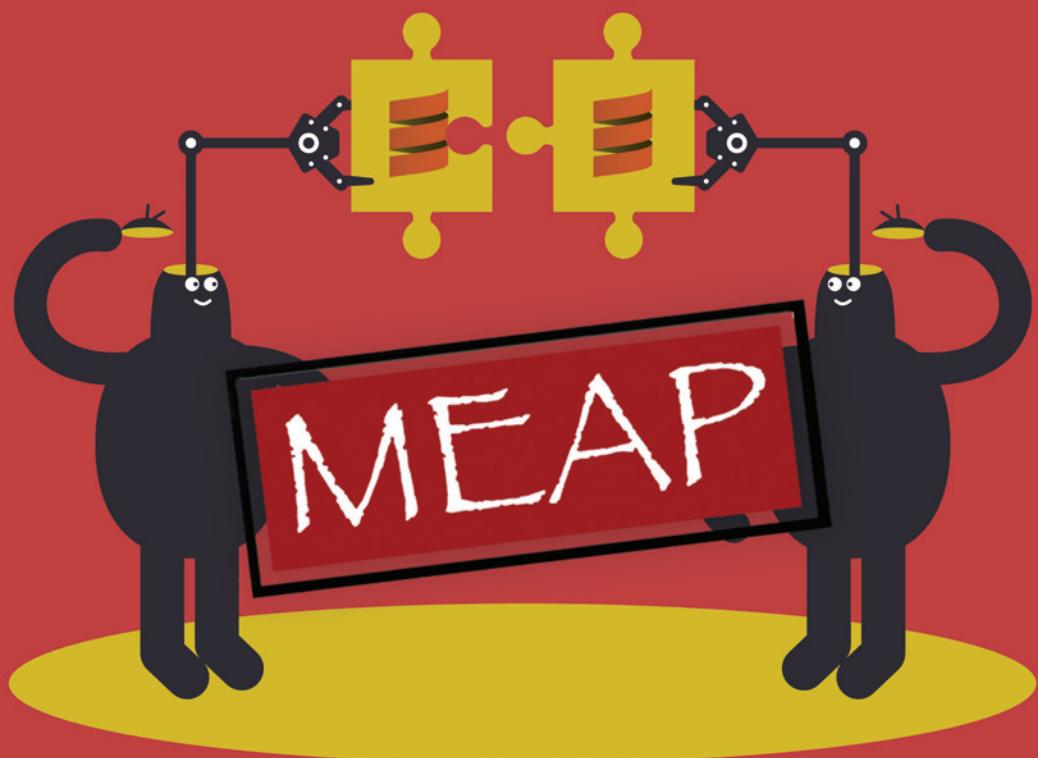


GET PROGRAMMING WITH SCALA



Daniela Sfregola



MANNING



MEAP Edition
Manning Early Access Program
Get Programming with Scala
Version 10

Copyright 2021 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP edition of *Get Programming with Scala*, and welcome to the magical world of Scala! I am really excited to share with you the units for this book that I have written so far: I hope you'll find them useful and that you'll provide your feedback to make this book even better.

Until recently, Object Oriented Programming (OOP) was considered the style to follow. But with the growing need for handling concurrency and multi-core processing, Functional Programming (FP) is becoming more and more popular. We are in a transition period where developers are familiar with the OOP paradigm, but they would like to learn more about FP. Scala is establishing itself in this niche, as it allows programmers to combine the two approaches.

Scala is a beautiful, powerful language that is extremely pleasant to use, but it has the reputation of being difficult to learn. With *Get Programming with Scala*, I want to teach you the basics of Scala and Functional Programming. After reading this book, you should be able to write fully working and practical programs in Scala, and you should have the tools to keep exploring the world of Functional Programming in Scala on your own. I am going to assume you have done some programming with an object-oriented language such as Python, Java, or C++. Do not worry if you don't know any functional programming: you are going to discover how useful it can be while reading this book!

Get Programming with Scala will begin by reviewing how you can write object-oriented code in Scala by learning about variables and values, classes and objects. You will then start your journey towards a more functional mind-set by discovering how functions and types can be used to produce powerful abstractions. You will also learn about testing and how to recognize and fix code that "doesn't look quite right."

Throughout the book, you will see lots of code examples and exercises to help you master the topics of each unit. The source code presented in the lessons and the solutions to the exercises are available online at <https://github.com/DanielaSfregola/get-programming-with-scala>.

I hope you'll find this book useful. Please post any questions, comments, or suggestions in the [Author Online Forum](#): your feedback is invaluable in improving *Get Programming with Scala*!

Thanks!

—Daniela Sfregola

brief contents

UNIT 0: HELLO SCALA!

Lesson 1: Why Scala?

Lesson 2: The Scala Environment

Lesson 3: sbt – Scala Build Tool

UNIT 1: THE BASICS

Lesson 4: Values and Variables

Lesson 5: Conditional Constructs and Loops

Lesson 6: Function as the most fundamental block of code

Lesson 7: Classes and Subclasses to represent the world

Lesson 8: The Vending Machine

UNIT 2: OBJECT-ORIENTED FUNDAMENTALS

Lesson 9: Import and Create Packages

Lesson 10: Scope your code with Access Modifiers

Lesson 11: Singleton Objects

Lesson 12: Traits as interfaces

Lesson 13: What time is it?

UNIT 3: HTTP SERVER

Lesson 14: Pattern Matching

Lesson 15: Anonymous Functions

Lesson 16: Partial Functions

Lesson 17: HTTP API with http4s

Lesson 18: The Time HTTP Server

UNIT 4: IMMUTABLE DATA AND STRUCTURES

Lesson 19: Case Classes to structure your data

Lesson 20: Higher Order Functions

Lesson 21: What is Purity?

Lesson 22: Option

Lesson 23: Working with Option: map and flatMap

Lesson 24: Working with Option: for-comprehension

Lesson 25: Tuple and Unapply

Lesson 26: Rock, Paper, Scissors, Lizard, Spock!

UNIT 5: LIST

Lesson 27: List

Lesson 28: Working with List: map and flatMap

Lesson 29: Working with List: properties

Lesson 30: Working with List: element selection

Lesson 31: Working with List: filtering

Lesson 32: Working with List: sorting and other operations

Lesson 33: The Movies Dataset

UNIT 6: OTHER COLLECTIONS AND ERROR HANDLING

Lesson 34: Set

Lesson 35: Working with Set

Lesson 36: Map

Lesson 37: Working with Map

Lesson 38: Either

Lesson 39: Working with Either

Lesson 40: Error Handling with Try

Lesson 41: The Library Application

UNIT 7: CONCURRENCY

Lesson 42: Implicit and Type Classes

Lesson 43: Future

Lesson 44: Working with Future: map and flatMap

Lesson 45: Working with Future: for-comprehension and other operations

Lesson 46: Database queries with Quill

Lesson 47: The Quiz Application: Part 1

UNIT 8: JSON (DE)SERIALIZATION

Lesson 48: JSON (De)serialization with circe

Lesson 49: Lazy Evaluation

Lesson 50: The IO type

Lesson 51: Working with the IO type

Lesson 52: Testing with ScalaTest

Lesson 53: The Quiz Application: Part 2

About this book

Until recently, Object-Oriented Programming (OOP) was considered the style to follow. But with the growing need for handling concurrency and multi-core processing, Functional Programming (FP) is becoming more and more popular. We are in a transition period where developers are familiar with the OOP paradigm, but they would like to learn more about FP. Scala is establishing itself in this niche, as it allows programmers to combine the two approaches.

Scala is a beautiful, powerful language that is extremely pleasant to use, but it has the reputation of being difficult to learn. With Get Programming with Scala, you are going to learn the basics of Scala and Functional Programming. After reading this book, you'll be able to write complete applications in Scala, and you'll have the tools to keep exploring the world of Functional Programming in Scala on your own. I am going to assume you have done some programming experience with any object-oriented language such as Python, Java, or C++. You do not need any prior functional programming knowledge: you are going to discover its beauty and usefulness in this book.

Get Programming with Scala will begin by showing you how to write object-oriented code in Scala by learning about values, classes, and objects. You will then start your journey towards a more functional mindset by discovering how you can use functions and types to produce powerful abstractions. You will also learn how to recognize and fix code that "doesn't look quite right."

Throughout the book, you will see lots of code examples and quizzes to help you master the topics of each unit. Its source code, exercises, and solutions are available on GitHub at <https://github.com/DanielaSfregola/get-programming-with-scala>.

Unit 0

Hello Scala!

Welcome to Get Programming with Scala! In Unit 0, you'll have an overview of Scala and everything you need to know to set up your development environment and start coding. In particular, you'll see the following subjects:

- Lesson 1 illustrates the key Scala features and why it is such a great programming language to learn. You'll discover the typical execution flow of a Scala program, and you'll get a glance at some Scala code.
- Lesson 2 shows you how to install and run the Scala REPL, a crucial tool to play and experiment. You'll write code snippets and see how the compiler interprets them.
- Lesson 3 introduces you to SBT – the Scala Build Tool. You'll install it and write your first Scala program by organizing your folders and source files according to the SBT standard.

After setting your environment up, you'll continue with Unit 1, where you'll review the basics of object-oriented programming in Scala.

1

Why Scala?

After reading this lesson, you'll be able to:

- Discuss the advantages of adopting Scala
- Describe the execution flow of a typical Scala program
- Define the key features of the Scala language

In this lesson, you'll discover why Scala is an excellent language to learn and why its adoption is increasing so rapidly. You'll see how Scala relates to the JVM and the key features that make it unique. You'll also start looking at snippets of Scala code to get an idea of its appearance. After giving you an overview of the Scala language, you'll continue with the next lesson, in which you'll install the Scala REPL and use it to interpret snippets of code.

1.1 Why Scala?

Why should you spend time and effort in learning Scala? Why is it becoming so popular? What are the advantages of adopting it? Let's discuss its main selling points.

THE JVM

The JVM is the standard platform for running Scala programs. Sun Microsystems introduced it in 1994 – more than 25 years ago! Since then, the industry has been extensively relying on it. The Java Community has also been extremely active, and it has produced an impressive amount of libraries. Thanks to its integration with Java, you can use all these libraries in your Scala programs: this is also true for any Java legacy code that you or your company may have.

A HYBRID LANGUAGE

Scala manages to combine two programming techniques that are considerably different: the object-oriented style with the functional one. When executing code on the JVM, the object-oriented approach can be more performant but prone to errors. When using mutable state, your program will allocate less memory: every time a change occurs, it will change the data in place. However,

sharing state can cause your application to suffer from data inconsistency issues due to multiple processes accessing and modifying the same portion of data.

A functional approach can be more readable and reusable but not as performant. Thanks to immutable data and structures, you can guarantee your program's correctness when dealing with concurrency: your data never changes, so it is safe to share it. Your code will also be easier to understand and reuse because all its components will be independent of external factors outside its control. But recreating data rather than updating can be an expensive operation, although it has massively improved thanks to numerous memory optimizations and efficient garbage collection strategies in recent years.

In Scala, you do not have to stick to a particular style, but you can take advantage of one or the other paradigm depending on the specific task you are solving. Have a look at figure 1.1 for a comparison of how the two approaches tackle different programming tasks.

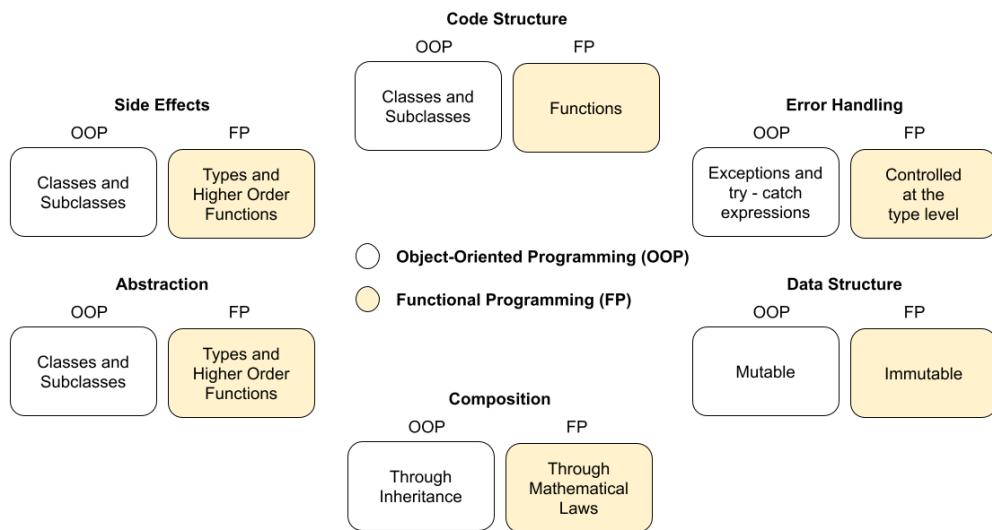


Figure 1.1: Comparison of the object-oriented and functional programming styles and how they handle different programming tasks.

CONCISE SYNTAX

Scala's programming style is relatively concise, particularly when compared to other languages like Java. Having a compact syntax can increase both the productivity and readability of your program. At the beginning of your journey with Scala, you may find it quite overwhelming. In this book, syntax diagrams will help you master new topics and make your learning path a bit smoother.

FLEXIBILITY

Scala is extremely flexible: you can achieve the same goal in more than one way, making the language extremely fun and exciting to use. The opportunity to choose between different programming paradigms allows you to gradually shift your mindset from one approach to another

without committing to a specific style since day one. For example, you can dip your toe in the functional programming world without any long-term commitment. In this book, you'll start writing Scala code using an object-oriented style and then gradually move to a more functional one.

CONCURRENCY

Thanks to its use of immutable data structures and expressive type system, dealing with concurrency should be less prone to errors than in other languages. As a result, Scala programs tend to utilize resources more efficiently, and they usually perform better under pressure.

BIG DATA AND SPARK

Thanks to Scala's features and optimizations at its compile level, the community has developed new performant tools for big data processing. Apache Spark is the most popular of these tools. Thanks to Scala's lazy evaluation, which is a topic you are going to discover in unit 8, Spark can perform optimizations at compile time that have huge impacts on its runtime performance:

"Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk."

Source: <https://spark.apache.org/>

1.2 Scala and the JVM

Scala takes its name from the word "scalable": Martin Odersky and his team designed it in 2004 with the intent of creating a language for the Java Virtual Machine (JVM) that can easily handle high volumes of requests.

To understand the execution of a Scala program, let's compare it with the typical execution flow of a Java Program: figure 1.2 provides a visual representation of the two processes. The JVM is a machine to perform tasks by executing a well-defined set of operations, called *bytecode*. A JVM language aims to translate its code into executable JVM bytecode, usually formed by multiple files with extension `*.class`. When coding in Java, you save your source files with extension `*.java` and use the compiler `javac` to produce a jar file containing the generated bytecode. When writing Scala code, your source files have extension `*.scala`. The Scala compiler called `scalac` is in charge of translating the code into bytecode. You can seamlessly add Java sources and depend on Java libraries in your Scala codebase. The Scala compiler fully integrates with the bytecode that the Java compiler produces, making the integration between the two languages straightforward.

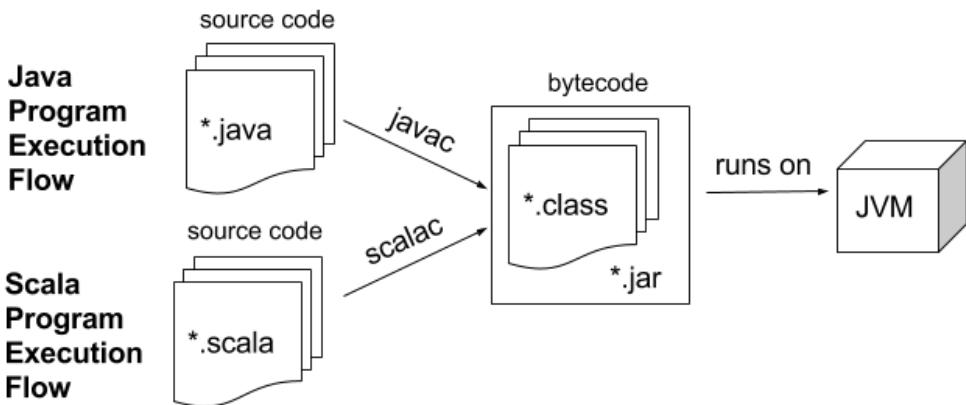


Figure 1.2: Comparison between the execution of a Java program with a Scala one. The Java source code has extension `*.java`, and `javac` translates it into bytecode files with extension `*.class`. You save Scala code in files with extension `*.scala`, and `scalac` is the compiler responsible for converting them into bytecode. The JVM is now ready to run the bytecode, usually conveniently grouped in a `*.jar` file.

Scala as a multi-platform language

Odersky and his team designed Scala for the JVM. A few open-source projects (some more experimental than others) are trying to develop new compilers so that you can use the language in other platforms too.

`scala-js` is a well-established open-source project that allows you to compile Scala code to JavaScript. It is a framework to build robust applications that run in your browser. It enables you to share code between the front-end and the back-end of your application. Visit <https://www.scala-js.org> for more information.

`scala-native` (see <http://www.scala-native.org>) is another project targeting Scala for the Low-Level Virtual Machine (LLVM) compiler. Its development is in progress, but looking promising: running Scala programs on the LLVM would optimize their speed and memory.

Finally, if you are looking for an alternative – more fun? – way of writing CSS code, have a look at `scalacss` (see <https://github.com/japgolly/scalacss>) for a “super type-safe CSS” in Scala.

1.3 Scala's key features

Since 2004, Scala has evolved a lot, but its fundamental features have not changed. In this section, you'll glance at a few fragments of Scala code: you'll learn about each of these topics in this book.

SCALA IS OBJECT-ORIENTED

An Object-Oriented programming language has a structure based on classes and subclasses. They have well-defined behaviors, and they exchange information through methods. Listing 1.1 provides an example of how to define a class:

Listing 1.1: MyClass example

```
// our first Scala example ①
class MyClass(name: String) { ②
  def sayHello(otherName: String) =
    s"Hi $otherName, my name is $name!" ③
}
```

① This is a comment

② This is a class with a parameter "name" and a method "sayHello"

③ Example of string interpolation. `s"..."` is the operator for string interpolation: it replaces `$otherName` and `$name` with their value.

In unit 1, you'll learn how to define classes and subclasses. In unit 4, you'll also discover "case classes" to present data in an immutable manner.

Singleton instances are instances that you should initialize at most once. Scala offers a dedicated syntax for them: you usually refer to them as "objects". Do not be confused by the term "object" as it can refer to an instance rather than a singleton in other languages. In unit 2, you'll learn about singleton objects in Scala and how not to be confused by the terminology. Listing 1.2 shows you an example of a singleton – or "object" – in Scala:

Listing 1.2: MySingleton example

```
object MyObject { ①
  val a = new MyClass("Scala") ②
  println(a.sayHello("Martin"))
}
```

① This is a singleton – or object

② This is an instance of the class MyClass

Exceptions are another typical feature of an object-oriented language. They represent code anomalies: you can throw and catch them to control your program's execution flow. In Scala, exceptions are similar to those in languages such as Java, Python, and C++. Listing 1.3 provides an example of how to throw and catch exceptions in Scala:

Listing 1.3: Example of throwing and catching exceptions

```
try {
  throw new IllegalStateException("ERROR!") ①
} catch {
  case ex: RuntimeException => ②
    println("Something went bad...")
}
```

① Example of throwing a `java.lang.IllegalStateException`

② Example of catching and handling any `RuntimeException`. Note that an `IllegalStateException` is a subclass of `RuntimeException`.

