it is returned in a `Some` wrapper:

```
scala> val pronouns = UserRenderer.pronounsForId(3)
val pronouns: Option[String] = Some(they/them)
```

If we want to retrieve the pronouns of all users that have specified it, we can iterate over all users, and for each of them yield their `pronouns` field, if it is defined. This is what the following method does, which you can add to the `UserRenderer` object:

```
def pronouns: Seq[String] =
  for {
    user     <- UserRepository.findAll
    pronouns <- user.pronouns
  } yield pronouns
```

Since we are effectively flat mapping, the result type is `Seq[String]`, and the resulting sequence contains three elements, because `pronouns` is only defined for the first three users:

```
scala> val pronouns = UserRenderer.pronouns
val pronouns: Seq[String] = List(he/him, she/her, they/them)
```

Exercises

1. Rewrite `pronounsForId` in terms of `map` and `flatMap` to get a better understanding of what the for comprehension we used gets translated to by the compiler. See Section 4.6 for a refresher on how these get desugared by the compiler. Before you rewrite the function, add some test cases to the `UnderRendererTest` that describe the expected behaviour. This way, you can make sure that you don't change the behaviour of the function when rewriting its implementation.
2. Repeat this exercise for the `pronouns` method.
3. When you learned about for comprehensions, you also learned about guards. Add a guard to `pronounsForId` that filters out pronouns that are blank. The method `isBlank` defined on `String` will come in handy for that. Add a new user to the `UserRepository` with a blank `pronouns` field in order to verify this works as expected.

9.6 Chaining options

Options can also be chained. A good example of how this can be useful is finding a resource, when you have several different locations to search for it and an order of preference. In our case, we prefer the resource to be found in the config directory, and we would like to fall back to the resource from the classpath.

Let's add a file `Resource.scala` to the `src/main/scala` directory of our `know-your-options` sbt project. It will contain a case class `Resource` as well as the companion object for that class:

```scala
final case class Resource(content: String)
object Resource {
  def resourceFromConfigDirectory: Option[Resource] = None
  def resourceFromClasspath: Option[Resource] = Some(
    Resource("classpath resource")
  )
  def resource: Option[Resource] = resourceFromConfigDirectory match {
    case Some(res) => resourceFromConfigDirectory
    case None      => resourceFromClasspath
  }
}
```

In the companion object, we define two parameterless methods, one returning the resource from the config dir, the other from the classpath. Both have a return type of `Option[Resource]`. The logic of preferring the resource from the config dir over the one from the classpath is implemented in a method called `resource`, using a pattern matching expression.

This pattern matching expression can be abstracted, though. Most of the code in our `resource` method would be the same if we were dealing with any other option values and types. This is why `Option[A]` has a chaining method called `orElse`, which implements this very logic. It expects another option as a by-name parameter, which serves as the fallback option. It's a by-name parameter because it might be expensive to compute this value. If the option you call `orElse` on is `None`, `orElse` will return the option passed to it, otherwise it returns the one which it was called on.

We can rewrite our `resource` method using `orElse`, and it will contain far less boilerplate code than before:

```scala
def resource: Option[Resource] =
    resourceFromConfigDirectory.orElse(resourceFromClasspath)
```

9.7 Purely functional data types

Do you remember our `SongTitle` data type, which we defined to understand the concept of value classes in Section 6.1? We made up some invariants that must hold true for this data type. In real applications, those invariants would likely be provided by domain experts or based on your careful research. The point is that many data types do have invariants that they must enforce. In our `SongTitle` value class, we used Scala's built-in `require` method to do that. In addition, we made the constructor private, so that `SongTitle` values can only be created through the factory method defined in the companion object:

```scala
class SongTitle private(val value: String) extends AnyVal
object SongTitle {
  def apply(title: String): SongTitle = {
    require(title.nonEmpty, "title must not be empty")
    require(title.size < 255, "title must be shorter than 255 characters")
    new SongTitle(title)
  }
}
```

Here is a short reminder of what `require` does: If the `Boolean` value passed to it is `false`, it will throw an `IllegalArgumentException`. I already mentioned that throwing exceptions isn't a common thing to do in Scala. Expressions that throw or catch exceptions don't play well with functional programming because they break referential transparency. You can't substitute an exception-throwing expression with its result any more, because your program won't behave the same — in one case, the thrown exception might be caught, in the other, it bubbles up uncaught and your program crashes. This is the very reason that people prefer other strategies of error handling in functional programming.

We won't look at how throwing and catching exceptions works in Scala in this book. Instead, let's think about how we can enforce the invariants of the `SongTitle` class in a way that doesn't break the assumptions of purely functional programming — in a way that is referentially transparent.

Well, instead of returning a value of type `SongTitle`, the `apply` factory method can return a value of type `Option[SongTitle]`. Then, if the invariants of this data type are not met by the arguments passed in to the factory method, it will no longer have to throw an exception. Instead, it can return a `Some[SongTitle]` if everything is okay, or a `None` if any of its invariants is violated by the input values. Let's add a file `SongTitle.scala` to the `src/main/scala` directory of the `know-your-options` sbt project, in which we'll define a purely functional version of the `SongTitle` data type:

```
class SongTitle private (val value: String) extends AnyVal
object SongTitle {
  def apply(title: String): Option[SongTitle] = title match {
    case ""                        => None
    case s if s.size >= 255 => None
    case _                         => Some(new SongTitle(title))
  }
}
```

Now, our `SongTitle` data type enforces its invariants in a purely functional way. This means it's easier to reason about code using this data type, and its factory method is composable with other pure functions.

One of the downsides is that since our value class is now wrapped in an option, which is a generic type, an instance of `SongTitle` will actually have to be created at runtime. Previously, the `SongTitle` instance would have been represented merely as a `String` value, which is the value it wraps.

We have lost the ability to provide meaningful error messages to the caller of our factory method. Returning an `Option[SongTitle]` is a safer alternative to the strategy of returning `null` if invariants are violated, which you sometimes find in languages like Java or Ruby. For now, we have to accept this downside. In the second book, you will learn about a few other data types for error handling. These are a bit more complex than `Option`, but they allow you to pass along errors to the caller as well.

Finally, you might be unsure about what to do with the `Option[SongTitle]` returned by the factory method. You might feel the urge to get the `SongTitle` value out of the `Option` container as soon as possible, so that you can create a valid `Track`. Well, we can't do that. Since there is probably no good default value for a `SongTitle`, there is no point in calling `getOrElse` on the `Option[SongTitle]`. Instead, we need to stay in the `Option` context as long as possible. If we want to create a track, it will be an `Option[Track]`. Probably, we're going to present an error message to the user if the `Option[Track]` ends up being empty.

9.8 Option in the standard library

The Scala standard library makes use of `Option` in a few places in order to provide a safer alternative to throwing exceptions. In this section, we'll look at some examples.

Parsing strings

Having to parse a `String` value and convert it into a value of a different type is a common need. For the built-in value types, Scala provides extension methods on `String`. These extension methods are always in scope. Being defined in the `Predef` object, they are imported automatically.

Up until Scala 2.12, only unsafe versions of these methods existed. `toInt`, `toLong`, `toBoolean`, and the other methods throw an exception if the `String` value cannot be parsed to the respective type. Here is an example:

```
scala> val x = "not a number".toInt
java.lang.NumberFormatException: For input string: "not a number"
...
```

A safer alternative is to use methods like `toIntOption`, `toLongOption`, or `toBooleanOption`. These return an `Option[Int]`, `Option[Long]`, and `Option[Boolean]`, respectively. If the `String` value cannot be parsed, the return value is `None`. Here are a few examples:

```
scala> val noBool = "no boolean".toBooleanOption
val noBool: Option[Boolean] = None

scala> val yes = "true".toBooleanOption
val yes: Option[Boolean] = Some(true)

scala> val no = "false".toBooleanOption
val no: Option[Boolean] = Some(false)
```

Since parse errors are expected, we shouldn't treat them as exceptions in our code. Make sure to use these methods for converting from `String` to built-in types, and design your own methods for parsing `String` to custom types in the same way.

Safer sequence processing

In the `boardgame` project in Chapter 3, we defined a class `Board`. This class used a method called `head` on the sequences passed to it. This was okay because we made sure the sequences were not empty while creating an instance of the class. In general, the `head` method should be avoided, though. Just like the unsafe `String` conversion methods you saw before, it throws an exception if its preconditions are not met — in this case, if the sequence is empty:

```
scala> val headlessChicken = Seq.empty[String]
val headlessChicken: Seq[String] = List()

scala> val head = headlessChicken.head
java.util.NoSuchElementException: head of empty list
  at scala.collection.immutable.Nil$.head(List.scala:599)
  at scala.collection.immutable.Nil$.head(List.scala:598)
  ... 36 elided
```

This is something we want to avoid. As a safe alternative to the `head` method, you should use `headOption` instead, which returns an `Option[A]`. If the sequence is empty, the returned value is `None`. Otherwise, it's the first element of the sequence in a `Some` wrapper:

```
scala> val headlessChicken = Seq.empty[String]
val headlessChicken: Seq[String] = List()

scala> val head = headlessChicken.headOption
val head: Option[String] = None
```

Other methods for sequence processing follow the same approach. An example of that is `find`, which allows you to look for the first element in the sequence for which a given predicate returns `true`. Here is its signature:

```
def find(p: A => Boolean): Option[A]
```

The returned `Option[A]` is `None` if the sequence is empty or the passed in predicate doesn't return `true` for any of its elements.

9.9 Summary

In this chapter you learned how to avoid the billion dollar mistake and how to work with possibly missing values in a safe and elegant way. You saw how to process `Option` values and learned that it's best not to try to escape the `Option` container too early. We looked at `Option` as a special kind of collection that can contain either one element or none and saw how to use for comprehensions for working with `Option` values. Finally, we looked at a purely functional way of enforcing a data type's invariants, by returning an `Option` instead of throwing exceptions.

You might still not be sure how to put everything you have already learned together. In the next chapter, we’re going to create a small example application that demonstrates how all the concepts we have covered so far come together.

10. Putting it all together

Throughout this book, you have come across a lot of concepts that are probably new to you. In addition, there was a lot of new syntax to get familiar with. Still, learning about individual concepts and how they are implemented in a language is not enough. This is why we're going to develop a small example application in this chapter. It will serve as a case study that demonstrates how the things you have learned in this book come together, and how they can be applied in practice.

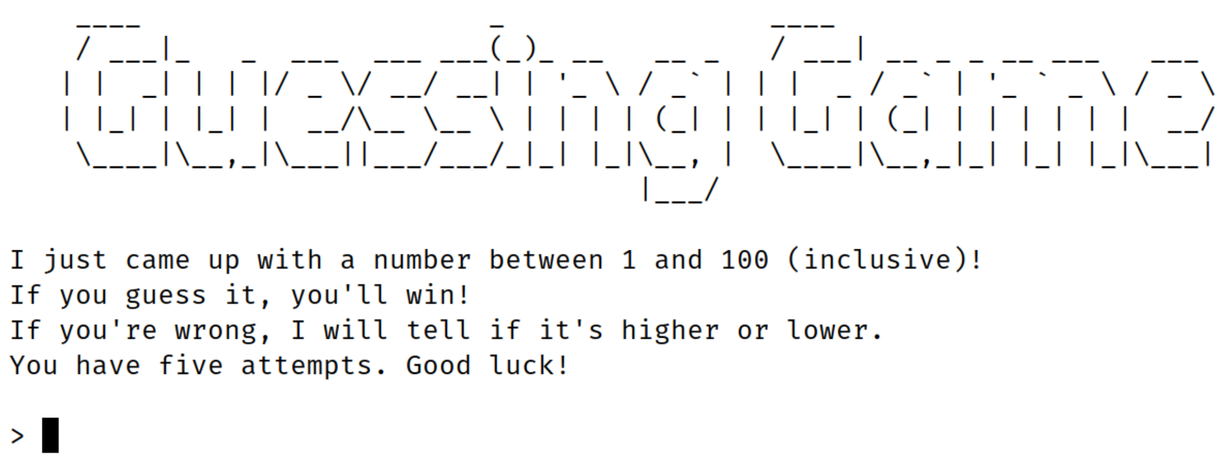
Our example will be a number guessing game. This is heavily inspired[1] by the Rust guessing game tutorial[2] that starts the freely available book The Rust Programming Language[3] by Steve Klabnik, Carol Nichols, and other people — one of the best learning resources available for Rust or any programming language.

10.1 Talking to the product owner

After talking to the product owner of the guessing game, we know more about the requirements. After doing a focus group study, the product owner and their team of usability engineers found out that what people are really looking for is a guessing game with a command-line interface.

One of the most important features the members of the focus group asked for is a startup banner. The user interface of the guessing game should look like the mockup depicted in Figure 10.1.