You'll learn about try-catch expressions in detail in unit 3, in which you are also going to discover partial functions. In unit 6, you'll master how to handle errors without throwing exceptions but using a more functional and safe approach that uses types.

Mutable data structures and assignments are also part of the language. The language's design discourages their use, but you can still take advantage of them when needed. In lesson 4, you'll

learn about the difference between a “value” and a “variable”. In units 5 and 6, you’ll also discover what the Scala Collections have to offer. Have a look at listing 1.4 for an example of how to use mutable assignments in Scala:

Listing 1.4: Example of mutable assignments

```
var a = "hello"
println(a) // prints hello
a = "Scala"
println(a) // prints Scala
```

SCALA IS FUNCTIONAL

Functional Programming languages base their entire structure on functions. Functions play a big part in Scala as its “first-class citizens”. In lesson 6, you’ll learn their basics and how to define them. In unit 4, you’ll discover functional purity and its several advantages and how you can use functions as parameters or return values: you refer to them as higher order functions. They allow you to create powerful abstractions that simplify and remove code duplication in a way that is usually not so easily achievable in an object-oriented approach. Listing 1.5 shows an example of a function that takes another function as its parameter and returns a new function.

Listing 1.5: Example of Higher Order Function

```
def divideByTwo(n: Int): Int = n / 2 ①
def addOne(f: Int => Int): Int => Int = ②
  f andThen(_ + 1)
def divideByTwoAndAddOne = addOne(divideByTwo) ③
```

① A function that takes an Int and returns an Int

② A function that takes a function from Int to Int as a parameter and that returns a new function from Int to Int

③ A function defined by compositing two existing functions

SCALA HAS A ROBUST TYPE SYSTEM

Scala is a statically typed language. Types define the acceptable range of values for your computation. Thanks to them, the compiler can check at compile time that your code doesn’t violate any constraints, which makes your code more reliable and less prone to errors at runtime. Scala has a type inference system, so you do not have to specify the intended type for every expression of your program, making your code less verbose. Languages that do not have a type inference system require you to provide types explicitly and tend to be more verbose. Scala’s type system is also quite flexible: you can reuse existing or create custom types to ensure business requirements at the type level. Starting from unit 4, you’ll discover the most common types the language has to offer. In unit 7, you’ll see how to use type classes and define custom types to enforce requirements at compile time.

SCALA’S INTEGRATION WITH JAVA

Scala is a JVM language, and its compiler is built on top of `javac`, the compiler for Java. You can easily integrate Java code into your Scala programs. Most IDEs that provide Scala support, such as

IntelliJ IDEA, can translate Java code into Scala code automatically: listing 1.6 shows how a snippet of Java code is automatically converted into Scala code by IntelliJ IDEA.

Listing 1.6: Java to Scala code example

```
// file Snippet.java
String dateAsString = "22.11.2017";
SimpleDateFormat format = new SimpleDateFormat( "dd.MM.yyyy" );
Date date = format.parse(dateAsString);

// file Snippet.scala
import java.text.SimpleDateFormat
import java.util.Date

val dateAsString = "22.11.2017"
val format = new SimpleDateFormat("dd.MM.yyyy")
val date: Date = format.parse(dateAsString)
```

Alternatively, you can also add files with extension `*.java` directly into your Scala project: `scalac` will recognize them as Java and compile them accordingly.

1.4 Summary

In this lesson, my objective was to give you an overview of the Scala language.

- You have learned about its unique features and why it is a great language to learn.
- You have discovered the typical execution flow of a Scala program and looked at Scala code snippets.

2

The Scala Environment

After reading this lesson, you'll be able to:

- Execute commands on the Scala REPL.
- Use the REPL to evaluate expressions.
- Install the `git` and `docker` development tools.

The Scala REPL (Read-Eval-Print-Loop) is a development tool to interpret Scala code snippets. It will be a crucial tool in learning Scala, and it doesn't require too much setup or infrastructure: you'll be able to play and experiment with the language by typing and evaluating fragments of code. You'll also install the `git` and `docker` development tools on your machine, which are not specific to Scala but used in some of the more advanced capstones. In the next lesson, you'll complete your machine's setup by installing `sbt` – a tool to build and run structured Scala programs.

2.1 The REPL Installation

In this section, you'll learn how to install the Scala REPL on Linux, macOS, and Windows using a package manager. Alternatively, you can also download Scala binaries from its official website: visit <https://scala-lang.org/download> and its "Download the Scala binaries" section for instructions on how to do this.

First, you need to check that you have installed Java 8+ JDK. Open the terminal and type the command `java -version` to check your java version. You should see a message similar to the following:

```
$ java -version
openjdk version "15.0.1" 2020-10-20
OpenJDK Runtime Environment (build 15.0.1+9-18)
OpenJDK 64-Bit Server VM (build 15.0.1+9-18, mixed mode, sharing)
```

If you need to install or upgrade your JDK, you can find instructions on how to do so on the OpenJDK's website at <https://openjdk.java.net/install>.

You are now ready to download and install the Scala REPL: depending on your operating system, you will have different instructions to follow.

LINUX

On Ubuntu and other Debian-based distribution, type the following command in the terminal:

```
$ sudo apt-get install scala
```

If you are using an RPM-based distribution, such as Red Hat Enterprise, type the following instruction:

```
$ sudo yum install scala
```

Use `emerge` package manager if you are on Gentoo:

```
$ emerge dev-lang/scala
```

MACOS

On macOS, you can install Scala using Homebrew by executing the following command in your terminal:

```
$ brew install scala
```

If you prefer, you can also use MacPorts as follows:

```
$ port install scala
```

WINDOWS

Visit <https://scala-lang.org/download> to download the MSI installer for Scala and open the file to complete the installation.

Once you have completed the installation, you should be able to execute the command `scala` and receive a message similar to the following:

```
$ scala
Welcome to Scala 2.13.4 (OpenJDK 64-Bit Server VM, Java 15.0.1).
Type in expressions for evaluation. Or try :help.

scala>
```

The symbol `scala>` indicates that the Scala REPL is running and ready to receive your commands.

2.2 The REPL Commands

A REPL (Read-Eval-Print-Loop) tool can receive instructions and interpret snippets of code. Let's see what the Scala REPL's main commands.

The Scala REPL identifies commands by the prefix ":". It considers any other instruction as Scala code snippets to compile and evaluate. Lots of commands are available, but you usually remember just a few of them. A list of the most useful commands are the following:

- `:quit` gracefully exits the interpreter:

```
scala> :quit
```

- `:help` lists all the commands available with a brief description. Use it to discover new commands or to check the syntax of an existing one:

```
scala> :help
All commands can be abbreviated, e.g., :he instead of :help.
:help [command]          print this summary or command-specific help
:completions <string>    output completions for the given string
// truncated as the list is quite long!
```

- `:reset` forgets any snippet evaluated so far. In other words, it restores the REPL to its initial state:

```
scala> :reset
Resetting interpreter state.
Forgetting this session history:

val a = "hello world"

Forgetting all expression results and named terms: a
```

- `:replay` resets the REPL and executes all the commands in the session history:

```
scala> :replay
Replaying: val a = "hello world"
a: String = hello world
```

- `:load <path>` interprets snippets of code from a file. Suppose you have started the REPL from a directory containing a file called `Test.scala` and with text `println(1 + 2)`. You can interpret the instructions in the `Test.scala` file as follows:

```
scala> :load Test.scala
Loading Test.scala...
3
```

Include the relative and absolute path of your file to load documents from other folders of your machine.

After reviewing the main REPL commands, let's see how you can use it to interpret code fragments.

2.3 The REPL Code Evaluation

You are now ready to interpret Scala expressions in the REPL. Let's start by evaluating the arithmetic expression `1 + 2`:

```
scala> 1 + 2
res0: Int = 3
```

Figure 2.1 analyzes the structure of your expression and its interpretation. It sums the integers 1 and 2 using the operator `+`, resulting in the value 3 of type `Int`. The REPL automatically saves the result as a value with a name having prefix `res` followed by an incremental number: in this case, it's `res0`.

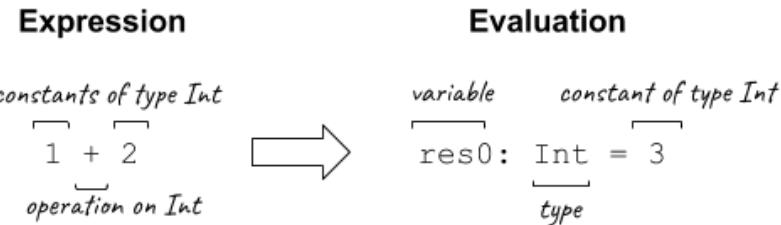


Figure 2.1: Analysis of the REPL's evaluation of an arithmetic expression. The interpretation of the sum of two integers produces a constant of type `Int`. The REPL saves the result in a value called `res0`.

You can use the result value `res0` in your following computations. For example, you can do:

```
scala> res0 + 39
res1: Int = 42
```

You can also request the result to be saved with a specific name:

```
scala> val x = 1 + 2
x: Int = 3

scala> x
x: Int = 3
```

Note that you have not specified any type explicitly in all the examples you have seen so far: thanks to its type inference, the Scala compiler infers that summing two integers returns an integer. When needed, you can provide an explicit return type: if not compatible with the expression provided, the compiler will display an error.

```
scala> val x: Int = 1 + 2
x: Int = 3

scala> val y: Double = x
y: Double = 3.0
// the compiler knows how to automatically convert Int to Double!

scala> val z: String = x
<console>:11: error: type mismatch;
 found   : Int
 required: String
      val z: String = x
                  ^
```

The compiler knows how to convert an `Int` into a `Double`, but it cannot do the same when transforming an `Int` into a `String`.

You can define and call functions using the REPL. For example, you can implement a function `sayHello` that takes one parameter `n` of type `String`, and it returns a `String`. Have a look at figure 2.2 for a diagram of how the REPL interprets a function.

```
scala> def sayHello(n: String) = s"Hi $n!"
sayHello: (n: String)String
```

```
scala> sayHello("Scala")
res1: String = Hi Scala!
```

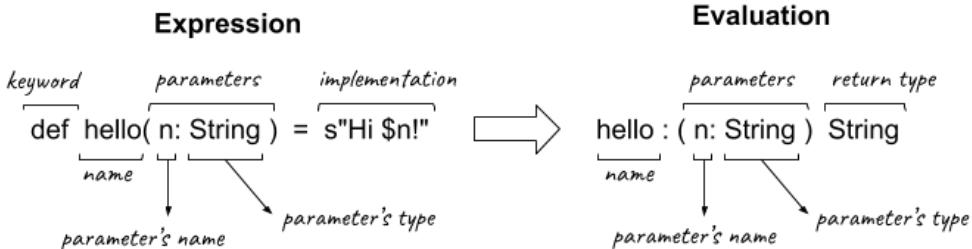


Figure 2.2: Analysis of the REPL's interpretation of a function definition. The REPL identifies the function by its name, parameter, and return type.

In the previous example, you don't have to specify the return type for the `sayHello` function: the compiler can correctly infer that its type to be `String`. When needed, you can specify the desired return type of a function, and the compiler will see if this is compatible with its implementation:

```
scala> def sayHello(n: String): String = s"hi $n!"
sayHello: (n: String)String

scala> def sayHello(n: String): Int = s"hi $n!"
<console>:10: error: type mismatch;
 found   : String
 required: Int
           def sayHello(n: String): Int = s"hi $n!
                                         ^"
```

Let's see how you can define a class and create an instance using the REPL:

```
scala> class MyClass(n: String)
defined class MyClass

scala> new MyClass("Martin")
res2: MyClass = MyClass@32cf48b7
```

Finally, you can implement and use singletons – or “objects” as follows:

```
scala> object MyObject
defined object MyObject

scala> MyObject
res6: MyObject.type = MyObject$@176d53b2
```

Scala Worksheet

Most IDEs with dedicated support for Scala, such as IntelliJ IDEA, often also support Scala Worksheets. These are files with extension `.sc` that can contain snippets of Scala code. Your IDE will start a REPL session to evaluate each snippet of code within the file and display its results within the text editor itself. Scala Worksheets are a valid alternative to manually starting a Scala REPL session from the command line.

2.4 Other Development Tools

To complete the environment setup, you'll also need to install other development tools: `sbt`, `git`, and `docker`. The next lesson will go into the details of how to install and use `sbt`. Let's see how to set `git` and `docker` up in this section.

2.4.1 Git Installation

Git is a commonly used open-source version control system, and is often pre-installed in Unix machines. You'll use it to download existing code and give you a head start in the more advanced lessons. Run the following command in the terminal to check if `git` is already installed on your machine:

```
$ git --version
git version 2.21.1
```

If you need to install `git`, please follow the instructions for your operating system at <https://github.com/git-guides/install-git>.

2.4.2 Docker Installation

Docker is a platform that helps you manage the complexity of your infrastructure using virtual machines. You'll use `docker` to run a temporary PostgreSQL database instance to use in your Scala applications. Follow the instructions on how to install Docker at <https://docs.docker.com/get-docker>. After its installation, you should be able to execute the `docker --version` command in your terminal and get an output similar to the following:

```
$ docker --version
Docker version 20.10.0, build 7287ab3
```

2.5 Summary

In this lesson, my objective was to show you how to use the Scala REPL.

- You have learned its most common commands.
- You have also written your first snippets of Scala code, and you have seen their evaluated values in the REPL.
- You have installed the `git` and `docker` development tools on your machine: you'll use these tools in the more advanced lessons in the book.

Let's see if you got this!

TRY THIS

Perform the following operations in the REPL:

1. Start the REPL and evaluate the following expression: "na" * 3 + 42.
2. Restore the REPL to its initial state using the appropriate REPL command.
3. Quit the REPL gracefully with the :quit command.

3

sbt – Scala Build Tool

After reading this lesson, you'll be able to:

- Run commands in sbt
- Create an sbt project
- Describe the structure of the files of a Scala project built with sbt

In the previous lesson, you have learned how to evaluate snippets of code using the Scala REPL. Programs have several components: they depend on other modules and libraries, and they are structured in multiple files and folders. In this lesson, you'll discover the basics of the Scala Build Tool, called `sbt`. You'll compile and run your first Scala program on the JVM using `sbt`. In this book, you'll use `sbt` to manage dependencies, compile your code, and run your Scala programs.

3.1 Why sbt?

You can build Scala projects using several build tools such as Maven, Ant, and Gradle, but `sbt` is the most common build tool in the community. `sbt` is a complex but powerful tool that manages your dependencies, and it defines the build cycle of your code. `sbt` has its syntax and predefined instructions. You can also load plugins, which are programs written using the `sbt` syntax, to add new commands or modify the build process's existing behaviors. Throughout this book, you are going to write Scala programs that use the default `sbt` configuration. You are not going to learn about the `sbt` advance uses and functionalities in this book, but you can have a look at its documentation at www.scala-sbt.org if you'd like to learn more about it.

Alternatives to sbt

The Scala community has a particular love-hate relationship with sbt: while some people think it is a great tool to work with, others dislike it for its complexity. Although extremely powerful, sbt can become incredibly complicated to understand and maintain.

sbt is undoubtedly the most popular choice for Scala projects, but not the only one. For example, you could use any build tool with JVM support, such as Maven or Gradle.

Scala also has another community-driven build tool, called `Mill`. It takes most of its design choices from another build tool called `Bazel`, and it aims at a syntax that is simple to use and a fast and predictable build. For more information, have a look at <https://github.com/lihaoyi/mill>.

3.2 sbt Installation

In this section, you'll discover how to install sbt using a package manager using sbt on Linux, macOS, and Windows. You can also install it from its binaries: have a look at www.scala-sbt.org/download for instructions on how to do so.

Before installing sbt, make sure you have Java 8+ JDK installed on your machine. Execute the command `java -version` to check which Java version your machine is running. You should see a message similar to the following:

```
$ java -version
openjdk version "15.0.1" 2020-10-20
OpenJDK Runtime Environment (build 15.0.1+9-18)
OpenJDK 64-Bit Server VM (build 15.0.1+9-18, mixed mode, sharing)
```

If missing, have a look at the previous lesson instructions on installing or upgrading your Java JDK.

Depending on your operating system, you'll need to use a different package manager to install sbt.

LINUX

On Ubuntu and other Debian-based distributions, you can execute the following command in the terminal:

```
$ sudo apt-get install sbt
```

If you are using an RPM-based distribution, such as Red Hat Enterprise, you can use yum:

```
$ sudo yum install sbt
```

If you are on Gentoo, type the following command on the terminal:

```
$ emerge dev-java/sbt
```

MACOS

You can use Homebrew to install sbt by running the following command in your terminal:

```
$ brew install sbt
```

If you prefer, you can also use MacPorts:

```
$ port install sbt
```

WINDOWS

Download the MSI installer for sbt at www.scala-sbt.org/1.x/docs/Installing-sbt-on-Windows. Open the file and follow the instruction on the UI to complete the installation.

After completing the installation, you should be able to open the terminal and execute the command `sbt`. When running sbt for the first time, it will download some dependencies: this could take a while and display lots of text in the console. It will happen only the first time: next time you start sbt, it will be a lot faster. Eventually, sbt will be up and ready to receive commands. Your terminal should look similar to the following:

```
$ sbt
[info] Loading project definition from /my/path
// ...
// Lots of output omitted here!
// ...
[info] sbt server started at local://some/path
sbt>
```

Congratulations! sbt is now running and ready to accept commands.

3.3 sbt Commands

sbt operates in the folder in which you typed the command `sbt`: make sure to be in the right folder! On startup, sbt looks for a specific folder structure – more on this in section 3.5 – to tentatively load your project and start its server. Once done, it will wait for commands to execute. The most common and useful ones are the following:

- `exit` gracefully terminates your session.
- `about` provides general information on the sbt version that you are running.
- `compile` triggers the compilation of your code.
- `run` compiles and runs your program on the JVM.
- `help` lists each sbt command followed by a brief description of their usage.
- `clean` deletes any file produced during the compilation.
- `reload` re-evaluates the configurations provided in the project.
- `new` creates an sbt project – you'll see it in action in the next section.
- `console` starts the REPL within an sbt project: all your code and dependencies are now accessible from the REPL for you to experiment with them.

Now that you have installed sbt, you are now ready to build your first sbt project.

3.4 Your first sbt Project

sbt has dedicated support to Giter8 templates: you can use them to create a skeleton for your Scala project.

What is Giter8?

Giter8 is a project by Foundweekends (see <http://www.foundweekends.org>) that allows you to host templates on GitHub and apply them to generate skeleton projects: sbt is one of the supported ones. It is becoming more and more popular in the community as one of the quickest ways to get started with many popular Scala libraries.