[1] A euphemism for *shamelessly stolen*.
[2] https://doc.rust-lang.org/book/ch02-00-guessing-game-tutorial.html
[3] https://doc.rust-lang.org/book/title-page.html



```
  ____                     _               ____
 / ___|_   _  ___  ___ ___(_)_ __   __ _  / ___| __ _ _ __ ___   ___
| |  _| | | |/ _ \/ __/ __| | '_ \ / _` | | |  _ / _` | '_ ` _ \ / _ \
| |_| | |_| |  __/\__ \__ \ | | | | (_| | | |_| | (_| | | | | | |  __/
 \____|\__,_|\___||___/___/_|_| |_|\__, |  \____|\__,_|_| |_| |_|\___|
                                   |___/

I just came up with a number between 1 and 100 (inclusive)!
If you guess it, you'll win!
If you're wrong, I will tell if it's higher or lower.
You have five attempts. Good luck!

> 
```

Figure 10.1: The user interface of the guessing game

Apart from these requirements, the product owner came up with the following specification:

- The game should create a random target number between 1 and 100
- The player should be allowed up to five guesses — if they haven't guessed the target number by then, they have lost and should see a game over banner
- Whenever a player has guessed wrong, and they still have at least one guess remaining, they should be presented with a hint, telling them whether the target number is lower or higher than the last guess
- When a player has guessed correctly, they should be shown a success message
- When a player enters something other than a valid number, they should be notified about their mistake, but this should not count as a guess; the number of remaining guesses should not be decreased

When it comes to non-functional requirements, we learn that *correctness* is the most important quality of the software.

10.2 Architectural decisions

In order to ensure that our guessing game is correct, we'd like to design our program in such a way that it's easy to test the domain logic with unit tests. This is why we decide to make use of an architectural style called Functional core, imperative shell[4].

What does this mean? We try to implement as much of the logic of our game in a purely functional core. That core has no dependencies to other parts of the program. Around the core, there is an imperative shell, which contains all the parts of the program that are not pure — functions that perform effects like reading from the standard input, printing to the standard output, or generating random numbers. We try to keep this imperative shell almost free of logic, which means we avoid conditional branching as much as possible.

10.3 Setting up the project

First, we create a new sbt project called `guessing-game`, using the `new` command:

```
$ sbt new dwestheide/minimal-scala-project.g8
[info] Loading global plugins from /home/daniel/.sbt/1.0/plugins
[info] Set current project to private (in build file:/home/daniel/)
[info] Set current project to private (in build file:/home/daniel/)
name [my-awesome-scala-project]: guessing-game

Template applied in /home/daniel/./guessing-game

$ cd guessing-game
```

In the newly created `guessing-game` directory, we create a few directories reflecting the package structure of our program:

```
.
├── src
│   ├── main
│   │   └── scala
│   │       └── guessinggame
│   │           ├── core
│   │           │   ├── domain
│   │           │   └── presentation
│   │           └── shell
│   └── test
```

[4] https://www.destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell

```
│       └── scala
│           └── guessinggame
│               └── core
│                   └── domain
```

If you’re on Linux or Mac OS X, you can use the `mkdir` command to create these directories:

```
$ mkdir -p src/main/scala/guessinggame/shell/
```

We have a top-level package called `guessinggame`, and nested packages called `core` and `shell` for the functional core and imperative shell, respectively. Inside the `core` package, there will be two subpackages — one for the domain model and one for the purely functional part of the presentation layer.

10.4 The functional core

Since the imperative shell depends on the functional core, and not the other way around, it’s best to start thinking about what the latter should look like.

The domain model

First of all, we need to represent the concept of a number that can be guessed — a *guessable number*. Another concept of our domain is the *game* in which we keep track of the target number to be guessed as well as the remaining guesses. Finally, a relevant concept of our domain is probably the *result* of a player’s guess — is their guess correct or wrong, or have they exhausted their available attempts at guessing the target number?

The guessable number

Let’s start with the concept of a guessable number. In the language of *domain-driven design*, this is a *value object* — an immutable value without an identity or lifecycle. A guessable number has some invariants it needs to enforce: Its minimum value is `1`, its maximum value is `100`. As such, it lends itself to be implemented as a Scala value class (see Section 6.1) with a private constructor and factory methods that enforce the invariants in a purely functional way, as described in Section 9.7.

In `src/main/scala`, you should have a subdirectory `guessinggame/core/domain`. The `GuessableNumber.scala` file should be located in that directory. Here is a first shot:

```
package guessinggame.core.domain

final class GuessableNumber private (val value: Int) extends AnyVal {
  def >(other: GuessableNumber): Boolean = value > other.value
  def <(other: GuessableNumber): Boolean = value < other.value
}

object GuessableNumber {
  def fromInt(n: Int): Option[GuessableNumber] = None
  def fromString(s: String): Option[GuessableNumber] = None
}
```

We define two methods > and < that merely delegate to the corresponding methods defined on `Int`. This makes it more convenient to use values of type `GuessableNumber` in comparisons.

The companion object has two factory methods: `fromInt` creates a `GuessableNumber` from an `Int`, enforcing the mentioned invariants. In addition, `fromString` creates a `GuessableNumber` from a `String` representation.

Before implementing the two factory methods properly, we should put our expectations into an automated test. In the `src/test/scala` directory, there should be a `guessinggame/core/domain` subdirectory as well. Let's add a file `GuessableNumberTest.scala` to it, with the following test cases:

```
package guessinggame.core.domain

import minitest.SimpleTestSuite

object GuessableNumberTest extends SimpleTestSuite {
  test("Can't create GuessableNumber from -1") {
    assertEquals(GuessableNumber.fromInt(-1), None)
  }
  test("Can't create GuessableNumber from 0") {
    assertEquals(GuessableNumber.fromInt(0), None)
  }
  test("Can create GuessableNumber from 1") {
    val received = GuessableNumber.fromInt(1).map(_.value)
    assertEquals(received, Some(1))
  }
  test("Can create GuessableNumber from 99") {
    val received = GuessableNumber.fromInt(99).map(_.value)
    assertEquals(received, Some(99))
  }
  test("Can create GuessableNumber from 100") {
    val received = GuessableNumber.fromInt(100).map(_.value)
    assertEquals(received, Some(100))
  }
  test("Can't create GuessableNumber from 101") {
    assertEquals(GuessableNumber.fromInt(101), None)
  }
  test("Can't create GuessableNumber from non-number string") {
    assertEquals(GuessableNumber.fromString(""), None)
  }
  test("Can create GuessableNumber from valid number string") {
    val received = GuessableNumber.fromString("42").map(_.value)
    assertEquals(received, Some(42))
  }
}
```

If you run the `test` command in the interactive sbt shell, at least some of the tests should fail, because `fromInt` and `fromString` always return `None` so far.

We start with `fromInt`. We want to return `None` if the passed in number is lower than `1` or higher than `100`. A pattern matching expression makes for an implementation that reads very similarly to the specification:

```
def fromInt(n: Int): Option[GuessableNumber] = n match {
  case x if x <= 0   => None
  case x if x > 100 => None
  case _             => Some(new GuessableNumber(n))
}
```

With `fromInt` in place, let's look at one possible implementation of the `fromString` method:

```scala
def fromString(s: String): Option[GuessableNumber] =
  for {
    n      <- s.toIntOption
    number <- GuessableNumber.fromInt(n)
  } yield number
```

We implement `fromString` in terms of `toIntOption` (see Section 9.8) and our own `fromInt` factory method.

We can use a for comprehension to perform a sequence of two transformations: First, we try to turn `s` into an `Int`. If that works, we try to turn the `Int` value into a `GuessableNumber`. If any of the two steps fails, the sequence short-circuits and `fromString` returns a `None`.

If you run sbt's `test` command again, the `GuessableNumberTest` should be green now.

The game and the result

The game represents the state of the application, as it changes over time. In object-oriented programming with an imperative mindset, it would be implemented as a mutable class, because the state changes throughout the lifetime of the application. In functional programming, even things that change over time are typically represented by immutable data types — we return a new instance of `Game` to apply a change, instead of mutating the `Game` in-place.

Next to the `GuessableNumber.scala` file, let's add a file `Game.scala`. A first, incomplete version of the `Game` class looks like this:

```scala
package guessinggame.core.domain

final case class Game private (
    targetNumber: GuessableNumber,
    remainingGuesses: Int
)

object Game {
  def initialState(targetNumber: GuessableNumber): Game =
    Game(targetNumber, remainingGuesses = 5)
  def fromTargetNumber(n: Int): Option[Game] =
    GuessableNumber.fromInt(n).map(initialState)
}
```

`Game` is a case class with two fields: the target number and the remaining number of guesses before the game is over. In order to enforce the invariant that the initial number of remaining guesses is five, we make the case class constructor private and add two factory methods in the companion object.

The `initialState` factory method is straightforward. The `fromTargetNumber` method delegates to `fromInt` and `initialState`. The `fromInt` method returns an `Option[GuessableNumber]`. In order to turn that into an `Option[Game]`, we need to pass a function of type `GuessableNumber => Game` to the `map` method.

Our solution makes use of a feature you haven't seen yet: The `initialState` method expects a `GuessableNumber` and returns a `Game`. When you pass the name of a method as an argument where a function value is expected, that method gets converted to a function automatically, if it has the correct signature. In our case, the `initialState` method gets converted to a function of type `GuessableNumber => Game`.

This conversion of a method type to a function type is also called *eta expansion*. Thanks to eta expansion, there is no need for us to use any of the following notations:

```
def fromTargetNumber(n: Int): Option[Game] =
  GuessableNumber.fromInt(n).map(n => initialState(n))

def fromTargetNumber(n: Int): Option[Game] =
  GuessableNumber.fromInt(n).map(initialState(_))
```

Next to the `GuessableNumberTest.scala` file, let's add a file `GameTest.scala` and define a few test cases for the `Game` class and its companion object:

```
package guessinggame.core.domain

import minitest.SimpleTestSuite
import scala.annotation.tailrec

object GameTest extends SimpleTestSuite {
  test("Created game has 5 remaining guesses") {
    val received = Game.fromTargetNumber(50).map(_.remainingGuesses)
    assertEquals(received, Some(5))
  }
  test("Created game uses passed in target number") {
    val validTargetNumbers = Seq(1, 23, 99, 100)
    validTargetNumbers.foreach { n =>
```

```
      val received = Game.fromTargetNumber(n).map(_.targetNumber.value)
      assertEquals(received, Some(n))
    }
  }
  test("Game cannot be created from invalid target numbers") {
    val invalidTargetNumbers = Seq(-1, 0, 101)
    invalidTargetNumbers.foreach { n =>
      assertEquals(Game.fromTargetNumber(n), None)
    }
  }
}
```

Running the `test` command from the interactive sbt shell should not show any failing tests in the `GameTest` test suite.

Before we can complete the `Game` class, we need to define the `Result` type. According to the specification, the result of a guess can be either of the following:

- *Success*: The target number has been guessed
- *Game over*: The guess was wrong, and there are no more remaining guesses
- *Wrong*: The guess was wrong (either too high or too low)

If the result is wrong, there must be a hint about whether the guess was too high or too low. We can use a sealed type(see Section 9.1) to represent this as valid Scala code. Next to the `Game.scala` file, add a `Hint.scala` file:

```
package guessinggame.core.domain

sealed trait Hint
object Hint {
  case object TooHigh extends Hint
  case object TooLow  extends Hint
}
```

`Hint` has two subtypes, both of implemented as case objects.

The three different results can be represented by a sealed type as well. Add a new file `Result.scala` next to the `Hint.scala` file:

```
package guessinggame.core.domain

sealed trait Result
object Result {
  final case class Wrong(nextState: Game, hint: Hint) extends Result
  case object Success                                 extends Result
  case object GameOver                                extends Result
}
```

`Result` has three subtypes: `Wrong` is a case class that contains the next version of the application state and a hint about what was wrong. `Success` and `GameOver` are case objects, because they have no need to transport any data.

Now we can add the skeleton of the most important part of the `Game` class — the `processGuess` method takes a `GuessableNumber` and returns a `Result`. The `Game` class will now look like this:

```
package guessinggame.core.domain

final case class Game private (
    targetNumber: GuessableNumber,
    remainingGuesses: Int
) {
  def processGuess(guess: GuessableNumber): Result = Result.GameOver
}
object Game {
  def initialState(targetNumber: GuessableNumber): Game =
    Game(targetNumber, remainingGuesses = 5)
  def fromTargetNumber(n: Int): Option[Game] =
    GuessableNumber.fromInt(n).map(initialState)
}
```

Before implementing the `processGuess` method properly, we need to specify what its behaviour should be. Let's add the following helper methods to the `GameTest` object:

```
private def setupTest(target: Int, guess: Int): (Game, GuessableNumber) =
  (createGame(target), createNumber(guess))

private def createNumber(n: Int): GuessableNumber =
  GuessableNumber
    .fromInt(n)
    .getOrElse(sys.error("Guessable number must be between 1 and 100"))

private def createGame(targetNumber: Int): Game =
  Game
    .fromTargetNumber(targetNumber)
    .getOrElse(sys.error("Target number must be between 1 and 100"))
```

These methods help us to create a pair of a guessable number and a game with a specific target number. In production code, we don't want to escape the `Option` container, but in this test, we want to fail fast if the `Game` and `GuessableNumber` cannot be created — if the test setup doesn't even work, there is no point in continuing.