Visit <https://github.com/foundweekends/giter8/wiki/giter8-templates> for a list of some of the templates available. Spark, Spark Job Server, Akka, Akka-HTTP, Play, Lagoom, Scala Native, http4s are just a few projects that have published an official giter8 template. Have a look at <http://www.foundweekends.org/giter8> for more information on Giter8.

Your first application prints “Hello, World!” in the console. Let’s create it by applying a Giter8 template called `scala/hello-world.g8`, using the following sbt command:

```
$ sbt
sbt> new scala/hello-world.g8

A template to demonstrate a minimal Scala application

name [Hello World template]: //press enter to use the default name

Template applied in ./hello-world-template

[info] shutting down server
```

This command has created a new directory, called `hello-world-template`, containing the code generated by the template. You can access, compile, and run the code as follows:

```
$ cd hello-world-template
$ sbt
sbt:hello-world> compile
// ...
// output omitted here!
// ...
[info] Compilation completed in 13.992s.
[success] Total time: 16 s, completed 8 Jan 2021, 18:27:20

sbt:hello-world> run
[info] running Main
Hello, World!
[success] Total time: 1 s, completed 8 Jan 2021, 18:34:20

sbt:hello-world>
```

Eureka! Congratulations on compiling and running your first Scala program using sbt.

Scala 3: Getting Started

At the time of this writing, Scala 3 has not been released yet. However, you can access its latest release candidate by running the applying the following Giter8 template:

```
sbt new scala/scala3.g8
```

This command will create an empty sbt project that uses Scala 3.0.0-RC1. You can also access the Scala 3 REPL from it by executing the command `sbt console`.

Which IDE to use?

Having the right tool to write code can make you extremely productive. The choice of using or not using an IDE is pretty subjective and often dictated by personal preferences.

Most people in the community use IntelliJ IDEA (see <https://www.jetbrains.com/idea>). It has dedicated support for Scala: it offers syntax highlighting, code completion, integrated REPL, jump to the source code, debugger, worksheet support, and more. It also has an sbt integration to help you build projects that respect the standard sbt project structure.

IntelliJ IDEA is full of useful features, but it can be demanding on the CPU and slow at times. If you prefer to use a text editor, Metals (<https://scalameta.org/metals/>) is an excellent alternative. It adds IDE-like features to text editors such as Visual Studio Code, Atom, Vim, Emacs, and Sublime. It has many of the functionalities of an IDE, but it is faster and lighter to run.

3.5 sbt Project Structure

sbt Scala projects have a standard structure. Let's see how you should organize your program by looking at the skeleton project you have created using the hello-world giter8 template in the previous section.

THE HELLO-WORLD-TEMPLATE FOLDER

By applying the template, you have created a new folder called `hello-world-template`: figure 3.1 shows its content. The `build.sbt` contains everything that sbt needs to know to compile and run your code: its Scala version, your application name, organization and version, and external libraries.



Figure 3.1: Files and directories in the `hello-world-template` folder. The `build.sbt` file contains the instructions on how to build your Scala program. The `project` folder sbt configurations and settings, while `src`

contains all your source code. sbt creates the `target` folder after you compile your code: it includes the JVM bytecode of your application.

Have a look at listing 3.1 for the settings used in your hello world application:

Listing 3.1: build.sbt file

```
// Lots of comments to give you an overview
// of the most common configurations
// sbt has to offer: have a look at them!

scalaVersion := "2.13.3" ②

name := "hello-world"
organization := "ch.epfl.scala"
version := "1.0"
```

② mandatory settings

In this file, you can add external library dependencies you'd like to use in your code. For example, you can add a dependency to `cats`, a popular library for functional programming, by adding the following line to your `build.sbt` file:

```
libraryDependencies += "org.typelevel" %% "cats" % "2.2.0"
```

sbt identifies external dependencies by their organization (i.e., "org.typelevel"), their name (i.e., "cats"), and version (i.e., "2.2.0").

The `project` directory contains everything that sbt needs to run: its configurations and code – more on this later in this section.

The `src` folder contains the source code for your project: you'll see how production and test code are separated.

Finally, sbt generates the `target` folder after compiling your code: it contains its bytecode together with external dependencies and resources so that you can run it on the JVM.

THE PROJECT FOLDER

Have a look at figure 3.2 for the content of the `project` folder in the root directory: it contains everything that sbt needs to run.

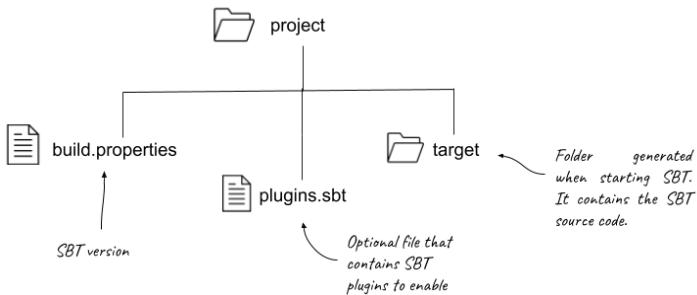


Figure 3.2: The project folder contains the sbt version to use (see `build.properties` file), the source code for sbt in the target folder, and any sbt plugin that needs to download and enable.

The `build.properties` file specifies the desired sbt version: this looks similar to the one shown in listing 3.2.

Listing 3.2: project/build.properties file

```
sbt.version=1.4.6
```

The hello-world giter8 template doesn't use any sbt plugin, but if you wish to do so, you can add custom functionalities to sbt by creating a file `plugins.sbt` in the `project` directory. Listing 3.3 demonstrates how to enable the sbt plugin `sbt-scoverage`: it measures and produces reports on your test coverage by running a custom command called `coverage`.

Listing 3.4: Example of plugins.sbt file

```
addSbtPlugin("org.scoverage" % "sbt-scoverage" % "1.6.1")
```

The `project/target` folder contains all the information that sbt needs to run. sbt parses it, download its source code of the expected version (see `build.properties`) and plugins (see `plugins.sbt`). The first time you launch sbt, it is going to take a few minutes to start up. After that, sbt reuses the already downloaded sources, and it will be much faster.

THE SRC DIRECTORY

Figure 3.3 shows the `src` directory's content in the root folder: it contains all the source code for your project.

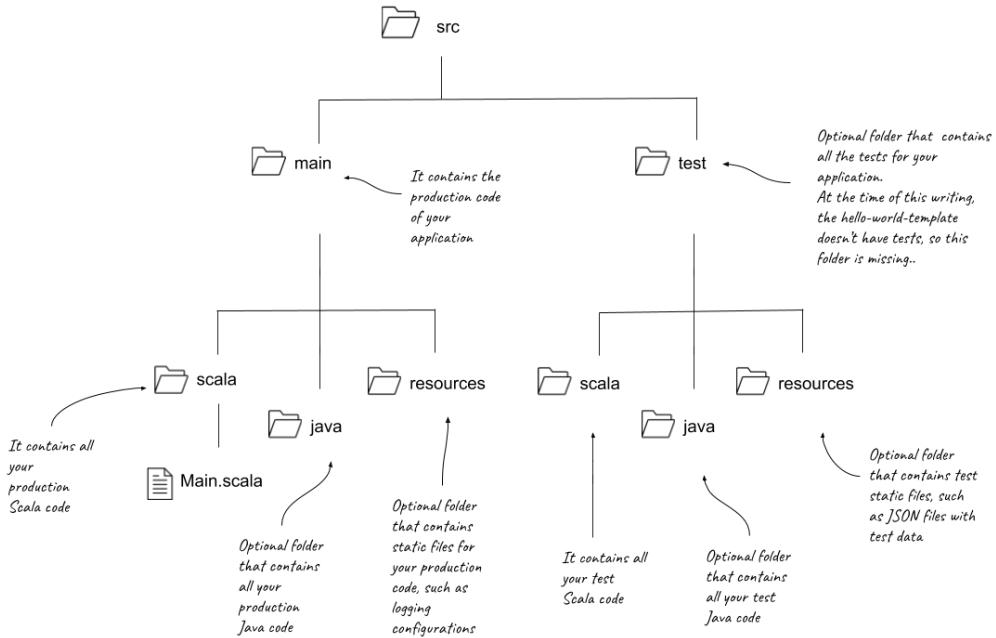


Figure 3.3: The `src` folder's content: the `main` directory contains all your production code, while the `test` one contains all your tests to verify your application's correctness. The `main` and `test` folders have the same structure: the `scala` and `java` folders include all the Scala and Java code, respectively. The `resources` one contains all the static files needed, such as logging and application configurations.

While the `src/test` one contains code to prove your program's correctness, the `src/main` folder includes your actual application. Note that the `hello-world giter8` template doesn't have tests, the reason why sbt will not create the `test` folder. Both `main` and `test` have three optional folders: `scala` to store all your `*.scala` files, `java` for any Java source code, and `resources` for any static ones. In `src/main/resources` you usually collect application and logging configurations files, while in `src/test/resources` is generally reserved for test data, such as JSON and XML files, to load and parse in your tests.

Let's have a look at the only `*.scala` that the `hello-world giter8` template has produced: the `Main.scala` file in the `src/main/scala` directory:

Listing 3.4: Main.scala

```
object Main extends App { ①
    println("Hello, World!") ②
}
```

- ① **Main** is a singleton object: you can instantiate it only once. It is also an `App`: the JVM executes it as the first instruction when your program starts.
- ② It prints "Hello, World!" to the console.

3 It prints “Hello, World!” to the console.

Some code elements will be new to you, but do not worry: you’ll learn about each component of this hello-world program later on in the book.

Have a look at figure 3.4 for a summary of the entire structure of an sbt project.

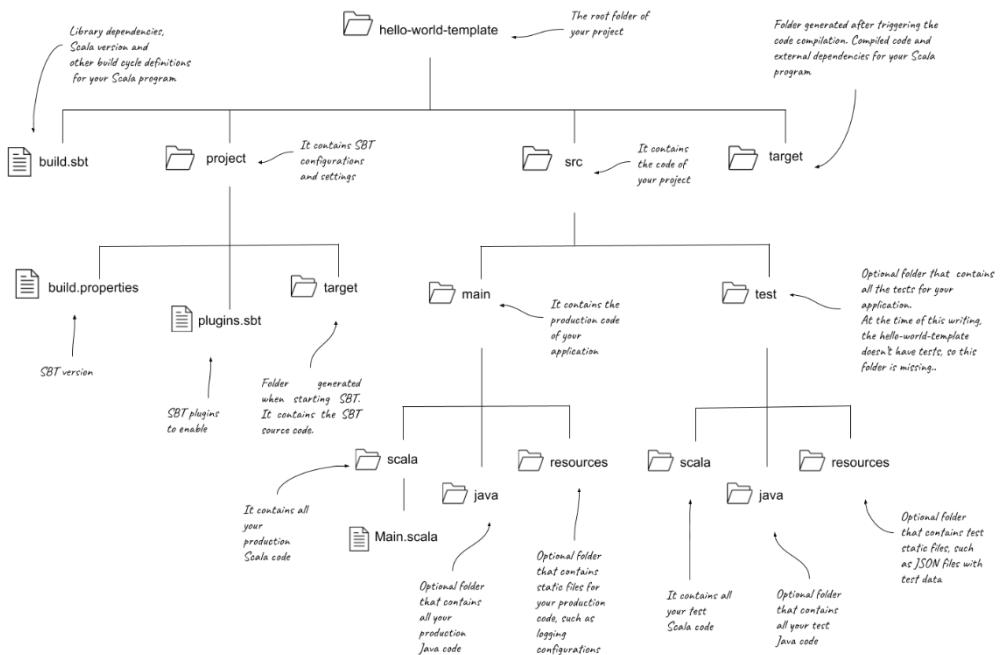


Figure 3.4: The structure of a typical Scala application built with sbt. While the `build.sbt` file and the `project` directory are specific to sbt, the `src` and `target` folders contain your application source code and bytecode, respectively.

3.6 Summary

In this lesson, my objective was to introduce you to sbt – the Scala Build Tool.

- You have installed sbt and discovered its main features.
- You have created, compiled, and run your first sbt projects.
- You have also learned about how sbt expects you to organize your files and folders.

Let’s see if you got this!

TRY THIS:

1. Modify `Main.scala` in the `hello-world-template` project by removing the following instruction: `extends App`. What is it going to happen? Validate your assumptions by running the modified code.

HINT: You can use your IDE's "jump to source" functionality to peek at the source code for App.

2. Change your hello-world-template to print your name in the console: compile and run it using sbt.

Unit 1

The Basics

In Unit 0, you have installed everything you need to start coding in Scala. In this unit, you'll begin writing Scala code. You'll learn how to program a vending machine in Scala using the same OOP principles already familiar to you. In particular, we will discuss the following subjects:

- Lesson 4 illustrates the difference between values and variables and the clear separation between mutable and immutable assignments.
- Lesson 5 introduces conditional constructs to control the execution flow of our program. You'll learn about loops to repeat the same operation multiple times.
- Lesson 6 shows you how to define functions in Scala.
- Lesson 7 demonstrates how to use classes to represent interactions between elements. You'll learn about abstract classes and how to express a class-subclass relation.
- Finally, you'll apply all the principles learned so far to code a vending machine that sells chocolate and granola bars to your customers in lesson 8.

Once comfortable with the basics, you'll continue with Unit 2, where you'll learn more about object-oriented fundamentals.

4

Values and Variables

After reading this lesson, you'll be able to:

- Write code that uses values to save and reuse computational results
- Implement programs that carefully take advantage of variables

Scala has a clear separation between mutable and immutable assignments. In Scala, “values” are immutable: you cannot modify them after creating them. “Variables” are mutable: they can refer to different instances over time. Deciding when to declare a value rather than a variable is essential for your code to be fast and bug-free. Variables are more straightforward to use in your code because you can modify them. However, they can make your program extremely difficult to maintain and lead to errors when different processes try to do so simultaneously. Values can be challenging to use because you cannot modify them once created. But they can make your program easier to understand: their assignments never change, so you can easily predict and test their evaluation. They also guarantee that your code will not be affected by concurrency issues, such as data inconsistencies, resources starvation, and deadlocks, when accessed by several threads. A fragment of code that multiple processes can access without causing concurrency issues is “thread-safe”. In the capstone for this unit, you’ll define both values and variables to name fragments of code and make your program more readable.

Consider this

Consider the other languages you have encountered so far in your career. Do they make a clear distinction between mutable and immutable assignments? If so, how? If not, can you think of the advantages and disadvantages of not having a clear separation between the two?

4.1 Values

Assignments improve your code's readability by breaking down the computation and by assigning meaningful names to it. They can also increase your program's performance by saving intermediate results that you can reuse and share between processes. Values are the most commonly used type of assignments in Scala.

In the next example, you'll see how values can improve your code's readability, particularly when dealing with complex scenarios. Suppose you are performing some calculation to mark exams with the following requirements: each exam has three questions – each of them scoring up to 3 points – and its final mark must be between 1 and 10. Several solutions are possible: let's pick one of them. You could calculate the average score for all the questions and then convert it to a number from 1 to 10. For example, you should follow these steps for three questions scoring 1.5, 2, and 2.5:

1. You compute the average score for all the questions: $(1.5 + 2 + 2.5) / 3$ is 2
2. You scale the average score from 1 to 10. Considering that the maximum score possible for each question is 3, then you can rescale it as follows: $2 * 10 / 3$ is about 6.6
3. You round 6.6 up to the closest integer and obtain 7.

Let's translate this procedure into code. For example, you could implement a function like the following:

Listing 4.1: Marking an Exam

```
def markExam(q1: Double, q2: Double, q3: Double) =  
    Math.round(((q1 + q2 + q3) / 3) * 10 / 3)
```

The function `markExam` replicates the calculations described above, but it is difficult to identify its objective and computation steps. You can improve the readability of the `markExam` function by using values with descriptive names:

Listing 4.2: Marking an Exam – a more readable version

```
def markExam(q1: Double, q2: Double, q3: Double) = {  
    val avgScore = (q1 + q2 + q3) / 3  
    val scaledScore = avgScore * 10 / 3  
    Math.round(scaledScore) ①  
}
```

① no need for the `return` keyword: the last expression is the one returned.

Values are a fundamental component of your code. In Scala, you declare them by using the keyword `val`. You can only initialize them once because they are immutable: if you try to reassign them, the code will not compile. Consider the following example:

```
scala> val a = 1
a: Int = 1

scala> a
res0: Int = 1

scala> a = 2
<console>:11: error: reassignment to val ①
```

① You cannot reassign a value, which is an immutable assignment

QUICK CHECK 4.1:

Can you use the `+=` operator with a value of type `Int`? What about the operator `--`?

When declaring a value, its type is optional: the compiler tries to infer it for you. If the compiler infers an unexpected type, you can provide it explicitly: the compiler will check if the type you specified is compatible with the assignment of your value and eventually return a compilation error:

```
scala> val b: Int = 1
      b: Int = 1

scala> val c: Double = 1 ①
      c: Double = 1.0

scala> val d: Int = "not-an-int" ②
<console>:10: error: type mismatch;
 found   : String("not-an-int")
 required: Int

scala> val d: Any = "not-an-int" ③
      e: Any = not-an-int
```

① The compiler knows how to fit an `Int` into a `Double`

② You cannot assign a `String` to an `Int`

③ `Any` is the root of the class hierarchy: you can assign any value to it.

Have a look at figure 4.1 for a syntax diagram on values in Scala.

Value - Immutable Assignment

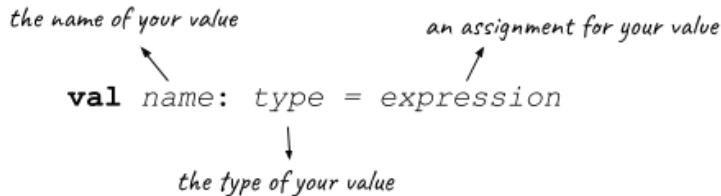


Figure 4.1: Syntax diagram of how to define a value in Scala. A value is immutable, so you can only assign it once.

QUICK CHECK 4.2:

Which of these expressions are valid? Use the REPL to validate your hypotheses.

1. val test: Double = 2.0
2. val test: Int = 2.0
3. val test: Any = 2.0

4.2 Variables

In the previous section, you have written a function to mark an exam. Let's expand it so that you can compute the average score of all exams you marked so far. Values are immutable, so you cannot use them to keep track of how your stats evolve, which is the "mutable state" of your program. In these cases, you can use "variables" as the following:

Listing 4.3: Keeping track of the mark statistics

```
var marksSum = 0 ①
var marksCount = 0 ①

def averageMark: Double =
    marksSum.toDouble / marksCount ②

def markExam(q1: Double, q2: Double, q3: Double): Int = {
    val avgScore = (q1 + q2 + q3) / 3
    val scaledScore = avgScore * 10 / 3
    val mark = Math.round(scaledScore).toInt ③

    marksSum += mark
    marksCount += 1
    mark ④
}
```

① example of a variable

② Converting marksSum to Double: the result of your division must be a Double. An integer divided by another integer produces an Int while diving a double by an integer returns a double.

③ Math.round results in a constant of type Long, so you need to convert it to Int.

④ No need for the return keyword: the last expression is the one returned.

In Scala, variables are very similar to values: while you can assign values only once, you can reassigned variables multiple times. Consider the following example:

```
scala> var a = 1
a: Int = 1

scala> a
res1: Int = 1

scala> a = 2
a: Int = 2
```

You do not need to provide a type for your variable: the compiler will infer it for you. When doing so, the compiler makes sure that the assignment is compatible with it. Have a look at figure 4.2 for a summary of how to define variables in Scala.

Variable - Mutable Assignment

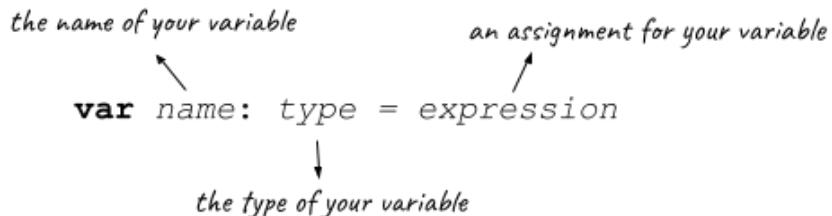


Figure 4.2: Syntax diagram of how to define a variable in Scala. A variable is a mutable assignment: you can reassign it more than once.

Think in Scala: val or var?

When in doubt, you should always try to use `val` instead of `var`.

Having immutable values makes your programs safe from concurrency inconsistency in exchange for some additional memory allocation; most of the time, you will not notice this extra memory usage thanks to the JVM's garbage collection.

Using mutable structures makes your programs challenging to test and reason about: you need to pay attention to how processes share data and how it evolves. Mutable structures may be more performant, but they also expose your code to complex concurrency issues such as data inconsistency, deadlocks, and resource starvation.

Before choosing to stick with a `var`, you should make sure to have tried everything else in your book of tricks: variables are always the last and least desirable option.

At this point in your Scala journey, you may think that using mutability is often the only option. But this is not the case: in future units and lessons, you'll discover many tools and techniques, such as case classes, that will allow you to get rid of `var` from your code without too much effort.

QUICK CHECK 4.3:

Can you use the `+=` operator with a `var` of type `String`? What about the operator `-=`?

4.3 Summary

In this lesson, my objective was to teach about the different types of assignments in Scala.

- You have learned how to declare values and that you can only assign them once.
- You have seen how to define variables and how you can reassign them multiple times.

Let's see if you got this!

TRY THIS

Modify your `markExam` function to keep track of both the lowest and the highest mark computed so far.

4.4 Answers to Quick Checks

QUICK CHECK 4.1:

The operators `+=` and `-=` cannot be used with `val`: a value is immutable and, once assigned, you cannot reassign it.

QUICK CHECK 4.2:

The first and third assignments are valid, while the second one is invalid:

4. Valid: an expression of type `Double` is compatible with `Double`
5. Invalid: the compiler doesn't know how to fit an expression of type `Double` into an `Int`. Even if the quantity `2.0` is equal to `2` from a mathematical perspective, the compiler doesn't automatically convert a `Double` to `Int` to avoid precision loss: `2.5` is not equivalent to neither `2` or `3`. Note that the opposite (i.e., assigning an `Int` expression to a `Double`) is valid.
6. Valid: `Any` is compatible with any type because it is at the root of Scala's class hierarchy.

QUICK CHECK 4.3:

You can use the operator `+=` together with a `var` of type `String`: you can reassign a variable, and the class `String` has an implementation for the function `+`. You cannot use the operator `-=` with a `var` of type `String` because the class `String` doesn't have a method called `"-"`.

5

Conditional Constructs and Loops

After reading this lesson, you'll be able to:

- Control the execution of your code using an if-else construct.
- Iterate through instructions using while and for loops

In lesson 4, you have learned how to define values and variables to store some computation results. But life is not always so linear: when performing a task, you may also need to make informed decisions by selecting one approach rather than the other. In this lesson, you'll discover how to combine different executions of your code through given conditions. You'll learn how to use the most common conditional constructs and loops in Scala. You'll use if-else constructs to check if the system should allow a requested operation in the capstone.

Consider this

In the previous lesson, you have written a program to mark exams from one to ten. Suppose you need to label a mark as follows: "Badly Failed" if the score is lower than four, "Failed" if between four and six excluded, "Passed" if between six and nine excluded, "Passed with Honors" otherwise. How would you implement this new functionality?

5.1 If-else construct

Suppose you have a list of events: each one has a number representing the day of the week it occurs, starting with one for Sunday and ending with seven for Saturday. You want to write a function that takes an integer representing a day of the week and returns either "weekday" or "weekend" accordingly. For example, you could come up with a solution similar to the following:

Listing 5.1: Categorize the day of the week

```
def categorizeDayOfWeek(n: Int) = {
    if (n == 1 || n == 7) {
        "weekend"
    } else if (n > 1 && n < 7 ) {
        "weekday"
    } else {
        "unknown"
    }
}
```

Listing 5.2 shows you another example of using the if-else construct that shows a function that returns a string to label an integer as positive, negative, or neutral.

Listing 5.2: Labeling a number

```
def label(n: Int) =
    if (n == 0) "neutral"
    else if (n < 0) "negative"
    else "positive"
```

The structure of the if-else construct is similar to other languages: figure 5.1 summarizes its Scala syntax. The keyword `if` is the only one that is mandatory, and it identifies the first condition. You can add more predicates by using the keywords `else if`. Finally, the keyword `else` defines the behavior for the values that don't satisfy any previous criteria.

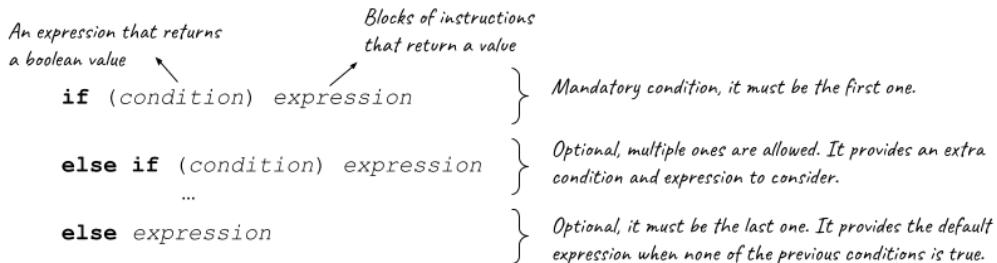


Figure 5.1: Syntax diagram of how to define an `if-else` construct in Scala. The `if` component is mandatory, while the `else if` and `else` ones are optional.

Expressions and Curly Braces: omit or not to omit?

Expressions are everywhere in Scala: they are in values, variables, conditional constructs, functions, and more!

Scala uses curly braces to identify expressions: one or more instructions make an expression. When an expression contains only one instruction, you can omit its curly braces. For example, the expressions `val a = 42` and `val a = { 42 }` are equivalent because they both assign the expression 42 to the value a.

Let's have another look at listing 5.2. You could have implemented the `label` function as follows:

```
def label(n: Int) = {
    if (n == 0) { "neutral" }
    else if (n < 0) { "negative" }
```

```
    else { "positive" }
}
```

You can omit the curly braces for the expressions associated to each if-condition because they consist of a single expression:

```
def label(n: Int) = {
  if (n == 0) "neutral"
  else if (n < 0) "negative"
  else "positive"
}
```

You can also eliminate the curly braces at the function level: the if-else construct is a single expression that identifies the body of the function `label`:

```
def label(n: Int) =
  if (n == 0) "neutral"
  else if (n < 0) "negative"
  else "positive"
```

You should omit curly braces where possible, but without compromising the readability of your code.

QUICK CHECK 5.1:

Write a function called `evenOrOdd` that takes an integer and prints a message to the console to tell if a number is even or odd.

HINT: use the modulo operator `%` for the remainder of a division (e.g., `5 % 2` returns `1`).

```
def evenOrOdd(n: Int)
```

5.2 While Loop

Imagine you'd like to write a function called `echo5` to print a given message on the console five times. You could write a function that has five print instructions:

```
def echo5(msg: String) = {
  println(msg)
  println(msg)
  println(msg)
  println(msg)
  println(msg)
}
```

Your implementation of the function `echo5` doesn't scale: what if you need to print 50 messages rather than just 5? What if their number changes according to the parameters passed to the function?

When you need to repeat an instruction until a particular condition is satisfied, you can use a `while` loop:

5.3: Repeating using a while loop

```
def echo5(msg: String) = {
    var i = 1
    while(i <= 5) {
        println(msg)
        i += 1
    }
}
```

Have a look at figure 5.2 for a syntax diagram of while loops in Scala: in a `while` loop, you repeat its instructions until its condition is no longer valid.



Figure 5.2: Syntax diagram of while loops in Scala: you evaluate the expression until the condition is valid.

Think in Scala: while loops as anti-patterns

`while` loops are part of the Scala language, but they are rarely used and often considered anti-patterns. `while` loops operate on mutable state: they keep track of how it evolves, and they stop once a predicate is respected. Scala discourages the use of mutability, so `while` loops are relatively rare in the code. In unit 5, you'll learn about how the Standard Scala Collections provides powerful alternatives to `while` loops through dedicated functions, such as `foreach` and `map`, to iterate in a functional and thread-safe way.

QUICK CHECK 5.2:

Write a function, called `count`, to print all the numbers from one to ten using a `while` loop.

5.3 For Loop

In the previous section, you have seen how to implement a function called `echo5` to print a given message five times using a `while` loop. In Scala, you can also use a `for` loop to achieve the same result:

5.4: Repeating using a `for` loop

```
def echo5(msg: String) = {
    for (i <- 1 to 5) { ①
        println(msg)
    }
}
```

- ① The expression '1 to 5' returns an iterable data structure called Range containing the numbers 1, 2, 3, 4, and 5 (right end included). You can also use the expression '1 until 6' to generate a range with the right end excluded.

When using a `for` loop, you repeat an operation over a finite set of values rather than using a stop condition. Have a look at figure 5.3 for a summary of `for` loops in Scala.

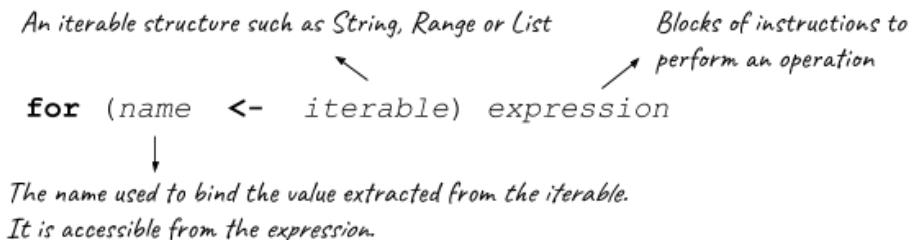


Figure 5.3: Syntax diagram on `for` loops in Scala. In a `for` loop, an expression is repeated for each of a finite set of values.

Scala also offers a more powerful alternative to the classic `for` loop you have just seen, called for-comprehension: you'll learn about it when discussing the class `Option` in unit 4.

QUICK CHECK 5.3:

Does the following expression compile? If so, what is its output?

```
for (a <- "hello") println(a)
```

Use the REPL to validate your hypothesis.

5.4 Summary

In this lesson, my objective was to teach about the basic conditional constructs in Scala.

- You how to use an if-else construct to express decision making in your code.
- You have also learned how to use while and for loops to repeat operations multiple times.

Let's see if you got this!

TRY THIS

Write a function to apply the discount to a given price as follows:

- 0% discount if the price is less than \$50

- 10% discount if the price is at least \$50 but less than \$100
- 15% discount if the price is at least \$100

5.5 Answers to Quick Checks

QUICK CHECK 5.1:

A possible implementation for the function `evenOrOdd` is following:

```
def evenOrOdd(n: Int) =
  if (n % 2 == 0) println(s"$n is even")
  else println(s"$n is odd")
```

QUICK CHECK 5.2:

You could implement the function `count` using a `while` loop as follows:

```
def count() = {
  var i = 1
  while(i <= 10) {
    println(i)
    i += 1
  }
}
```

QUICK CHECK 5.3:

The expression compiles, and it prints each letter of the word “hello” in a new line of the console:

```
scala> for (a <- "hello") println(a)
h
e
l
l
o
```

6

Function as the most fundamental block of code

After reading this lesson, you will be able to:

- Identify the components of a function
- Implement functions in Scala
- Define function parameters with defaults

When coding, functions are the most fundamental block of code. They provide instructions on how to perform a given task: without them, you will not be able to define how your application works. Scala treats functions as its “first-class citizens”: they are an essential component of your program with different uses and shapes. For example, you can use partial functions, anonymous functions, higher-order functions, and more! In this lesson, you’ll review the basics of functions in Scala and analyze their different components. You’ll also see how to define a function. In the capstone for this unit, you’ll use functions to provide instructions on how to operate your vending machine.

Consider this

You can have many different types of functions in Scala: abstract, concrete, anonymous, high order, pure, impure, and many more! I'll gradually introduce you to most of them later on in the book. Consider the programming languages you have encountered so far: what kind of functions do they offer?

6.1 Functions

Suppose you want to determine if a given number is even. To achieve this, you could write the following:

Listing 6.1: The isEven function

```
def isEven(n: Int): Boolean = {
    n % 2 == 0
}
```

Listing 6.1 implements a function called `isEven`. In Scala, a function identifies some computation that takes zero or more parameters, and that may – or may not – return a value. In Scala, a function can throw exceptions or never terminate:

```
def myFunc(n: Int): Boolean = {
    if (n < 0) throw new RuntimeException("Error!")
    n % 2 == 0
}

def myFunc2(n: Int): Boolean = {
    if (n < 0) myFunc2(n)
    n % 2 == 0
}
```

For any negative number, the function `myFunc` throws a runtime exception. The function `myFunc2` never terminates as it keeps calling itself – more on this when I'll introduce you to pure and impure functions.

Figure 6.1 provides an example of the different components of a function in Scala. It has two main parts: a signature that defines what it does and an implementation that details how it does it. You can split a function signature further into its name, parameters, and return type. Let's review each component.

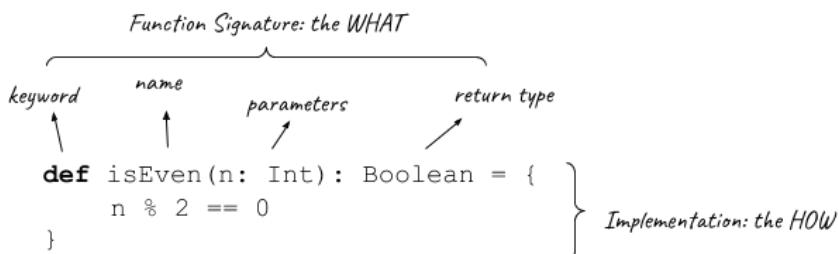


Figure 6.1: The different components of a Scala function. The function signature describes what a function does, and it has the `def` keyword, a name, some parameters, and a return type. The implementation provides detailed instructions on how a function does it.

THE KEYWORD DEF AND A FUNCTION NAME

When declaring a basic function, use the keyword `def` followed by a name. When using the keyword `def`, a function must always have a name:

```
def myFunc() = ??? ①
```

① placeholder for future implementation. More information in the sidebar “The symbol ???”.

The symbol ???

The symbol `???` is one of the most useful and practical features in Scala. It is a placeholder for future code: it tells the compiler that its implementation is currently missing, but you will add it later. If the interpreter evaluates the symbol `???` at runtime, it will throw a `NotImplementedError` exception.

Scala can infer types so that you can often omit them. When using the symbol `???` provide an explicit return type, even if not mandatory: the compiler doesn't have much information, so it will struggle to infer the correct one. Being explicit can also help you remind later on what your intention was and what your implementation should return.

The symbol `???` is a useful tool when designing how the different components of your program interact. It allows you to focus on your code's high-level structure rather than focusing on implementation details.

Starting with an overall design rather than with a detailed implementation is an excellent practice to develop programs that are easy to use, maintain, and understand.

PARAMETERS

A function can have zero or more parameters. If present, they are grouped by parentheses and separated by commas. A parameter is identified by a name and a type. The following are valid function declarations:

```
def myFunc = ???
def myFunc() = ???
def myFunc(a: Int) = ???
def myFunc(a: Int, b: String) = ???
```

When invoking a function, you can refer to function parameters by name and change their order:

```
def myFunc(a: Int, b: String) = ???

myFunc(1, "Scala")           // valid
myFunc("Scala", 1)          // does not compile!!
myFunc(a = 1, b = "Scala")   // valid
myFunc(b = "Scala", a = 1)   // valid
```

You can also specify default values for its parameters:

```
def myFunc(a: Int, b: String = "Hello") = ???

myFunc(1, "Scala")           // valid
myFunc(1)                   // valid
myFunc(a = 1)                // valid
myFunc(b = "Scala")          // does not compile!!!
myFunc(a = 1, b = "Scala")    // valid
myFunc(b = "Scala", a = 1)    // valid
```

You should declare parameters without defaults first: this is common practice but not mandatory. If your function specifies parameters with defaults before others, you must invoke parameters by their name, also called “named parameters”, to trigger the use of their defaults:

```
def myFunc(a: Int = 1, b: String) = ???

myFunc(1, "Scala")           // valid
myFunc("Scala")              // does not compile!!!
myFunc(b = "Scala")          // valid
```

The function call `myFunc("Scala")` does not compile because its first parameter must have type `Int`. On the other hand, `myFunc(b = "Scala")` is valid because the compiler understands that the value refers to the second parameter and that it should use the default value for the first one.

RETURN TYPE

A type defines a subset of values. For example, the type `Boolean` represents the finite set of the values `true` and `false`. A function return type tells you the possible values it can return. A function must always have a return type: you can either provide it explicitly or let the compiler infer one for its implementation. If an implementation is missing, the compiler cannot infer its return type, and you must provide it explicitly. If the implementation of your function is not compatible with its return type, the compiler provides an error message explaining the type found and the type requested:

```
def myFunc(a: Int) // does not compile! Its return type is missing

def myFunc(a: Int): Int    // valid
def myFunc(a: Int) = a + 1      // valid

def myFunc(a: Int): Int = {    // valid
  a + 1
}

def myFunc(a: Int): Boolean = {   // does not compile!
  a + 1                         // found: Int, required: Boolean
}
```

Return type: to omit or not to omit?

You have learned that the compiler can infer the return type of a function from its implementation. You may be wondering when to omit it and when not.

As a rule of thumb, always provide the return type – particularly for public or complex functions!

When providing an explicit return type to your function, you make your code more readable by giving more information about it. You are also taking full advantage of the compiler: by doing so, you are asking the compiler to confirm that your implementation matches the expected return type.

Unless the function is straightforward and with limited scope, specify the return type: you will discover bugs more efficiently and improve your code's readability.

IMPLEMENTATION

The implementation of a function is an ordered sequence of instructions that produces a value that must be compatible with its return type. You do not need to use a `return` keyword in Scala: the value produced by the last expression is the one returned. When a function has no implementation, it is “abstract”. The compiler can infer that it is abstract by its lack of implementation, so you do not need to use any keyword for it: in Scala, the `abstract` keyword is for classes only, not functions.

```
def myFunc(a: Int): Boolean          // valid
abstract def myFunc(a: Int): Boolean // does not compile!
```

```
// invalid keyword
// abstract

def myFunc(a: Int): Int = {           // valid
    a + 1
}
```

In lesson 5, you have learned about Scala expressions. In particular, you have learned that you can omit braces for expressions that are composed of only one instruction: this is still valid also in the context of the implementation of a function:

```
def myFunc(a: Int): Int = {           // valid
    a + 1
}

def myFunc(a: Int): Int = a + 1      // valid

def myFunc(a: Int): Int =           // valid
    a + 1

def myFunc(a: Int): Int = {           // valid
    println(..extra operation here...)
    a + 1
}

def myFunc(a: Int) =                // does not compile!
    println(..extra operation here...) // missing parentheses
    a + 1
```

Have a look at figure 6.2 for a summary of the syntax needed for the several components of a function you have seen so far.