While the three methods above are needed for setting up the test data, we need two more helpers for describing our assertions. Before we implement these, add the following line to the import statements at the top of the `GameTest.scala` file:

```
import guessinggame.core.domain.Result._
```

Firstly, we need a method that allows us to assert that a given `Result` is a `Wrong`, additionally checking a condition on the `Wrong` value — for example, whether `targetIsLower` is `true` or `false`. The following method, which should be added to the `GameTest` object, allows us to do that:

```
private def assertWrong(result: Result, check: Wrong => Boolean): Unit =
  result match {
    case wrong: Wrong => assert(check(wrong))
    case other        => fail(s"Result is not Wrong, but $other")
  }
```

Secondly, we need a helper method for performing a specific number of guesses. The method should return the current result if one of these three conditions is fulfilled:

- the guess is correct

- the game is over
- it's the last of the specified number of attempts

Let's add the `performGuess` method to the `GameTest` object:

```
@tailrec
private def performGuess(
    game: Game,
    guess: GuessableNumber,
    attempts: Int
): Result =
  if (attempts > 1) {
    game.processGuess(guess) match {
      case res @ (Success | GameOver) => res
      case Wrong(next, _) =>
        performGuess(next, guess, attempts - 1)
    }
  } else game.processGuess(guess)
```

Here we make use of tail recursion, counting down the number of remaining attempts when calling ourselves. Processing a guess relies on a pattern binder for a pattern alternative to match and bind either the `Success` or `GameOver` singleton object.

Now that we have all our helper methods in place, let's add the following tests to the `GameTest` object, which specify the expected behaviour of the `processGuess` method:

```
test("Processing guess leads to game over after five attempts") {
  val (game, guess) = setupTest(target = 50, guess = 1)
  val endResult     = performGuess(game, guess, 5)
  assertEquals(endResult, GameOver)
}
test("Processing correct guess leads to success") {
  val (game, guess) = setupTest(target = 50, guess = 50)
  val endResult     = performGuess(game, guess, 1)
  assertEquals(endResult, Success)
}
test("Processing guess that is too high") {
  val (game, guess) = setupTest(target = 50, guess = 51)
  val endResult     = performGuess(game, guess, 1)
  assertWrong(endResult, _.hint == Hint.TooHigh)
}
test("Processing guess that is too low") {
  val (game, guess) = setupTest(target = 50, guess = 49)
  val endResult     = performGuess(game, guess, 1)
  assertWrong(endResult, _.hint == Hint.TooLow)
}
```

If you execute the `GameTest`, most of these tests should still fail, because the `processGuess` method defined in the `Game` class always returns `GameOver`. Let's replace this with a proper implementation:

```
def processGuess(guess: GuessableNumber): Result = guess match {
  case `targetNumber` => Result.Success
  case _ if isOver    => Result.GameOver
  case n              => Result.Wrong(nextState, hint(n))
}

private def nextState: Game =
  if (isOver) this
  else this.copy(remainingGuesses = remainingGuesses - 1)

  private def isOver: Boolean = remainingGuesses <= 1

  private def hint(n: GuessableNumber): Hint =
  if (n > targetNumber) Hint.TooHigh
  else Hint.TooLow
```

Again, the pattern matching expression makes our code look very similar to the specification.

The presentation layer

We're done with the actual domain logic now — apart from the holes left to you as an exercise. The next piece of the puzzle is the presentation layer, or view model. There is not a lot of presentation logic in this example application. Mostly, the presentation layer consists of static `String` values.

Nevertheless, there are places where `String` values need to be assembled dynamically, depending on values from the domain model. These can and should be implemented as pure functions as well. In principle, those pure functions can also be covered with unit tests, if you want to make sure that the user interface conforms to the specification provided by the product owner. We don't do that in this case study, but I encourage you to try it out.

In the `src/main/scala` directory, please make sure you have a subdirectory `guessinggame/core/presentation`. In that directory, add a file `Presentation.scala` with the following content:

```
package guessinggame.core.presentation

import guessinggame.core.domain._

object Presentation {
  val introduction: String =
    """
    ____                     _                ____
    / ___|_   _  ___  ___ ___(_)_ __   __ _   / ___| __ _ _ __ ___   ___
   | |  _| | | |/ _ \/ __/ __| | '_ \ / _` | | |  _ / _` | '_ ` _ \ / _ \
   | |_| | |_| |  __/\__ \__ \ | | | | (_| | | |_| | (_| | | | | | |  __/
    \____|\__,_|\___||___/___/_|_| |_|\__, |  \____|\__,_|_| |_| |_|\___|
                                      |___/

     #I just came up with a number between 1 and 100 (inclusive)!
     #If you guess it, you'll win!
     #If you're wrong, I will tell if it's higher or lower.
     #You have five attempts. Good luck!""".stripMargin('#')

  val prompt: String = "\n> "

  val invalidNumber: String =
    "\nPlease enter a valid number between 1 and 100."

  def successMessage(number: GuessableNumber): String =
    s"\nThat's damn right! ${number.value} it is!"

  def hint(guess: GuessableNumber, hint: Hint): String = {
    val comparator = hint match {
      case Hint.TooHigh => "lower"
      case Hint.TooLow  => "higher"
    }
    s"""
    |Your guess, ${guess.value}, is wrong.
    |The target number is ${comparator} than that.""".stripMargin
  }

  val gameOver: String =
    """

    ____                                            _
   / ___| __ _ _ __ ___   ___    _____   _____ _ __| |
  | |  _ / _` | '_ ` _ \ / _ \  / _ \ \ / / _ \ '__| |
  | |_| | (_| | | | | | |  __/ | (_) \ V /  __/ |  |_|
   \____|\__,_|_| |_| |_|\___|  \___/ \_/ \___|_|  (_)

    """

  val problemWithSetup: String =
    "Problem setting up the game. Will exit. So sorry! :("
}
```

The Presentation object contains a lot of fields defined as multi-line strings, particularly the two banners, which were created with the help of the Ascii-Art Generator[5]. There are also a few methods whose return value depends on the input. An example of that is the hint method. The important thing is that everything in this object is pure — there are no effects whatsoever, and all the methods are referentially transparent.

Exercise

Add a PresentationTest that tests the behaviour of the methods defined in the Presentation object.

10.5 The imperative shell

We have managed to implement the domain logic and a lot of the presentation logic as a purely functional core. This allowed us to cover the most important parts of our application with unit tests. What's missing is the code that orchestrates everything and provides the missing interaction with the outside world — the imperative shell.

[5] https://www.ascii-art-generator.org/

Input and output

The imperative shell is all about input and output. Our guessing game is a command-line application: We need to read the numbers the player types in from the standard input, and we need to print various messages to the standard output.

Let's start with the output and encapsulate all message printing in a class called `MessagePrinter`. The imperative shell is typically much more difficult to test than the functional core. However, since it's mostly free from logic, it doesn't need as many tests as the core, where all the complex decision making is implemented.

In this case study, we won't create any tests for the imperative shell. Nevertheless, let's not make it more difficult than necessary to add tests: Instead of printing to the standard output directly, we'll endow our `MessagePrinter` with a `java.io.PrintStream` that needs to be passed to its constructor. This way, we can pass in either `System.out` or another `PrintStream` that captures all output in a `String`.

Make sure the subdirectory `guessinggame/shell` exists in the `src/main/scala` directory. In the `shell` directory, let's add a file `MessagePrinter.scala` that looks like this:

```scala
package guessinggame.shell

import guessinggame.core.domain.{GuessableNumber, Hint}
import guessinggame.core.presentation.Presentation

import java.io.PrintStream

private[shell] class MessagePrinter(out: PrintStream = System.out) {
  def showIntro(): Unit = out.println(Presentation.introduction)

  def showErrorMessage(): Unit = out.println(Presentation.invalidNumber)

  def showSuccessMessage(
      targetNumber: GuessableNumber
  ): Unit =
    out.println(Presentation.successMessage(targetNumber))

  def showGameOverMessage(): Unit = out.println(Presentation.gameOver)

  def showHint(
      guess: GuessableNumber,
      hint: Hint
  ): Unit = out.println(Presentation.hint(guess, hint))

  def showProblemWithSetup(): Unit = println(Presentation.problemWithSetup)
}
```

The `MessagePrinter` is only responsible for printing to a print stream — it delegates to the purely functional presentation layer for everything else, and it's devoid of any branching logic.

Note that we provide a default argument for the `out` parameter, so that creating an instance of this class for non-test purposes becomes as convenient as possible. Also, please pay attention to the visibility modifier that makes the whole class only visible within the `guessinggame.shell` package. This prevents us from accidentally using it outside of the shell, particularly in the functional core.

For the input, we're going to use a similar strategy. We want to be able to read from somewhere else than standard in. In the `shell` directory, let's add a `package.scala` file in which we can define a package object for the `guessinggame.shell` package:

```
package guessinggame

package object shell {
  type InputReader = String => String
}
```

Here, we define a type alias for `String => String`. An `InputReader` takes an input prompt to be displayed and returns a `String` from some input. While `String => String` looks like the type of a pure function, it's actually far from it — we want to read from the standard input after all. Unfortunately, the Scala type system can lie about these things, and you won't notice. Our strategy is to put all impure functions into the imperative shell. We don't expect any function defined in the shell to be pure, regardless of its signature.

We're not doing anything with the `InputReader` type alias now. However, it will be used in the main application.

The main application

We have already taken care of all of the I/O by means of the `MessagePrinter` and `InputReader` types. What's left is mostly to glue everything together in an executable application. This application must implement the game loop and do some orchestration, but it will delegate almost all the logic to the functional core.

Let's create a file `Application.scala` next to the `MessagePrinter.scala` and `package.scala` files and add a first skeleton of the `Application` class:

```scala
package guessinggame.shell

import guessinggame.core.domain._
import guessinggame.core.presentation.Presentation

import scala.io.StdIn
import scala.util.Random
import scala.annotation.tailrec

private[shell] class Application(
    messagePrinter: MessagePrinter = new MessagePrinter,
    inputReader: InputReader = prompt => StdIn.readLine(prompt),
    randomGenerator: () => Int = () => Random.nextInt(99) + 1
) {
  def run(): Unit = {
    messagePrinter.showIntro()
    loop(launchOrExit())
  }

  @tailrec
  private def loop(game: Game): Unit = {
    // to be implemented
  }

  private def readGuess(prompt: String): Option[GuessableNumber] =
    GuessableNumber.fromString(inputReader(prompt))

  private def launchOrExit(): Game =
    Game.fromTargetNumber(randomGenerator()).getOrElse {
      messagePrinter.showProblemWithSetup()
      sys.exit(-1)
    }
}
```

Let's walk through this class step by step. First of all, we make sure this class is only visible within the `guessinggame.shell` package, just as we did with the

`MessagePrinter`. In the class parameter list, there are three parameters, each of them with reasonable default arguments for a non-test environment:

- the `messagePrinter` to use; for testing purposes, it's possible to inject a `MessagePrinter` that prints to a different print stream than standard out, so that the output can easily be inspected
- the `inputReader` to use: the default is to read a line from `StdIn`; for testing purposes, reading from different inputs is conceivable
- the `randomGenerator` to use: random number generation is also impure, as it entails talking to the outside world; the default uses the `nextInt` method from `scala.util.Random`; for testing purposes, it's possible to inject a different parameterless function that is deterministic, always returning the same constant value

The `run` method is the entry point to the application. It will show the intro banner and start the game loop, passing in a newly created `Game` instance.

The `loop` method expects a `Game` and is supposed to be tail-recursive. This means it will call itself until there is a successful guess or the game over condition has been reached. We'll look at an implementation of this method soon.

The `readGuess` method makes use of the supplied `InputReader` and tries to parse the input `String` as a `GuessableNumber`, delegating to the respective factory method in the `GuessableNumber` companion object. We're going to use this method in the `loop` method.

The `launchOrExit` method makes use of the provided `randomGenerator` in order to create a `Game` based on a random target number. Remember that `fromTargetNumber` returns an `Option[Game]`, because this factory method enforces the invariants of the `Game` type — the target number must be in the range of 1 to 100. While our `randomGenerator` should not generate any numbers outside of that range, we still handle the possibility. The best thing we can do in such a case is to tell the player that there was a problem setting up the game and do an early exit. It's an unexpected failure, probably due to a programming error, that we can't handle properly.

Implementing the loop

The game loop is the crucial piece of the imperative shell. While we try to keep the imperative shell free from any logic, you will see that it's not always entirely possible.

In the `loop` method, we must react differently depending on whether the player typed in a valid number or not. If they did, the action we must take depends on the result of the guess. However, the control flow happening here is rather straightforward and can be implemented with two pattern matching expressions. The more complex domain logic that computes the result of a guess is located in the functional core, which is covered with unit tests.