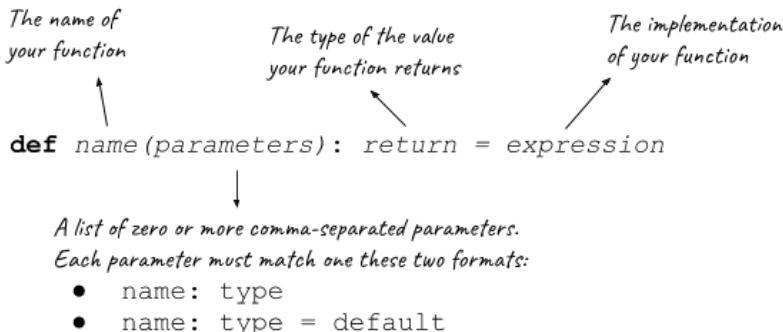


Figure 6.2: Syntax Diagram for a function in Scala and its different components.

QUICK CHECK 6.1

Which of the following functions do not compile? Why?

1. def myFunc(x) = x + 1
2. def myFunc(x: Int) = x + 1
3. def myFunc(x: Int = true): Int
4. def myFunc(x: Int)
5. def myFunc(x: Int): Int = x == 1
6. def myFunc(x: Int) = x == 1

6.2 Summary

In this lesson, my objective was to teach you about the basics of functions in Scala.

- You have learned that a function has a signature and an implementation.
- You have also seen how to define and invoke a function with default parameters.

Let's see if you got this!

TRY THIS

Define a function `pow` that takes two parameters of type `Int`, and that returns an `Int`: call the first parameter `exponent` and the second one `base` with a default of two. The function should calculate the power of a number as follows: if the parameter `exponent` has a value of three and `base` two, it should compute 2^3 . Implement the function using a loop, without using the function `Math.pow`.

6.3 Answers to Quick Checks

QUICK CHECK 6.1

The answers are the following:

1. It doesn't compile: the parameter `x` must have a type.
2. It compiles.
3. It doesn't compile: `true` is a value of type `Boolean`. The compiler doesn't know how to fit a value of type `Boolean` into an `Int`, so you cannot use it as a default for `x`.
4. It doesn't compile: an abstract function must have a return type.
5. It doesn't compile: its return type is `Int`, but the expression `x == 1` returns a value of type `Boolean`.
6. It compiles.

7

Classes and Subclasses to represent the world

After reading this lesson, you will be able to:

- Represent real-world elements and their interactions
- Design class-subclass relations

After discovering the basics of functions in Scala, you'll learn about classes in this lesson. The concept of class is fundamental to the object-oriented paradigm, and it allows you to represent elements and interactions from the real world. Classes, subclasses, and abstract classes enable you to group entities that have common shapes and behaviors. If you are familiar with other object-oriented programming languages, you'll find similarities with Scala's concept of class. Table 7.1 recaps the different types of classes Scala can offer. In the capstone, you'll use a class to represent your vending machine.

Table 7.1: Summary of the types of classes in Scala. They match the definitions of most other object-oriented languages.

Term	Definition
Class	A representation of elements of the same kind from the real world. All its functions have an implementation.
Subclass	A class that inherits behaviors from another class, called superclass. In Scala, a class can only have up to one superclass.
Superclass	A class whose methods are inherited by one or more classes, called subclasses.
Abstract Class	A class in which one or more methods may be abstract (i.e., they do not have an implementation).

Consider this

Imagine you need to represent the lifecycle of the animals of a zoo that hosts cats, dogs, chickens, and llamas. How would you present them? Are there any common behaviors that characterize the lifecycle of these animals?

7.1 Class

Suppose you need to represent a robot with a name in your application. One possible solution is to define a class using the keyword `class`:

Listing 7.1: The class Robot

```
class Robot(name: String)
```

You can create two instances of a robot, called Tom and John, by using the keyword `new` followed by the class name and its parameters:

```
val tom = new Robot("Tom")
val john = new Robot("John")
```

If you try to access the parameter `name` of a `Robot` instance, the compiler will stop you because its parameter is not accessible.

```
tom.name // compiler error: "value name is not a member of Robot"
```

To access the parameter `name`, you need to declare it as a value (or variable) in the class declaration:

Listing 7.2: The class Robot: name as val

```
class Robot(val name: String) ①
```

① `name` is declared as an immutable value

You should now be able to create robots and access their names as follows:

```
val tom = new Robot("Tom")
val john = new Robot("John")

tom.name // returns "Tom"
john.name // returns "John"
```

You can also specify a default value for a class parameter: the compiler will use it when creating an instance of the class without providing a specific value for that parameter. Let's change the class `Robot` to use the string "Unknown" as the default value for the parameter name:

Listing 7.3: The class `Robot`: name with a default value

```
class Robot(val name: String = "Unknown")
```

You can now create a robot without providing a name explicitly:

```
val noName = new Robot
noName.name // returns "Unknown"
```

In general, a class can have zero or more parameters. You can declare functions in it to define some behavior specific to the class. You can refer to a function defined in a class as "method". Let's define a method for a robot to produce a greeting message:

Listing 7.4: The class `Robot`: a greeting functionality

```
class Robot(val name: String = "Unknown") {
    def welcome(n: String) = s"Welcome $n! My name is $name"
}
```

Your robots can now greet people as follows:

```
val tom = new Robot("Tom")
tom.welcome("Martin") // returns "Welcome Martin! My name is Tom"
```

Have a look at figure 7.1 for a summary of how to define and initialize a class.

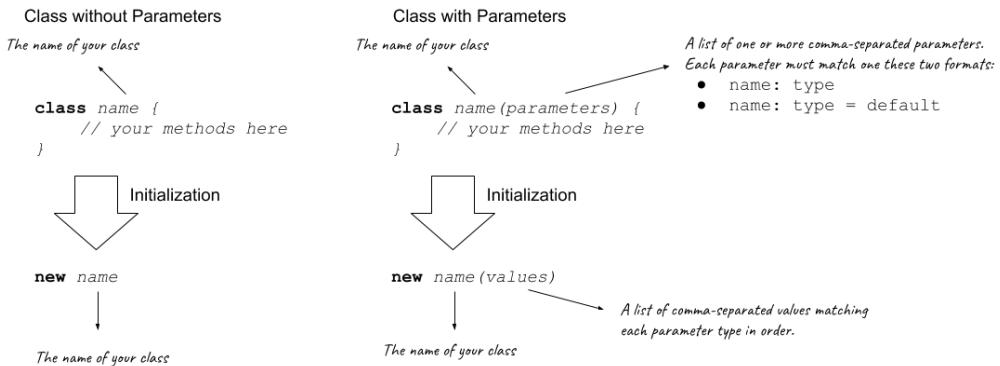


Figure 7.1: Syntax Diagram on how to define and initialize a class in Scala.

Named Parameters

A class can have zero or more parameters. When more than one class parameters have the same type, you should use named parameters to improve readability and avoid ambiguity.

For example, consider a class to represent coordinates:

```
class Coordinate(latitude: Double, longitude: Double)
```

You can create an instance of the class `Coordinate` as follows:

```
new Coordinate(42.42, 24.24)
```

When creating an instance of the class `Coordinate`, you need to remember that the first parameter represents the latitude, while the second represents the longitude. The compiler cannot help you remember their correct order because both the parameters have the same type. A little trick to avoid any ambiguity is to use named parameters:

```
new Coordinate(latitude = 42.42, longitude = 24.24)
```

Because you are explicitly naming the parameters, you are free to change their order: the following expression is syntactically valid, and it produces an equivalent result:

```
new Coordinate(longitude = 24.24, latitude = 42.42)
```

As a rule of thumb, you should use named parameters every time it can improve your code's readability and mistakes related to the order of the parameters that the compiler cannot detect at compile time.

QUICK CHECK 7.1

Create a class `Person` with a name of type `String` and an age of type `Int` – defaulted to 0. Define a method called `presentYourself`, for the class `Person`: it takes no parameters, and it returns a string to communicate the name and age of a `Person`. Create two instances and see what the `presentYourself` method returns for each of them: Martin is 18, and Bob is 0 years old.

7.2 Subclass

Suppose you want to represent robots to welcome people in a specific language. For example, you may need an English robot and an Italian one. Your concept of robot now requires a few

specializations: some are English, some are Italian. Figure 7.2 illustrates how you could express this relation with a class `Robot` and the subclasses `ItalianRobot` and `EnglishRobot`:

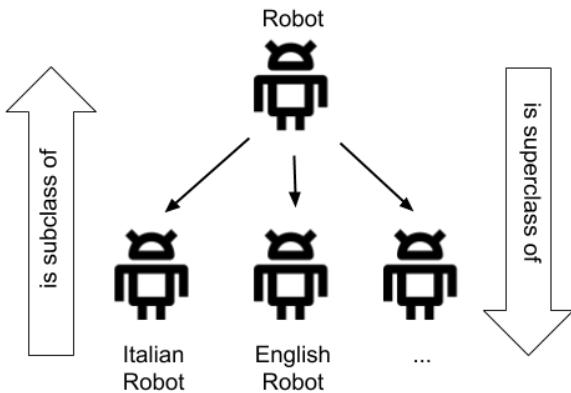


Figure 7.2 Visual representation of hierarchies between robots. `Robot` is the superclass for `ItalianRobot` and `EnglishRobot`, while `ItalianRobot` and `EnglishRobot` are subclasses of `Robot`. The relation has a tree structure: a class can have multiple subclasses, but it can only have one superclass.

Listing 7.5 shows how you can translate this class-subclass relation into code. The class `Robot` defines a generic robot. The classes `ItalianRobot` and `EnglishRobot` extend the class `Robot` by providing a new implementation for the method `welcome`.

Listing 7.5: Italian and English robots

```

class Robot(val name: String = "Unknown") {
    def welcome(n: String) = s"Welcome $n! My name is $name"
}
class ItalianRobot(name: String) extends Robot(name) { ①

    override def welcome(n: String) =
s"Benvenuto $n! Il mio nome e' $name"
}

class EnglishRobot(name: String, country: String) ②
extends Robot(name) { ①

    override def welcome(n: String) =
s"Welcome $n, I am $name from the country of $country!"
}
  
```

① A subclass of `Robot`

② A class can have more parameters than its superclass

A class can have more parameters than its superclass: `EnglishRobot` has an additional one called `country`. You need to specify the keyword `override` because you are redefining a method that already has an implementation. When overriding an abstract function, you can omit the keyword `override`.

In Scala, use the keyword `extends` to express a class-subclass relation. You also use the keyword `override` when redefining a function defined in a superclass. The keyword `override` also tells the compiler that you are deliberately redefining a method: the compiler makes sure that the function you are overriding matches an existing one having the same signature (i.e., the same name, parameters, and return type). Have a look at figure 7.3 for a summary of how to express a class-subclass relation in Scala.

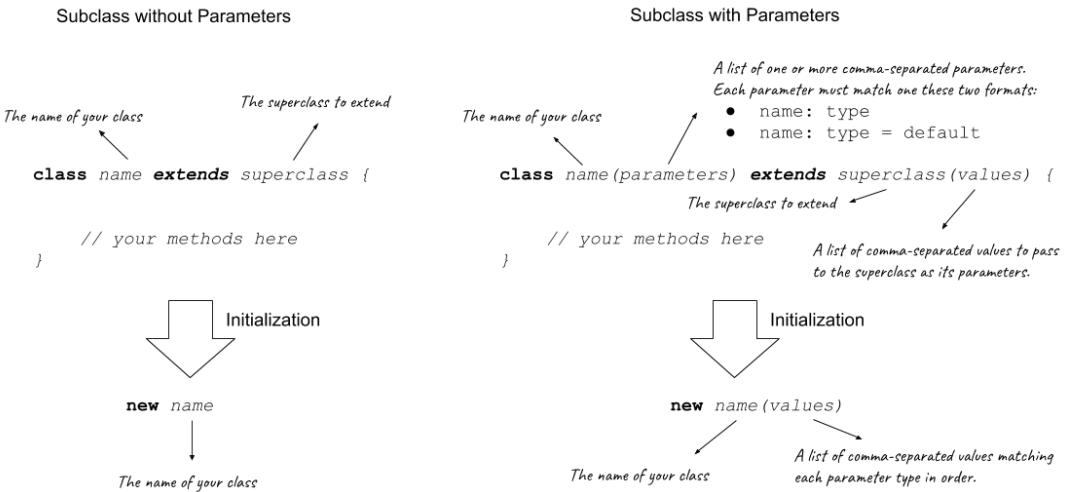


Figure 7.3: Syntax Diagram of how to define and initialize a subclass in Scala.

A class can only have one superclass. In unit 2, you'll learn how to express multiple-inheritance using traits: they are relatively similar to interfaces in other object-oriented programming languages.

QUICK CHECK 7.2

In Quick Check 7.1, you defined a class `Person`. Create two subclasses of the class `Person`: one to represent a teacher, the other to represent a student. A student should have an additional parameter to track its id.

Scala 3: Optional Braces

Scala 3 introduces new indentation rules that make some occurrences of braces optional. For example, consider the following snippet of code:

```
abstract class A {
    def f: Int
}
```

```
class B(n: Int) extends A {
  def f: Int = n * 2
}

You could rewrite it using optional braces as the following:
abstract class A:
  def f: Int

class B(n: Int) extends A:
  def f: Int = n * 2

The code examples in this book do not use optional braces to ensure compatibility with previous versions of Scala.
```

7.3 Abstract class

In the previous section, you have created subclasses for a robot. Because the class `Robot` already provides an implementation for the method `welcome`, their subclasses don't have to re-implement it. Suppose you want to ensure that every robot must implement the `welcome` function from scratch without reusing an existing implementation. You can achieve this by declaring the class `Robot` as abstract by adding the keyword `abstract` and by providing only the function signature for the method `welcome`:

Listing 7.6: The abstract class Robot

```
abstract class Robot(name: String) { ①
  def welcome(n: String): String ②
}
```

① An abstract class

② An abstract function has no implementation

If your class is non-abstract, then all its methods must have an implementation. If this is not the case, the compiler provides an error message to remind you what methods you need to implement:

```
scala> class MyRobot(name: String) extends Robot(name)
<console>:11: error: class MyRobot needs to be abstract, since method welcome in class Robot of
  type (n: String)String is not defined
```

Figure 7.4 provides a syntax diagram of abstract classes in Scala.

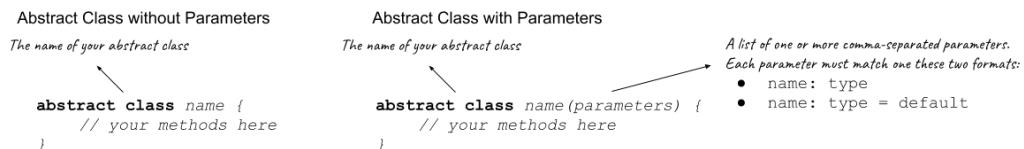


Figure 7.4: Syntax Diagram on how to define an abstract class in Scala. Note that you cannot initialize abstract classes.

QUICK CHECK 7.3

In Quick Check 7.1, you have defined the class `Person`. Change its implementation to force all its subclasses to implement a method called `hello`, which takes one `String` parameter representing a name, and it returns another `String`.

7.4 Summary

In this lesson, my objective was to teach you about the different kinds of Scala classes, a fundamental component of the object-oriented paradigm.

- You have discovered how to use classes to represent elements of the real world.
- You have also learned how to use a class-subclass relation to express inheritance and how to abstract commonalities using abstract classes.

Let's see if you got this!

TRY THIS

A bar serves cold and hot beverages. You can request to add more ice to a cold drink or to reheat a hot one. Express these relations using classes and subclasses.

7.5 Answers to Quick Checks

QUICK CHECK 7.1

You can define the class `Person` as follows:

```
class Person(name: String, age: Int = 0) {
    def presentYourself = s"My name is $name and I am $age"
}
```

The following code creates two instances of the class `Person`:

```
val martin = new Person("Martin", 18)
martin.presentYourself // returns "My name is Martin and I am 18"

val bob = new Person("Bob")
bob.presentYourself // returns "My name is Bob and I am 0"
```

QUICK CHECK 7.2

The following code creates two subclasses of the class `Person`:

```
class Person(name: String, age: Int = 0) {
    def presentYourself = s"My name is $name and I am $age"
}

class Teacher(name: String, age: Int)
extends Person(name, age)

class Student(name: String, age: Int, id: String)
extends Person(name, age)
```

QUICK CHECK 7.3

You should change the implementation of the class `Person` as follows:

```
abstract class Person(name: String, age: Int = 0) {  
    def presentYourself = s"My name is $name and I am $age"  
    def hello(n: String): String  
}
```

©Manning Publications Co. To comment go to [liveBook](#)

©Manning Publications Co. To comment go to [liveBook](#)

Licensed to Mark Hatcher <markallanhatcher@gmail.com>

8

The Vending Machine

In this lesson, you will:

- Implement functions and classes
- Code using variables and values
- Use if-else constructs

In this capstone, you'll implement a class, load it into the REPL, and interact with it. In particular, you'll represent a vending machine that sells two products: a white chocolate bar for \$1.50 and a granola bar for \$1.00. Let's keep things simple and assume that your appliance doesn't give any change back.

A customer can buy an item by selecting a specific product and inserting money into the vending machine. Once the vending machine receives the request, it should check that the product is available and the given money is enough: if all the checks are successful, it should collect the money and release the product.

Let's analyze the business requirements and identify the main components of your vending machine.

8.1 The Vending Machine

Figure 8.1 summarizes the execution flow of the interaction between a customer and the vending machine. A customer requests a product to buy. Following this operation, the vending machine should check if the product is in stock and if the money is enough to cover its cost. When rejecting the request, it should display a human-readable text explaining what went wrong. When successful, it should collect the money and release the product and a message that acknowledges the purchase.

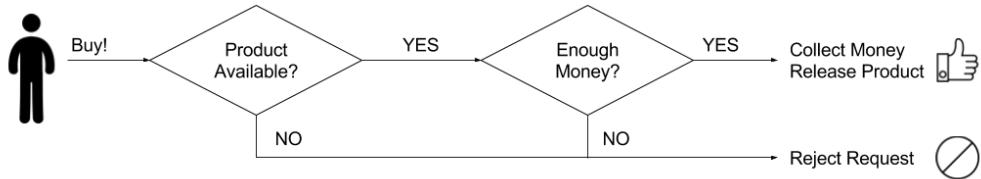


Figure 8.1: Summary of the execution flow for a vending machine. The customer's request to buy a product should be satisfied if and only if the product is available and the inserted money is enough.

Let's first define the API of your vending machine and then focus on its implementation.

8.1.1 The VendingMachine class and its APIs

First of all, you need to identify the main components of your vending machine. A vending machine has a cash register containing the money collected so far and a counter for each product type. Create a new file called `VendingMachine.scala`, and add the following code:

Listing 8.1: The VendingMachine class

```
// file VendingMachine.scala
class VendingMachine {
    var chocolateBar = 0 ①
    var granolaBar = 0 ①
    var totalMoney = 0.0 ②
}
```

① Counter of type Int because 0 is a constant of type Int.

② Counter of type Double because 0.0 is a value of type Double.

A customer can perform one action: requesting a product by giving money in exchange. Let's define a function called `buy` that takes two parameters: a `String` for the product and a `Double` for the money. It should return a human-readable message with the outcome of the request. Add the following method to your class:

Listing 8.2: API for VendingMachine

```
class VendingMachine {
    //...
    def buy(product: String, money: Double): String = ???
}
```

The vending machine also needs to perform the following operations:

1. it should check that the requested product is available;
2. it should verify that the inserted money is sufficient;

3. if all the validations are successful, it should complete it by collecting the money and releasing the product.

Let's translate the following three operations into the following code:

Listing 8.3 Main Operations for a VendingMachine

```
class VendingMachine {
    //...
    def isProductAvailable(product: String): Boolean = ???
    def isMoneyEnough(product: String,
                      money: Double): Boolean = ???
    def completeRequest(product: String,
                        money: Double): String = ???
}
```

You can now implement the function `buy` as the following:

Listing 8.4: Implementation of function buy for VendingMachine

```
class VendingMachine {
    //...
    def buy(product: String, money: Double): String = {
        if (!isProductAvailable(product))
            s"Sorry, product $product not available"
        else if (!isMoneyEnough(product, money))
            "Please, insert more money"
        else
            completeRequest(product, money)
    }

    def isProductAvailable(product: String): Boolean = ???
    def isMoneyEnough(product: String,
                      money: Double): Boolean = ???
    def completeRequest(product: String,
                        money: Double): String = ???
}
```

① You can omit curly brackets here because the function consists of an expression that contains only one if-else construct.

All the pieces are connected! You can now replace the occurrences of the symbol `???` with an implementation.

8.1.2 The Vending Machine and its operations

The vending machine needs to check if the selected product is available and if the money is enough for it to complete a customer request. You have already defined how these functions should look like, but you still need to implement them. For example, you could do the following:

Listing 8.5 Implementing checks for VendingMachine

```
class VendingMachine {

    var chocolateBar = 0
    var granolaBar = 0

    //...

    def isProductAvailable(product: String): Boolean = {
        val productQuantity = {
            if (product == "chocolate") chocolateBar
            else if (product == "granola") granolaBar
            else 0
        }
        productQuantity > 0
    }

    def isMoneyEnough(product: String, money: Double): Boolean = {
        val cost = if (product == "chocolate") 1.5 else 1
        money >= cost
    }
}
```

- ① Any unknown product has zero quantity.

If the request is successful, the vending machine should complete it by collecting the money and releasing the selected product:

Listing 8.6: Implementing completeRequest for VendingMachine

```
class VendingMachine {

    var chocolateBar = 0
    var granolaBar = 0

    var totalMoney = 0.0

    //...

    def completeRequest(product: String, money: Double): String = {
        collectMoney(money)
        releaseProduct(product)
        s"There you go! Have a $product bar"
    }

    def collectMoney(money: Double) =
        totalMoney += money

    def releaseProduct(product: String) =
        if (product == "chocolate") chocolateBar -= 1
        else if (product == "granola") granolaBar -= 1
}
```

The implementation of your vending machine is now complete and ready to use!

8.1.3 Let's try it out

Save the code for your `VendingMachine` class in a file called `VendingMachine.scala` and load it into the REPL:

```
$> scala
Welcome to Scala 2.13.4 (OpenJDK 64-Bit Server VM, Java 15.0.1).
Type in expressions for evaluation. Or try :help.
scala>
```

First, you need to load the code from the file `VendingMachine.scala` and create a vending machine:

```
scala> :load /my/path/to/VendingMachine.scala
defined class VendingMachine

scala> val machine = new VendingMachine
machine: VendingMachine = VendingMachine@415787d7
```

Let's buy a chocolate bar using \$1.5: the vending machine rejects the operation because it contains no products:

```
scala> machine.buy("chocolate", 1.5)
res0: String = Sorry, product chocolate not available

scala> machine.chocolateBar
res1: Int = 0

scala> machine.granolaBar
res2: Int = 0
```

Let's add 2 chocolate bars and 1 granola to the machine:

```
scala> machine.chocolateBar += 2
machine.chocolateBar: Int = 2

scala> machine.granolaBar += 1
machine.granolaBar: Int = 1
```

You should now be able to buy a chocolate bar:

```
scala> machine.buy("chocolate", 1.5)
res3: String = There you go! Have a chocolate bar
```

If you try to buy a product with not enough money, the vending machine rejects the request:

```
scala> machine.buy("chocolate", 1)
res4: String = Please, insert more money
```

Once a product is no longer available, you should no longer be able to buy it:

```
scala> machine.buy("granola", 2)
res5: String = There you go! Have a granola bar

scala> machine.buy("granola", 2)
res6: String = Sorry, product granola not available
```

So far, the vending machine should have collected \$3.50:

```
scala> machine.totalMoney
res7: Double = 3.5
```

8.2 The ugly bits of our solution

Congratulations on completing your first capstone! Your vending machine implementation respects the given requirements, but a few improvements are possible: let's see what these are and what techniques you'll learn to help improve your code.

EVERYTHING IS PUBLICLY ACCESSIBLE

Variables and methods are visible and modifiable without restrictions. Third parties could easily compromise the security of your vending machine by resetting counters and by overriding functions. In unit 2, you'll discover how to use access modifiers to protect your code's sensitive information from unwanted exposure and modifications.

VARS ARE EVIL

You should avoid using `var` in your code: you should limit their scope and use them only when mutability introduces an impressive performance increase. Your program is incapable of dealing with multiple concurrent requests because of its `vars`: what would happen if two calls for the same product arrive at precisely the same time? What if your vending machine has the resources to complete only one of them? In unit 4, you'll learn about case classes and how they can help us overcome these issues by representing data in an immutable manner.

STRING AS THE REPRESENTATION OF A PRODUCT

Representing products as `String` does not provide any information about the products that the machine can offer. Is the product no longer available, or did the user just mistyped its name? Is it "choco" or "chocolate"? Is "ChOCOlate" the same as "chocolate"? In unit 2, you'll see how to represent a finite set of values using sealed traits and objects.

STRING AS RETURN TYPE

Returning `String` to represent your program's outcome is not expressive enough: using an appropriate type would give you a more explicit indication of the operation's result. Suppose you need to execute some additional computation depending on your program's outcome: the only way to understand if the request was successful is to interpret the returned text using a parser. This approach is error-prone and fragile because it is dependent on the specific words used in returned messages. In unit 6, you'll discover the types `Try` and `Either` and how they can help you have meaningful return values for your computation.

YOU VENDING MACHINE IS NOT CONFIGURABLE

Your machine is not configurable: if you need to add a new product or change prices, you need to change your code. In unit 4, you'll learn about data structures, such as `List` and `Map`, and how you can use them to abstract details like products and prices.

8.3 Summary

In this capstone, you have implemented a vending machine.

- You have designed and implemented an easy-to-use API that respects the given business requirements.
- You have also seen which aspects of your implementation are not ideal and mentioned which techniques you'll learn in this book to help you overcome its limitations.

Unit 2

Object-Oriented Fundamentals

In Unit 1, you have reviewed the basics concepts of object-oriented programming in Scala. In this unit, you'll keep examining its fundamentals. You'll implement an SBT executable application to return the current date and time for a given time zone using `java.time`, one of the internal packages available. In particular, you'll learn about the following subjects:

- Lesson 9 demonstrates how to structure your program in logical packages and import internal and external ones into your code.
- Lesson 10 shows you how to use access modifiers to control the scope of your values, functions, and classes and to prevent your program from exposing sensitive information and unwanted modifications.
- Lesson 11 illustrates objects as built-in singleton support in the language. You'll also discover how to implement static methods and new constructors for a given class through its companion object.
- Lesson 12 introduces traits as a tool to define commonalities between different classes, one of its most popular use cases. You are probably already familiar with this concept: many object-oriented languages refer to it as "interface".
- Finally, you'll apply these concepts and implement an sbt application to return the current time in a given timezone in lesson 13.

Once you have learned how to create an sbt executable application, you'll continue with Unit 3, in which you'll see how to implement a simple HTTP server.

9

Import and Create Packages

After reading this lesson, you'll be able to:

- Import and reuse code from other packages
- Structure your program in logical packages

After reviewing the basics of object-oriented programming in unit 1, you'll learn about packages. They allow you to organize your software in logical groups to navigate your codebase more easily. By importing them, you can reuse them in several parts of your application. You can also publish them – making their code available online for other people to download and use: you refer to a published package as "library". In the capstone, you will create a dedicated logical unit for your time application, and you will import and use an internal package called `java.time`.

Consider This

Suppose you want to create a web service that exposes an API. It should receive data through, for example, a POST request, perform some business validation, and store the information in a database. What are the main components of your service? What package structure would you use to organize your code?

9.1 Import an existing Package

Imagine you want to write a program to read the content of a file into a `String`. This task would be quite challenging and complex without the use of a package or library. You would need to identify the location of the memory cells for each binary file. You would read and convert them to text after applying a particular encoding (e.g., UTF-8). You would also need to handle many edge cases, such as trying to access a file that could be corrupted, missing, or using an unexpected encoding.

Luckily, you don't have to deal with such low-level operations, but you find an internal package or library to do all the work for you. The following snippet of code shows how you can do this by exploiting the `scala.io` module:

Listing 9.1: Implementing `readFileToString`

```
import scala.io.Source ①

def readFileToString(filename: String) =
  Source.fromFile(filename) ②
    .getLines() ③
    .mkString("\n") ④
```

- ① It includes `Source` from the package `scala.io`
- ② It returns a `BufferedSource`, which is a reference to the file.
- ③ It reads the file and creates an instance of type `Iterator[String]` by parsing each line of the file as `String`.
- ④ It concatenates all the strings using the char "`\n`"

Scala imports are quite versatile: you can import all the code defined inside a package, or you can be extremely selective by including a specific class, function, or value in your scope. You can also specify an alias for the element you imported to make your code more expressive or to resolve clashes between components with the same name. Listing 9.2 provides a few examples of what you can do with imports in Scala.

Listing 9.2: Several uses of import

```
import scala.concurrent.duration._ ①
import scala.concurrent.duration.FiniteDuration ②
import scala.concurrent.duration.Duration.Inf ③
import scala.concurrent.duration.Duration.create ④
import scala.io.{Source => Src} ⑤
import scala.math.{BigDecimal, BigInt} ⑥
```

- ① It imports everything from `scala.concurrent.duration`
- ② It imports the class `FiniteDuration` from `scala.concurrent.duration`
- ③ It imports the value `Inf` of the object `Duration` from `scala.concurrent.duration`
- ④ It imports the function `create` of the object `Duration` from `scala.concurrent.duration`
- ⑤ It imports the file `Source` from `scala.io`, and it creates an alias `Src` for it
- ⑥ It imports the classes `BigDecimal` and `BigInt` from `scala.math`

Try to import just what you require in your scope: it will make your compilation faster and your code more readable by listing all the packages your class requires. You can add imports in a file, a class, an object, a function, and even a value. In the following example, the code for `scala.io.Source` is accessible only from within the scope of the function `readFileToString`:

```
def readFileToString(filename: String) = {
  import scala.io.Source

  Source.fromFile(filename)
    .getLines()
    .mkString("\n")
}
```

Have a look at figure 9.1 for a summary of the different usages of Scala imports you have seen so far.

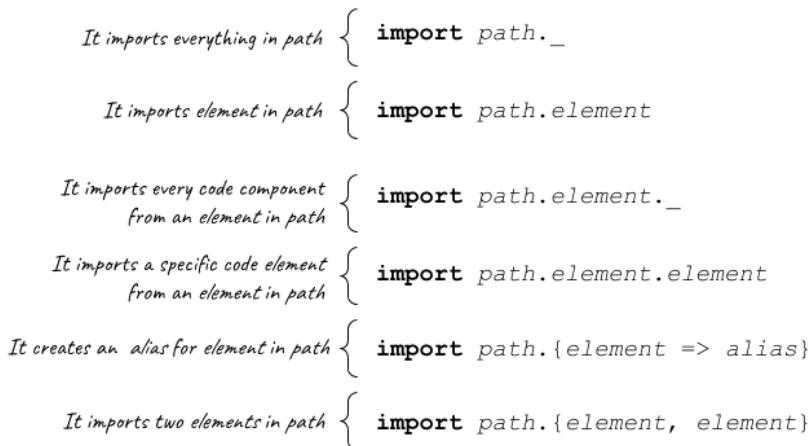


Figure 9.1: A syntax diagram for the functionalities of an import in Scala. Imports allow you to add existing code elements into your scope: you can add all the code in a package, a single class, object, function, or even value.

Scala automatically adds the packages `scala`, `java.lang`, and `scala.Predef` to the scope of your files with extension `.fscala` so that you can use the most common features and classes of the language without using imports.

Scala 3: An alternative import syntax

Scala 3 introduces an alternative syntax for imports: you can use the symbol `*` instead of `_`, and `as` instead of `=>`. For example, consider the following import instructions:

```

import scala.concurrent.duration._
import scala.io.{Source => Src}

You can rewrite them as:
import scala.concurrent.duration.*
import scala.io.{Source as Src}
  
```

QUICK CHECK 9.1:

Import the package `java.sql.Date` into your scope and create an alias called “`SqlDate`”.

9.2 Create a Package

In section 9.1, you saw how to write a function to read a file into a string by importing an internal package. You are going to see how you can create a custom package. Because packages are tools

to organize an application composed of multiple files, you will not use the REPL, but you will create an sbt project instead. Use your IDE to create an empty sbt project – most of them have dedicated support to do this in just a few clicks. Alternatively, you can reuse the `hello-world-template` gitter8 template you have seen in lesson 3 by typing the following command in your terminal:

```
$ sbt
sbt> new scala/hello-world.g8
```

Suppose you want to create a package called `my.example.io` so that you can reuse the code shown in listing 9.1 in other parts of your application. First, create a directory with relative path `src/main/scala/my/example/io` then create a file, called `MyExample.scala`:

Listing 9.3: Creating a package `my.example.io`

```
// File src/main/scala/my/example/io/MyExample.scala

package my.example.io ①

import scala.io.Source

class MyExample { ②

    def readFileToString(filename: String) =
        Source.fromFile(filename).getLines().mkString("\n")
}
```

① The class `MyExample` belongs to a package called `my.example.io`

② You need a `class` here because a function can live only inside a class or singleton object.

A package name doesn't have to match a specific directory structure in your sbt project, but it is a good practice that makes source code easier to find and maintain. Scala automatically includes all the other files that belong to your package in your scope, so you do not need to import them. Have a look at figure 9.2 for a summary of how to create packages in Scala.

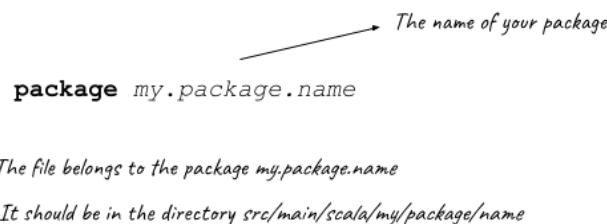


Figure 9.2: Syntax diagram on how to create a Scala package. Ensure your package name matches your directory structure: its files will be easier to find.

QUICK CHECK 9.2:

Create a new package in an sbt project, called `my.quick.check`. Add two files to it:

- `TestA.scala` contains a class called `TestA`

- TestB.scala contains a class called TestB

9.3 Summary

In this lesson, my objective was to teach you about packages in Scala.

- You have discovered how to include code from libraries and internal packages.
- You have also learned how to create packages to logically group code with similar goals and functionality.

Let's see if you got this!

TRY THIS

Create a package in an sbt project, called `my.areas`, that contains two files: `Circle.scala` and `Square.scala`.

9.4 Answers to Quick Checks

QUICK CHECK 9.1:

The following import includes the class `java.sql.Date` as `SqlDate` into your scope:

```
import java.sql.{Date => SqlDate}
```

QUICK CHECK 9.2:

The files should have the following path and content:

```
// File src/main/scala/my/quick/check/TestA.scala
package my.quick.check

class TestA

// File src/main/scala/my/quick/check/TestB.scala
package my.quick.check

class TestB
```

10

Scope your code with Access Modifiers

After reading this lesson, you will be able to:

- Make your values, variables, functions, and classes accessible only from their class
- Limit your code accessibility to a class and its subclasses

In lesson 9, you have learned about packages. Protecting your code by limiting its accessibility is a good practice, especially when creating external modules for third parties. By doing so, you also reduce your code's public entry points, making it easier to use. This approach also prevents external manipulations that could expose sensitive data or introduce undesired behavior in your code. In this lesson, you'll learn about the Scala access modifiers, which are reserved keywords to change your code elements' visibility. You should use them to protect and hide information that you should not expose to the public. You'll use access modifiers to implement auxiliary functions accessible within their class in the capstone.

Consider this

Suppose you are designing a program to manage bank accounts. Your business requirements demand not to expose their balance unless its owner requests it. How would you structure your code to enforce this requirement?

10.1 Public, the Default Access Modifier

Imagine you need to track the guests and the costs of a party. People registering for the event should be the only publicly accessible functionality. The following snippet provides a possible implementation:

Listing 10.1: The class Party

```
class Party {
    var attendees = 0

    def register(guests: Int) =
        attendees += guests
}
```

Every time your code does not use an explicit access modifier, the compiler applies the default access level, which is the public one. A code element marked as public has no access restrictions: anyone can see its value and modify it. The public access level doesn't have an explicit keyword: if you want to declare an element as public, you don't provide any access modifier keyword.

```
public val a = "hello" ①
val b = "world" ②
```

- ① it doesn't compile: the keyword public doesn't exist
- ② it compiles: the value is publicly accessible.

QUICK CHECK 10.1:

In the REPL, declare a value, called `name`, containing the string "Scala": make it publicly usable.

10.2 Private

Let's have another look at the following snippet of code about registering attendees for an event that you have implemented in listing 10.1:

```
class Party {
    var attendees = 0

    def register(guests: Int) =
        attendees += guests
}
```

The variable `attendees` is publicly accessible: external users may break your application by resetting its assignment to any value of their choice. You can prevent this by marking the variable `attendees` as private, making it no longer accessible outside the class `Party`.