Without further ado, let's look at our `loop` method:

```
@tailrec
private def loop(game: Game): Unit = {
  import messagePrinter._
  readGuess(Presentation.prompt) match {
    case None =>
      showErrorMessage()
      loop(game)
    case Some(guess) =>
      game.processGuess(guess) match {
        case Result.GameOver => showGameOverMessage()
        case Result.Success  => showSuccessMessage(game.targetNumber)
        case Result.Wrong(nextState, hint) =>
          showHint(guess, hint)
          loop(nextState)
      }
  }
}
```

First of all, you'll notice an interesting importing technique: We're doing a locally scoped wildcard import of all the fields and methods of the `messagePrinter` — you cannot only import classes or singleton objects, but also the fields and methods of an instance of a class. This comes in handy so that we can call the methods defined in `MessagePrinter` directly instead of having to write `messagePrinter.showErrorMessage()` we can write `showErrorMessage()`.

You'll notice that the first thing our `loop` method does is to read a guess made by the player. Now, if the player did not type in a valid number between 1 and 100, `readGuess` will return `None`. We use pattern matching to distinguish that case from the happy path. If `readGuess` returns `None`, we will show an error message and call `loop` again. Since the number of remaining guesses must not be decreased, we pass in the same `game` we received. If the player typed in a valid number between 1 and 100, the second case will match, which expects the `Option[GuessableNumber]` to be defined.

If we have a valid guess, we delegate to the game when it comes to processing the guess. We don't care about the logic hidden in the Game class. All we need to do after calling processGuess is to pattern match on the returned Result. Remember that a Result can be either GameOver, Success, or Wrong.

In case it's a GameOver or a Success, all we need to do is show the corresponding messages to the player. In doing so, we leave the loop method and the game loop comes to an end.

In case the Result is a Wrong, we destructure it, binding its fields to the nextState and targetIsLower identifiers. We'll show a hint to the player and call the loop method again, this time with a new version of the Game, which was provided to us in the nextState field of the Wrong. Since we call loop recursively, the game loop continues.

A launchable application

I mentioned that the run method serves as the entry point for the application. What's still missing is a singleton object with a main method, so that we have an application that the Java runtime can start (see Section 3.2). The main method merely needs to create a new instance of the Application class and call its run method. Let's add an Application companion object to the Application.scala file:

```scala
object Application {
  def main(args: Array[String]): Unit = {
    val app = new Application()
    app.run()
  }
}
```

We create a new Application without passing in any arguments — the default arguments we defined for the class parameters are just fine for bootstrapping an application for a non-test environment.

Now you should be able to run our guessing game by executing the run command in the interactive sbt shell.

Exercises

1. Create a helper class for capturing the output sent to a `PrintStream` and providing it as a `String`. You will likely need a `ByteArrayOutputStream` to achieve this. Also, it probably helps to read the documentation for PrintStream[6] before implementing the helper.
2. Use the helper class to create an `ApplicationTest`. In this test the `Application` should be injected with a `MessagePrinter` that uses a `PrintStream` provided by your helper. Moreover, you can pass in an `inputReader` and a `randomGenerator` with deterministic return values. Test that the captured output is what you expect if the first guess is correct.

10.6 Packaging the application

One last bit remains before we can consider our task done. We need a way to ship an executable application, so that people can actually play our game. We'll keep things simple and decide to ship it as a ZIP archive that contains scripts for launching the game on the command-line, both for Linux/Mac OS X, and for Windows.

Since this is such a common task, there is an sbt plugin we can use for this: sbt Native Packager[7]. Let's create a file `plugins.sbt` in the `project` subdirectory of the `guessing-game` directory. In that file, we will add a single line that makes sure the sbt Native Packager plugin will be available in our sbt project:

```
addsbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.4.1")
```

Then, in the `build.sbt` file in the `guessing-game` directory, we're going to add another line to actually enable the plugin that allows us to package our command-line application:

```
enablePlugins(JavaAppPackaging)
```

If you are still in the interactive sbt shell, you'll need to quit it. In your terminal, in the root directory of the `guessing-game` sbt project, use the following command:

[6]https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/PrintStream.html
[7]https://www.scala-sbt.org/sbt-native-packager/

```
$ sbt universal:packageBin
```

This will package all your code, the Scala standard library, and launcher scripts for Linux/OS X and Windows into a ZIP file. You can find the ZIP file in the `target/universal` directory. If you unzip it, the extracted directory will have the following contents:

```
.
├── bin
│   ├── guessing-game
│   └── guessing-game.bat
└── lib
    ├── default.guessing-game-0.1.0-SNAPSHOT.jar
    └── org.scala-lang.scala-library-2.13.3.jar
```

The `lib` directory contains two *jar files*. The first one contains all the compiled Scala classes of our application. The second one contains the compiled Scala classes of the Scala standard library, which is the only dependency of our application. If we had more dependencies, their JAR files would end up int the `lib` directory as well.

Since I'm on Linux, I can start the game by launching the `guessing-game` script. The same script is applicable for Mac OS X users. Windows users will have to launch `guessing-game.bat`. The only requirement is that users have a Java runtime environment on their machine. For me, launching the game looks like this:

```
$ bin/guessing-game

     ____                     _                ____
    / ___|_   _  ___  ___ ___(_)_ __   __ _   / ___| __ _ _ __ ___   ___
   | |  _| | | |/ _ \/ __/ __| | '_ \ / _` | | |  _ / _` | '_ ` _ \ / _ \
   | |_| | |_| |  __/\__ \__ \ | | | | (_| | | |_| | (_| | | | | | |  __/
    \____|\__,_|\___||___/___/_|_| |_|\__, |  \____|\__,_|_| |_| |_|\___|
                                      |___/

I just came up with a number between 1 and 100 (inclusive)!
If you guess it, you'll win!
If you're wrong, I will tell if it's higher or lower.
You have five attempts. Good luck!

>
```

10.7 Summary

In this chapter, we have developed a small, but complete command-line application. You have seen how many of the concepts and language constructs you have learned about in the book interact. You saw how to separate logic and effects by employing a “functional core, imperative shell” architecture, and how this allowed us to test all of the domain logic of our application with ease.

You will find the complete code of the guessing game project on GitHub[8]. If you had difficulties implementing some of the missing pieces, or if you want to compare your solution with the one I came up with, feel free to take the time and look at the code.

In the next and final chapter, we’ll look back at what you have learned and what the next steps might be.

[8]https://github.com/dwestheide/guessing-game/

11. Downtime

Exciting though it might be, nobody can be on and about exploring forever without rest. It’s time for some downtime, giving you the chance to refresh your mind for a while — and to reflect on what you have learned so far. Let’s look back at the journey so far and where it may lead you.

11.1 Summary

We have covered a lot of ground throughout this book. You learned how to use the REPL, which is an important tool in your day-to-day work when programming in Scala. It allows you to quickly try something out without the ceremony of creating any source files and tests. It’s also a great way of exploring new programming interfaces, whether it’s the Scala standard library or other third-party open-source libraries.