Listing 10.2: The class Party: attendees are private

```
class Party {
    private var attendees = 0

    def register(guests: Int) =
        attendees += guests
}
```

In Scala, you use the `private` access modifier to prevent access to functions, values, and classes that should not be used publicly: when a code element is `private`, you can only access it from inside its class. For example, consider the following snippet of code:

Listing 10.3: Defining a class Test

```
class Test {
    val configA = "I am public"
    private val configB = "I am private"
}
```

Let's use the REPL to define a class `Test`, create an instance for it, and access its values:

```
scala> class Test {
|     val configA = "I am public"
|     private val configB = "I am private"
| }
class Test

scala> val test = new Test
test: Test = Test@3128213f

scala> test.configA
res0: String = I am public

scala> test.configB
^
      error: value configB in class Test cannot be accessed as a member
      of Test from class
```

QUICK CHECK 10.2:

Consider the following snippet of code: is `age` accessible from the class `Student`? Use the REPL to validate your hypothesis.

```
class Person {
    private val age = 18
}

class Student extends Person
```

10.3 Protected

After tracking the number of attendees, suppose you need to estimate the cost of an event. You don't want this sensitive information to be public or having external sources making changes to it. At the same time, you'd like to make this functionality reusable for any event to ensure you calculate costs consistently. In this case, you can use the `protected` access modifier: a protected code element is accessible only from its class and its subclasses. Let's see how you could use the `protected` access modifier to control the access to the function computing the cost estimation of an event:

Listing 10.4: The class Event

```
class Event {

    protected def estimateCosts(attendees: Int): Double = ①
        if (attendees < 10) 50.00 else attendees * 12.34
}

class Party extends Event { ② }
```

```

private var attendees = 0

var cost = estimateCosts(attendees) ③

def register(guests: Int) =
    attendees += guests
}

```

- ① Usage of the protected access modifier
- ② Party is a subclass of Event
- ③ Party has access to Event's protected code elements.

The `protected` access modifier allows code to be visible from the current class and any code element that extends it, such as a subclass or interface – a topic that you'll discover in lesson 12.

QUICK CHECK 10.3:

Can you access the function `estimateCosts` from outside the class `Event`? Use the REPL to validate your hypothesis.

10.4 Which Access Level to use?

You have now learned several ways to change the visibility of your code. When deciding which access level to use, the temptation of using the public default access level everywhere is quite strong. But this is not a good practice. Exposing your code as public can be confusing for its users because it makes its entry points challenging to identify. By blocking access to portions of your program, you are preventing others from overriding and improperly reuse them. Finally, the more code you expose to the public, the more complicated future changes will be: you will not be able to control what parts of your system you can change without breaking external code that depends on it. As a rule of thumb, pick the most restrictive access modifier that satisfies your needs, and avoid using the public access modifier unless strictly necessary. Table 10.1 summarizes the usage of the access modifiers you have seen so far.

Table 10.1 Comparing access modifiers in Scala. A code element is public by default: you can access it from everywhere. You can access it from its class and subclasses when protected and only from its class when private.

	Private	Protected	Default (Public)
Class	yes	yes	yes
Subclass	no	yes	yes
Global	no	no	yes

10.5 Summary

In this lesson, my objective was to teach you about access modifiers in Scala, which are keywords that control the visibility of your code.

- You have discovered that your code elements are public by default: anyone can access and modify them.
- You have learned that the private access modifier blocks access to your code outside the current scope.
- You have also seen that the protected access modifier makes it visible from the current class or any of its subclasses.

Let's see if you got this!

TRY THIS

Define a class `Employee` with three values: a name of type `String`, an age of `Int`, and a salary of type `Double`. While its name should be publicly accessible, its age should be visible only from its class and subclasses, and its salary should be private.

10.6 Answers to Quick Checks

QUICK CHECK 10.1:

Start the Scala REPL and type the following expression:

```
scala> val name = "Scala"
name: String = Scala
```

QUICK CHECK 10.2:

The value `age` is not accessible from the class `Student` because it is visible only from an instance of the class `Person` and not from any of its subclasses.

```
scala> class Person {
|   private val age = 18
| }
|
| class Student extends Person
class Person
class Student

scala> val student = new Student
student: Student = Student@62d1dc3c

scala> student.age
^
error: value age is not a member of Student
```

QUICK CHECK 10.3:

You cannot publicly access the function `estimateCosts` of the class `Event`. You declared `estimateCosts` as protected: you can access it from either an instance of `Event` or any of its subclasses, such as the class `Party`.

```
scala> class Event {
|   protected def estimateCosts(attendees: Int): Double =
|     if (attendees < 10) 50.00 else attendees * 12.34
| }
```

```
class Event

scala> val event = new Event
event: Event = Event@6af78a48

scala> event.estimatedCosts(5)
^
error: value estimatedCosts is not a member of Event
```

11

Singleton Objects

After reading this lesson, you will be able to:

- Implement a singleton object
- Use objects as executable programs
- Define static functions in a companion object
- Create factory methods using the `apply` function

The essence of Object-Oriented Programming is representing elements of the real world using classes that describe their behavior through methods. In specific scenarios, such as configurations and main entry points, you need to instantiate a coding element at most once: you usually call it "singleton". In Scala, you can create singletons using objects elegantly and concisely. You are also encouraged to clearly distinguish between static and non-static methods: the non-static ones act on a specific instance of a class, while the static ones belong to its general definition. In Scala, you implement non-static methods in a class and static methods in an object. In the capstone, you'll use an object to define the main entry point of your application.

Consider This

Can you think of a few possible use cases for singletons? What about static methods for a class? Can you think of a few examples?

11.1 Object

Consider the following class to represent a robot:

```
abstract class Robot(name: String) {  
    def welcome: String  
}
```

The function `welcome` is abstract, it takes no parameters, and it returns a value of type `String`. Suppose you want to represent a vocabulary of sentences that a robot can use when asked to speak. This vocabulary must be autonomous from any particular implementation of a robot: it must be unique in the whole application and shared between multiple sources. Listing 11.1 shows how you can implement it using an object:

Listing 11.1: A vocabulary object

```
object Vocabulary {  
    val sentenceA = "Hi there!"  
    val sentenceB = "Welcome!"  
    val sentenceC = "Hello :)"  
}  
  
Vocabulary.sentenceA ①  
Vocabulary.sentenceB ②
```

- ① returns “Hi there!”
- ② returns “Welcome!”

When using the object `Vocabulary`, you refer to it by name. Note that you do not use the keyword `new` because you cannot request to instantiate an object explicitly. The compiler guarantees that the JVM instantiates an object no more than once – in other words, it treats it as a singleton. The first time you request access to an object, the JVM allocates it in memory; following references to the same object do not trigger any new instantiation because the JVM reuses its first memory allocation: this model is called “singleton pattern”.

Think in Scala: the term “object”

Many object-oriented languages use the word “object” to indicate an instance of a class. A few examples are Java, Python, JavaScript, and C++.

In Scala, “object” has a well defined and specific meaning: it refers to a singleton, not an instance of a class. When referring to Scala code, make sure to use the term “object” correctly to avoid confusion!

The use of singletons is not unique to Scala. Still, its implementation is extremely elegant and convenient to use compared to other languages: listing 11.2 shows the comparison between implementing a singleton in Java, JavaScript, and Scala.

Listing 11.2: Singleton in different languages

```
// Java  
public class MySingleton {  
    private static MySingleton instance = null;
```

```

private MySingleton() {}

public static MySingleton getInstance() {
    if(instance == null) {
        instance = new ClassicSingleton();
    }
    return instance;
}

// JavaScript
var MySingleton = (function () {
    var instance;

    function createInstance() {
        var object = new Object("my-instance");
        return object;
    }

    return {
        getInstance: function () {
            if (!instance) {
                instance = createInstance();
            }
            return instance;
        }
    };
})();

// Scala -> only 2 words!
object MySingleton

```

Have a look at figure 11.1 for a syntax diagram on how to define objects in Scala.

```

object name {
    // your code here
}

```

The name of the singleton object to create

Figure 11.1: Syntax Diagram on how to implement an object in Scala.

QUICK CHECK 11.1

Consider the following snippet of code. Does this code compile? Why? Use the REPL to validate your hypothesis.

```

object MySnippet
new MySnippet

```

11.2 Executable Object

A program usually has at least one entry point to start its execution: you can refer to this as “main” or “executable”. Note that a main for a program is a singleton: you cannot have more than one instance of the same main when running a program.

In Scala, your program’s entry point is an object with a `main` function: this behavior is consistent across the JVM languages. Listing 11.3 demonstrates how to create an executable object that prints the text “Hello World!” in the console.

Listing 11.3: Hello World!

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

Scala also offers another more elegant and concise way of defining a program’s entry points: you’ll discover it in the next lesson when you’ll learn about traits.

QUICK CHECK 11.2

When running your program with SBT, what happens if it cannot find its main entry point? What if it finds more than one? Verify your hypothesis by writing two simple programs and executing them with SBT: one with no main, the other with two.

Scala 3: The @main annotation

Scala 3 offers you an alternative way of declaring the entry point of an executable program that uses the annotation `@main`. For example, you could rewrite the snippet of code from listing 11.3 as following:

```
object HelloWorld {  
  
    @main def hello: Unit = {  
        println("Hello world!")  
    }  
}
```

The `@main` annotation extracts command line arguments, and it ensures they match the function signature. For example, consider the following `@main` method:

```
object Hello {  
  
    @main def echo(n: Int, word: String): Unit = {  
        println(word * n)  
    }  
}
```

You can execute it in sbt using the command `run` and pass command line argument to it:

```
$ sbt  
..
```

```
sbt> run
[info] running echo
Illegal command line: more arguments expected
[success] Total time: 1 s, completed 20 Feb 2021, 13:09:07
// It fails because you didn't provide any command line arguments

sbt> run test
[info] running echo test
Illegal command line: java.lang.NumberFormatException: For input string: "test"
[success] Total time: 0 s, completed 20 Feb 2021, 13:09:40
// It fails because the first parameter is not a number

sbt> run 2 test
[info] running echo 2 test
testtest
[success] Total time: 0 s, completed 20 Feb 2021, 13:09:14
// It prints the text "testtest"
```

11.3 Companion Object

In Scala, a *companion object* is an object that has the same name as another existing class. Consider the `Robot` class you have seen in section 11.1 and suppose you want to define a function to return the most talkative robot. Listing 11.4 shows a possible implementation:

Listing 11.4: The most talkative robot

```
abstract class Robot(name: String) { ①

    def welcome: String
}

object Robot { ②
    def mostTalkative(r1: Robot, r2: Robot): Robot = {
        val r1Size = r1.welcome.length ③
        val r2Size = r2.welcome.length
        if (r1Size >= r2Size) r1 else r2
    }
}
```

① The class `Robot`

② The companion object for `Robot`

③ You are invoking the method `welcome` on `r1`, and computing the length of its returned value.

An example usage for `mostTalkative` and the object `Vocabulary` implemented in listing 11.1 is following:

```
val tom = new Robot("Tom") {
    def welcome = Vocabulary.sentenceA
}
```

```
val alice = new Robot("Alice") {
  def welcome = Vocabulary.sentenceB
}
Robot.mostTalkative(tom, alice)
```

The function `mostTalkative` operates on two robots, rather than just one: it doesn't belong to a specific instance of `Robot`, but it still has to do with the class `Robot`. The function `mostTalkative` is a static function of the class `Robot`. A "static" method is a function acting on a class rather than on a specific instance of that class.

QUICK CHECK 11.3

Consider the function `welcome` defined in the class `Robot`. Is it a static method? Why?

In the next section, you'll discover a particular static method dedicated to creating class instances, called `apply`.

11.4 The Apply Method

In lesson 7, you have learned that a class can have one constructor, and you can use it by using the keyword `new`. What if you want to specify different ways of creating a class? How can you do it with only one constructor? The `apply` function of a companion object can help in solving your problem.

Assume you have a `Person` class, and you can create an instance for it by calling the constructor using the keyword `new` and providing its parameters. You'd also like to define an alternative way of creating a person by merging two existing ones. Listing 11.5 shows you how to do it using the `apply` function:

Listing 11.5: The class Person and its companion object

```
class Person(val name: String, val age: Int)

object Person {

  def apply(p1: Person, p2: Person): Person = { ①
    val name = s"Son of ${p1.name} and ${p2.name}"
    val age = 0
    new Person(name, age)
  }
}
```

① you must provide an explicit return type for `apply`

Once you have defined an `apply` method, you can use it in the same way as any other static method implemented in the companion object. You can declare multiple `apply` functions in the companion object, as long as their signature is uniquely identifiable by the compiler. Consider the following snippet of code:

```
object Person {

    def apply(name: String): Person = new Person(name, 0)

    def apply(age: Int): Person = new Person("Mr Unknown", age)

}
```

The two `apply` methods have the same name and return type, but they take different parameters: the first takes a string, while the second one an integer. The two implementations can co-exist because the compiler can select one or the other by looking at the types of their parameters. You can refer to the concept of having multiple functions with the same name but different parameters with the term "function overloading".

Developers often use the `apply` function to create class instances. The compiler offers you some syntactic sugar so that you can omit the function name `apply`. When passing a parameter to an object, it looks for an `apply` method defined in that object with parameters that match the ones you have given. Consider the following snippet of code: the two expressions `Person.apply(tom, alice)` and `Person(tom, alice)` are equivalent.

```
val tom = new Person("Tom", 24)
val alice = new Person("Alice", 23)

Person.apply(tom, alice)
Person(tom, alice)
```

The `unapply` method: a quick preview

You have discovered that the `apply` method is a static function to construct class instances. The method `unapply` is complementary to it: it deconstructs a class into its parameters.

`unapply` is a static function that lives in a class's companion object: it takes an instance class and returns its decomposed representation. For example, you could identify a `Person` by providing a pair with its name and age. The `unapply` function allows you to use pattern matching, a powerful and expressive tool of the Scala toolbox to match conditions (e.g., age bigger than 18) based on the decomposition of a class: you are going to learn about this in the next unit.

You'll learn about the `unapply` method later in the book when discussing case classes and the type `Tuple` in lesson 25.

Figure 11.2 provides a summary of the methods of a companion object that you have seen so far.

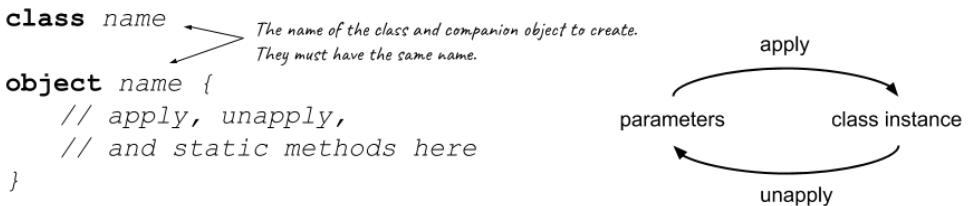


Figure 11.2: A syntax diagram for companion objects, their use, and a visual representation of the duality between the `apply` and `unapply` methods.

From Scala 3, the compiler automatically generates an `apply` function for all your classes: you can omit the `new` keyword even if you haven't manually defined an `apply` method for it.

```

class Test(n: Int)
object Test // a companion object without an apply method

new Test(5) // works in all Scala versions
Test(5) // works in Scala 3+
  
```

QUICK CHECK 11.4

Consider this snippet of code. Are the expressions `new Dog("Trigger")` and `Dog("Trigger")` equivalent? Why?

```

class Dog(val name: String)

object Dog {
    def apply(name: String): Dog =
        new Dog(s"$name The Dog")
}
  
```

11.5 Summary

In this lesson, my objective was to teach you about objects and their use in Scala.

- You have seen that objects are a convenient implementation for singletons.
- You have also discovered that companion objects have the same name as an existing class and that you can use them to declare static methods that refer to it.
- Finally, you have learned about the `apply` method as the standard way of creating class instances.

Let's see if you got this!

TRY THIS

In listing 11.5, you have added an `apply` method to create an instance of a person given two existing ones. Create new functions for the class `Person` to:

- Determine the oldest between two people
- Create a person by copying the parameters of an existing one

11.6 Answers to Quick Checks

QUICK CHECK 11.1

The code doesn't compile. The REPL returns the following output:

```
scala> object MySnippet
defined object MySnippet

scala> new MySnippet
<console>:11: error: not found: type MySnippet
```

The first expression successfully defines an object called "MySnippet". The second one is incorrect: you cannot request the creation of an object. When using the keyword new, the compiler looks for a class or type called "MySnippet": an object is neither, so the compiler rejects it with a missing type error.

QUICK CHECK 11.2

When SBT doesn't find any entry point, it will throw a `RuntimeException` with the message "No main class detected". If it detects more than one, it provides a list of the available entry points so that you can choose the one to execute:

```
Multiple main classes detected, select one to run:

[1] AnotherHelloWorld
[2] HelloWorld

Enter number: [pick your entry point here]
```

QUICK CHECK 11.3

The method `welcome` is not static: it acts on a `Robot`'s instance, the reason why you defined it in the class `Robot` rather than in its companion object.

QUICK CHECK 11.4

The two expressions are not equivalent. The first returns a `Dog` instance with the name "Tigger" because it invokes its constructor directly. The second one returns one named "Tigger The Dog" because it calls the `apply` method defined in its companion object.

12

Traits as interfaces

After reading this lesson, you will be able to:

- Declare an interface using a trait
- Implement classes, objects, and traits that conform to one or more interfaces
- Define a closed set of values using sealed traits

After discovering singleton objects in lesson 11, let's learn about traits. Earlier in this book, you have seen that a class can have up to one superclass: you cannot express multiple-inheritance using classes and abstract classes. Traits are very similar to abstract classes but with a fundamental difference: a class can inherit from one or more traits. You can use them to express multiple-inheritance: you cannot achieve the same with abstract classes. You can use traits to define interfaces to represent a set of features (hence the name "traits") that your class must have: this is a crucial object-oriented concept that you find in many languages, such as Go, Kotlin, and Java. In future units, you will learn that traits are a lot more than just a way to express an interface: you'll see some of their superpowers and why this makes them a more expressive and powerful tool. In this lesson, you'll discover how to create a trait and implement classes, objects, and other interfaces that conform to one or more interfaces. You'll also learn about sealed traits: they allow you to define a closed set of implementations for your interface that the compiler guarantees at compile time. You'll finally discover the enumeration syntax that Scala 3 introduces as an alternative to sealed traits. In the capstone, you'll use the `App` trait to implement the entry point of your application.

Consider this

Consider the programming languages that are already familiar to you. Do they support the concept of interface? How do they express it? Do they support multiple-inheritance?

12.1 Defining traits

Suppose you need to represent a zoo's animals and make sure they all have some common behavior: they all sleep, eat and move. Let's define an interface to enforce these functionalities. Listing 12.1 shows you how to achieve this using a trait:

Listing 12.1: The trait Animal

```
trait Animal {
    def sleep = "Zzz"
    def eat(food: String): String
    def move(x: Int, y: Int): String
}
```

In Scala, use the keyword `trait` and a name to identify an interface. In Scala 2, a trait can have abstract and fully implemented coding elements, such as functions, values, and variables. Listing 12.2 provides another example of a trait:

Listing 12.2: The trait Nameable

```
trait Nameable {
    def name: String
}
```

From Scala 3, a trait can also have parameters. For example, in Scala 3 you can redefine the trait `Nameable` as the following:

```
trait Nameable(name: String) // valid in Scala 3 only!
```

Have a look at figure 12.1 for a summary of how to define a trait.

```
trait name {
    // your code here
}
```

Figure 12.1: Syntax diagram of how to define a trait in Scala: you can use them to implement interfaces.

QUICK CHECK 12.1

Define an interface called `Printable` that enforces the presence of a function called `print`: it takes no parameters, and it returns `Unit`.