Speaking of tools, you got familiar with the basics of sbt, the scala build tool. You know how to compile your source code, how to run your tests, and how to start a new REPL session from the interactive sbt shell. By working through the examples, you should also have become comfortable with Metals, the Scala extension for VS Code and other editors. Moreover, you were introduced to the Minitest library, a tiny open-source library for writing tests.

Getting comfortable with those tools that are pervasive in the ecosystem of a programming language is important. In addition to that, you already learned quite a bit about Scala as a language, and about the idea of functional programming that is one of its core foundations. You learned that functions can be first-class values that can be passed around: Functions or methods can take other functions as parameters, or they can return new functions. This is a very powerful idea that is at the very heart of functional programming — and it allows for abstractions that are hardly possible otherwise. A great example of this is Scala’s collection library, which we took a peek at by looking at a few of the methods available for processing sequences. In addition to that, you learned about the importance of immutable values and data structures, and how pure functions make it easier to reason about your code.

You saw how pattern matching, case classes, and for comprehensions help you to write readable code that expresses the intent rather than what needs to be done to achieve it. You also learned how to use value classes and, later on, sealed types and case objects to make the intent of your code even clearer, and how to use implicit classes to extend existing types with new methods.

Of course, we also looked at Scala's object-oriented side. An interesting concept here is the idea of a singleton object as a language construct. Moreover, Scala's mechanisms for defining nested packages and its support for specifying the visibility of classes, values, or methods on the level of these nested packages is quite powerful. You saw how it can help you design modules with clear boundaries.

We took a first glance at Scala's powerful type system, and you got an idea of how polymorphic data types and methods can help with abstracting over concrete types, making your code more re-usable. You learned about the important difference between strict and lazy evaluation and how to make use of lazy evaluation in Scala, using by-name parameters and lazy values.

Finally, you saw how Scala tries to avoid the million-dollar mistake by establishing an alternative to null references, namely `Option[A]`. Moreover, you saw that for comprehensions are not just for sequences, but can also be used to process options, which are just a special kind of collection.

In a case study, we developed a small example application that demonstrates how all the different pieces come together, and how logic and effects can separated in practice into a purely functional core and an imperative shell.

11.2 The road ahead

With all you have learned so far, you should have the tools to deepen your knowledge of Scala through further exercises, and to write simple applications, splitting them into cleanly separated modules.

However, while you have a good overview of what the Scala language has to offer, and what writing Scala code feels like, the learning has just begun. If this book has made you curious and excited about Scala, you're invited to read and work through the second book in the series as well, *Scala from Scratch: Understanding*. As of this writing, the goal is to have it released in the summer of 2020.

There are quite a few things we haven't looked at yet in this book, and which you can expect to be covered in depth in the upcoming book:

For one, you only took a brief glance at sequences. For many applications, it's important to get a deeper understanding of all the other data structures the Scala collections library has to offer. Also, we haven't uncovered the full strength of using sealed traits or sealed abstract classes yet. Together with case classes and case objects, they allow you to design *algebraic data types*, which are a great tool for modelling the domain of your application. The guessing game case study gave you a first impression of that.

Then, there is the whole topic of functional error handling: In this book, we have sometimes used exceptions to assert invariants. Later we switched to returning options to make our methods referentially transparent. Scala has a few other data types available that allow you to handle errors without losing information about what went wrong, and it's definitely a good idea to embrace this way of error handling when programming in Scala.

As already mentioned, functions as values is one foundation of functional programming. There are a few powerful techniques that rely on this foundation, like partially applied functions and currying. These allow you to write highly re-usable code with minimal boilerplate.

Additionally, we have just scratched at the surface of the powerful type system. Many Scala libraries are based on advanced features like implicit parameter lists. At some point, it will be inevitable to delve into these advanced topics as well.

One of the promises of Scala is that, as a functional programming language, it's really well-suited for concurrent programming. The Scala standard library has some neat stuff supporting that, and there are some third-party libraries that go even further.

Also, while you learned the basics of testing in Scala, there is an advanced testing technique which is particularly suited to functional code — property-based testing. It's definitely worthwhile exploring this technique.

Finally, you don't really know anything about the Scala ecosystem yet, so you are limited to what's available in the Scala standard library for now. In the second book, you will learn how to make use of third-party libraries and you'll get introduced to a few popular ones. You'll also get a deeper understanding of sbt, which is the tool that allows you to build, test, and release your applications or libraries.

11.3 Taking action

So you have finished this book, you want to continue, but *Scala from Scratch: Understanding* is still taking its time to be released? Here are a few things you can do in the meantime:

The issue of documentation

In this book, you have not learned how to document your Scala code. It's a bit difficult to put long listings of code into a book without making things look ugly. However, the existing guides for *ScalaDoc*, Scala's code documentation format and tool, are really good. I invite you to take a look at ScalaDoc section of the style guide[1].

ScalaBridge

If you are a member of an underrepresented group in tech, you can continue learning as a student at one of the ScalaBridge workshops that are being organised throughout the world by passionate members of the Scala community. The ScalaBridge website[2] has a list of upcoming events.

Many students attending the workshops have no prior experience with Scala, or even with programming in general. I would like to encourage you to think about becoming a mentor at one of the workshops as well. Teaching others what you have already learned will expand and strengthen your knowledge — having to explain concepts and answering questions, you are forced to think about them in a new way and look at things from entirely new perspectives. Plus, it's great for connecting with other people in the Scala community, and you'll be doing your part in building an inclusive Scala community.

Scala Exercises

If you want to continue learning, there is a great website called Scala Exercises[3] that has tons of exercises that can be solved in the browser. I would recommend to start with the exercises for the Scala standard library for now.

[1] https://docs.scala-lang.org/style/scaladoc.html
[2] https://scalabridge.org
[3] https://www.scala-exercises.org/

An even better way of learning Scala is to join the Scala track at Exercism[4]. Exercism allows you to practice your programming skills in various languages, with the help of a team of very welcoming mentors.

Scala Best Practices

In this book, I've done my best to teach you more than just Scala the language. I've tried to give you an idea of how to write idiomatic Scala, about best practices that have emerged over the years, and about pitfalls that people have learnt to avoid.

That being said, Nicolas Rinaudo[5] went out of his way to compile a much more extensive collection of best practices. All of them are explained in full detail, stating the motivation for each practice and an example. If you want to continue your journey with Scala 2.13, Nicolas' Scala Best Practices[6] are a real treasure trove. Some of the practices refer to concepts we haven't discussed yet. But most of them should already be valuable to you now.

[4]https://exercism.io/tracks/scala
[5]https://twitter.com/nicolasrinaudo
[6]https://nrinaudo.github.io/scala-best-practices/