12.2 Extending traits

After creating an interface, let's see how you can define classes, objects, and other traits that conform to it. For example, you can implement a class `Cat` that extends the trait `Animal` as follows:

Listing 12.3: The Cat class

```
class Cat extends Animal {
    override val sleep = "sleepy cat!" ①
    def eat(food: String) = s"the cat is eating $food" ②
    def move(x: Int, y: Int) = s"the cat is moving to ($x,$y)" ②
}
```

① You are redefining a fully implemented function, so you must use the keyword `override`.

② You can omit the keyword `override` because you are providing an implementation for an abstract function.

You can also extend more than one interface at the same time. For example, you can define a `Dog` class to represent an animal with a name:

Listing 12.4: The Dog class

```
class Dog(val name: String) extends Animal with Nameable { ①
    def eat(food: String) = s"$food $food"
    def move(x: Int, y: Int) = "let's go to ($x, $y)!"
}
```

① the keyword `val` marks the field name of the class `Dog` as publicly accessible

The trait `Nameable` requires your class to implement `def name: String`, a method without parameters and without parentheses called `name` that returns a `String`. The class `Dog` defines a publicly accessible field called `name`, which is considered a valid implementation for `def name: String`. In Scala, values and variables can implement or override functions with no parameters and no parentheses.

You can now define functionalities that act on any instance that conforms to a given interface:

```

val tiggerTheDog = new Dog("Tigger")
val cat = new Cat

def feedTreat(animal: Animal) =
    animal.eat("treat")

feedTreat(tiggerTheDog) // compiles!
feedTreat(cat)         // compiles!

def welcome(nameable: Nameable) =
    println(s"Hi, ${nameable.name}!")

welcome(tiggerTheDog) // compiles!
welcome(cat)          // doesn't compile: Cat doesn't extend Nameable

```

In Scala, you can define a coding element that inherits from an interface using the keyword `extends`; if you need to conform to more than one trait, you can do so using the keyword `with`. The compiler will guarantee that it respects its interfaces, or it will fail with an error message listing methods and fields that you still need to implement. Figure 12.2 summarizes how to extend traits in Scala.

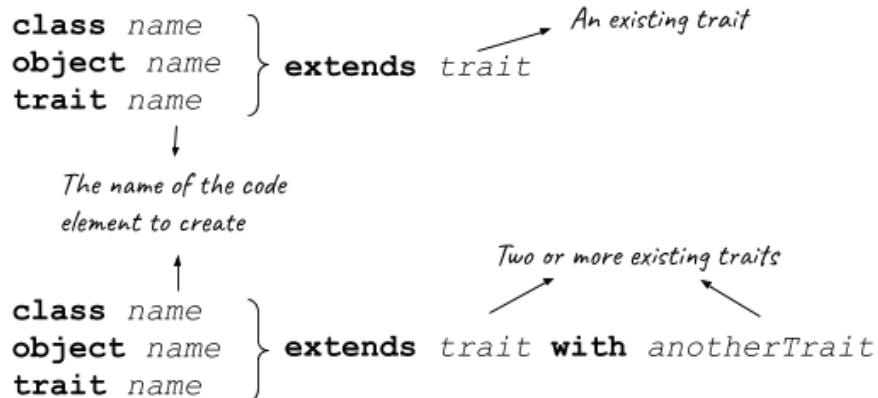


Figure 12.2: Syntax Diagram on creating classes, objects, and traits that conform to an interface using the keywords `extends` and `with`.

The trait App

In lesson 11, you have learned that Scala identifies a program's entry points by looking for a `main` function with a specific signature. The following snippet of code provides an example of an executable object:

```

object HelloWorld {

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

```

You can achieve the same result by extending `App`. This trait is part of the standard Scala packages: the compiler automatically adds it into your scope without adding an import instruction. The trait `App` makes writing application entry points much more straightforward and concise: it automatically includes a main function with the correct signature to ensure that the compiler detects your object as an entry point for your program. Thanks to the trait `App`, you can rewrite the above snippet of code as follows:

```
object HelloWorld extends App {
    println("Hello world!")
}
```

Extending the trait `App`, rather than implementing a `main` function, is the conventional way of creating an executable Scala object.

Quick Check 12.2

Which of the following statements are correct? Use the REPL to validate your hypothesis.

1. A class can extend a class
2. A class can extend an object
3. A class can extend a trait
4. An object can extend a class
5. An object can extend an object
6. An object can extend a trait
7. A trait can extend a class
8. A trait can extend an object
9. A trait can extend a trait

12.3 Sealed traits

Suppose you want to represent all the suits in a deck of playing cards – the ones you usually use for a game of Poker: ♣ Clubs, ♦ Diamonds, ♥ Hearts, ♠ Spades. You need to express the general concept of a symbol, but you also need to make sure that there are precisely four of its implementations. You can achieve this by using a “sealed” trait:

Listing 12.5: Suits of Poker Cards

```
sealed trait Suit

object Clubs extends Suit
object Diamonds extends Suit
object Hearts extends Suit
object Spades extends Suit
```

Use a “sealed” trait to limit the elements that extend it. When using the keyword `sealed`, you inform the compiler that all the components that extend the trait are in the same file where the

interface is declared. Thanks to the keyword `sealed`, the compiler knows all the possible implementations for a given trait, and it can warn you if your code forgets to consider all of them – more on this when you are going to learn about pattern matching. In other words, the compiler can infer that a suit is either clubs or diamonds or hearts or spades. You can also refer to this concept of having a limited number of possible implementation as “union type” or “co-product”. Figure 12.3 provides a syntax diagram for sealed traits in Scala.

```

sealed trait name {
    // your code here
}

```

The code snippet shows the definition of a sealed trait named `name`. A handwritten note above the first line says *The name of the sealed trait to create*. An arrow points from this note to the `name` identifier. Below the code, a handwritten note says *- You must declare all its implementations into the same file*.

Figure 12.3: Syntax diagram of how to define a sealed trait in Scala. You must declare all the implementations of a sealed trait in that same file.

QUICK CHECK 12.3

Define a trait, called `Currency`, with only three possible implementations: USD, CAD, and EUR.

12.4 Enumeration in Scala 3

Let’s consider your implementation of all the suits in a deck of playing cards. Listing 12.6 shows you how you can implement this more concisely in Scala 3 using enumeration:

Listing 12.6: Suits of Poker Cards using enumeration

```

enum Suit {
    case Clubs, Diamonds, Hearts, Spades
}

```

You can now refer to a heart’s suit with the expression `Suit.Hearts`. You can also perform other operations on it, such as list all its implementations or pick an implementation based on their declaration order:

```

scala> Suit.Hearts
val res0: Suit = Hearts

scala> Suit.values
val res1: Array[Suit] = Array(Clubs, Diamonds, Hearts, Spades)

scala> Suit.fromOrdinal(0)
val res2: Suit = Clubs

```

Scala 3 introduces the keyword `enum` to define an enumeration type. You can then provide implementations for your enumeration using the keyword `case`. You can also have parameters when defining an enumeration type:

Listing 12.7: A list of countries and codes

```
enum Country(val code: String) { ①
    case Italy extends Country("IT")
    case UnitedKingdom extends Country("UK")
    case UnitedStates extends Country("US")
    case Japan extends Country("JP")
}
```

① Making code a value using the keyword `val` to make it accessible

You can then refer to Italy's code using the expression `Country.Italy.code`.

Have a look at figure 12.4 for a summary of how to use enumeration in Scala 3.

```
enum name {
    case implementation ...
    case implementation
}
```

Figure 12.4: Syntax Diagrams on how to define enumeration in Scala 3. The keyword `enum` identifies an enumeration type. It can have one or more implementations, which the keyword `case` identifies.

QUICK CHECK 12.4

In Quick Check 12.3, you have implemented a sealed trait `Currency` with three implementations: re-implement it using enumeration syntax for Scala 3.

12.5 Summary

In this lesson, my objective was to teach you about traits and enumeration in Scala.

- You have discovered how to define interfaces and implement classes, objects, and traits that conform to those interfaces.
- You have learned about sealed traits: they can express the concept of union types, which are types defined by the finite set of their possible values.
- Finally, you have seen how to define enumeration in Scala 3.

Let's see if you got this!

TRY THIS

Implement an interface to ensure the presence of a field color. Define two classes that conform to your interface: one to represent furniture, the other clothes.

12.6 Answers to Quick Checks

QUICK CHECK 12.1

The implementation of the trait `Printable` is the following:

```
trait Printable {
    def print(): Unit
}
```

QUICK CHECK 12.2

The answers are the following:

1. True – A class can extend a class
2. False – A class can extend an object
3. True – A class can extend a trait
4. True – An object can extend a class
5. False – An object can extend an object
6. True – An object can extend a trait
7. False – A trait can extend a class
8. False – A trait can extend an object
9. True – A trait can extend a trait

QUICK CHECK 12.3

You can implement the trait `Currency` and its implementations as follows:

```
sealed trait Currency
object USD extends Currency
object CAD extends Currency
object EUR extends Currency
```

QUICK CHECK 12.4

Your implementation of the enumeration `Currency` should look similar to the following:

```
enum Currency {
    case USD, CAD, EUR
}
```

13

What time is it?

In this lesson, you will:

- Create a package for your application
- Import and use the `java.time` package
- Code using classes, objects, and traits together with access modifiers
- Define an executable object

Timezones can be challenging to deal with, particularly in distributed teams and collaborations. In this capstone, you'll implement an executable application that uses sbt to prints the current time in a given timezone.

13.1 What time is it?

The goal of this capstone is to create a small program using sbt: it should start, ask the user to enter a timezone, use the input to compute the current time in it, and display the result in the terminal in a human-readable format, such as RFC 1123 (e.g., "Fri, 27 Apr 2018 11:44:35 +0200"). Let's keep things simple and let the script crash if the user enters an invalid timezone – you'll learn how to handle exceptions in the next unit.

13.1.1 Sbt Project Setup

First, you need to set an sbt project up. There are different ways of achieving this: you can create an sbt project using your IDE or apply the hello-world giter8 template by typing the command `sbt new scala/hello-world.g8`. When using the hello-world giter8 template – when doing so, do not forget to delete the `Main.scala` file in the `src/main/scala` folder. Alternatively, you can also create an empty sbt project manually as follows:

1. In your project directory, create a `build.sbt` file containing your scala version, the name, and your project version: listing 13.1 provides an example.

Listing 13.1: An example of build.sbt file

```
name := "what-time-is-it"
version := "0.1"
scalaVersion := "2.13.4" // your scala version here, for example "2.13.4"
```

2. Create a directory `project` with a `build.properties` file (see listing 13.2) with your sbt version.

Listing 13.2: An example of project/build.properties file

```
sbt.version = 1.4.7 // your sbt version here, for example 1.4.7
```

3. Finally, create the folder `src/main/scala`: it will contain your application's source code.

You will not need to add any external dependency into your `build.sbt` file for this capstone. You'll use `java.time` to compute the current date and time, which is already accessible as an internal module of the Scala language.

Finally, let's define a package, called `org.example.time`, to contain the code for your application: create a directory with relative path `src/main/scala/org/example/time`.

13.1.2 The Business Logic Layer

The `TimePrinter` class takes care of our business logic. It belongs to the `org.example.time` package, which defines the logic used to compute the current date and time for a given timezone. It uses three elements of the package `java.time`:

- `java.time.ZoneId` lists the time zones supported, and it allows you to perform operations on them.
- `java.time.ZonedDateTime` represents a date with time and a well-defined timezone.
- `java.time.format.DateTimeFormatter` converts a temporal event (e.g., date, time, date-time) to and from a string. You can either use one of the already-defined formatters or create a custom one.

A `TimePrinter` has a formatter as its parameter and a publicly accessible method `now`, which takes one parameter representing a timezone as a string: you do not need to expose its other functions. Listing 13.3 shows a possible implementation for it:

Listing 13.3: The TimePrinter class

```
// file src/main/scala/org/example/time/TimePrinter.scala
package org.example.time

import java.time.format.DateTimeFormatter
import java.time.{ZoneId, ZonedDateTime}

class TimePrinter(formatter: DateTimeFormatter) { ①

    def now(timezone: String): String = {
        val dateTime = currentDateTime(timezone)
```

```

    dateTimeToString(dateTime)
}

private def currentTime(timezone: String): ZonedDateTime = {
  val zoneId = ZoneId.of(timezone)
  ZonedDateTime.now(zoneId)
}

private def dateTimeToString(dateTime: ZonedDateTime): String =
  formatter.format(dateTime)
}

```

- ① Your implementation works with any formatter

You are now ready to implement the entry point of your program.

13.1.3 The TimeApp Executable Object

Your next task is to define an executable object, called `TimeApp`, containing the logic to ask a timezone in standard input, compute the timezone and print it in the terminal. Listing 13.4 shows how you can do this:

Listing 13.4: The TimeApp object

```

// file src/main/scala/org/example/time/TimeApp.scala

package org.example.time

import java.time.format.DateTimeFormatter
import scala.io.StdIn

object TimeApp extends App { ①

  val timezone = StdIn.readLine("Give me a timezone: ") ②
  val timePrinter =
    new TimePrinter(DateTimeFormatter.RFC_1123_DATE_TIME) ③
  println(timePrinter.now(timezone)) ④
}

```

- ① `TimeApp` is an executable object because it extends the trait `App`

- ② It requests a string in input from the terminal

- ③ It creates an instance of `TimePrinter`

- ④ It computes and prints the current time in the timezone

Your application is now ready to run.

13.1.4 Let's try it out!

It's time to see your program in action. Navigate to the root folder for your project and execute the command `sbt run` to compile and execute the code. After a few seconds, you should see that the application is waiting for you to provide a timezone:

```

[info] Running org.example.time.TimeApp
Give me a timezone:

```

For example, you can enter the timezone “Asia/Tokyo” and get a result that looks similar to the following:

```
[info] Running org.example.time.TimeApp
Give me a timezone: Asia/Tokyo
Fri, 24 Aug 2020 05:50:31 +0900
```

If you enter an invalid or unrecognized timezone, the script will error with an exception:

```
[error] (run-main-0) java.time.zone.ZoneRulesException: Unknown time-zone ID: invalid
[error] java.time.zone.ZoneRulesException: Unknown time-zone ID: invalid
[error] at java.time.zone.ZoneRulesProvider.getProvider(ZoneRulesProvider.java:272)
[...]
[...] more stack trace here...
[...]
[error] at java.lang.reflect.Method.invoke(Method.java:497)
[error] Nonzero exit code: 1
[error] (Compile / run) Nonzero exit code: 1
```

Play around with your application: can you spot any bugs or non-ideal behaviors? Let’s discuss a few of them in the next section.

13.2 The ugly bits of our solution

Congratulations on completing your capstone! Your implementation respects the requirements, but a few improvements are possible: let’s see some of them and what techniques you’ll learn to overcome them.

ERROR HANDLING

If a user enters an invalid timezone, your application crashes with a nasty and difficult to understand exception. In unit 3, you’ll see how to catch and throw exceptions. You’ll also learn how to create custom exceptions to provide your users with a more descriptive message about the error and its cause.

TIMEZONE REPRESENTATION

Your application is entirely dependent on the `java.time` package and its definition of timezone. For example, it considers “UTC” a valid timezone, while “utc” is invalid. The timezone “Asia/Tokyo” is also not equivalent to “ASIA/TOKYO”. Being dependent on the `java.time` package has two significant implications:

- A change in the `java.time` package drastically affects your application, and it can potentially break it without you realizing it.
- In the future, if you wish to migrate to a different time library, it will be excruciating and painful because your definition of “valid timezone” lives inside the package `java.time`.

In future units, you’ll see how to overcome them thanks to data structures such as Map: they will allow you to have a clear separation between how you represent your data and how you manipulate it.

13.3 Summary

In this capstone, you have implemented an sbt application that asks the user for a string representing a valid timezone, and it prints the current date and time in a given timezone.

- You have created an sbt project, and you have used the `java.time` package to define your business logic.
- You have implemented the entry point of your application, and you have seen it in action.
- Finally, we have discussed possible improvements for your implementation thanks to the techniques you'll see in future units.

Unit 3

HTTP Server

In Unit 2, you have mastered the fundamentals of object-oriented programming in Scala and built an executable sbt application to print the current date and time in the given timezone. In this unit, you'll adapt the code you have written for the previous capstone to transform your application into an HTTP server. You'll define an HTTP API to return the current date and time for a given country, rather than a given timezone, using `http4s`, a popular library to manage HTTP requests and responses. In particular, you'll learn about the following subjects:

- Lesson 14 teaches you about pattern matching, a powerful and useful tool to combine your program's different execution flows based on some predefined condition.
- Lesson 15 shows you what anonymous functions are and how to define them more quickly and concisely.
- Lesson 16 introduces partial functions as a type of anonymous function defined only for some input values. You'll use partial functions when implementing the routing of your HTTP server.
- Lesson 17 illustrates how to use the library `http4s` to handle GET HTTP requests and responses.
- Finally, you will apply these concepts and implement an HTTP server to return the current date and time of a given country in lesson 18.

Once you have learned how to implement and run an HTTP server, you'll continue your journey with Unit 4, in which you'll see how to represent your data using immutable structures.

14

Pattern Matching

After reading this lesson, you'll be able to:

- Write code that uses pattern matching as an alternative to an if-else construct.
- Pattern match over a closed set of values.

After discovering how to express interfaces in Scala using a trait, you'll see how to use pattern matching. You can use pattern matching as an alternative to an if-else construct, particularly useful when having many different condition branches. The Scala pattern matching looks similar to the switch/case statement of other languages, such as Java, JavaScript, and C++. It is often more powerful and versatile than in other languages, thanks to its expressive syntax and dedicated support for sealed elements and classes. In this lesson, you'll learn how pattern matching compares to an if-else construct and its base uses. You'll also discover how to define pattern matching over a sealed set of values and how the compiler can warn you if you forget to consider any of them. Pattern matching also has dedicated support for classes, but you'll see how this works later in the book when learning about case classes. In the capstone, you'll use pattern matching to map country names to their timezones.

Consider this

Suppose you need to write a function that takes an integer as the parameter, and it returns a string representing its corresponding month of the year. If you implement it using an if-else construct, how many condition branches (i.e., if/else if/else expressions) you'll have to define? Can you think of any alternative and more maintainable solutions?

14.1 If-else construct versus Pattern Matching

Suppose you have to write a function to convert a number into its corresponding day of the week, starting with 1 for Sunday. You could use an if-else construct and implement it as follows:

Listing 14.1: Converting number to a day of the week

```
def dayOfWeek(n: Int): String =
  if (n == 1) "Sunday"
  else if (n == 2) "Monday"
  else if (n == 3) "Tuesday"
  else if (n == 4) "Wednesday"
  else if (n == 5) "Thursday"
  else if (n == 6) "Friday"
  else if (n == 7) "Saturday"
  else "Unknown"
```

Although this program works as expected, it is not particularly easy to read because of the long list of cases to consider. Its predicate condition always has the same structure: “is `n` equal to some number?”. You are trying to express a one-to-one mapping between numbers and days of the week: pattern matching is a good fit for some refactoring that makes your code more readable and concise. Listing 14.2 shows you how to rewrite the function using pattern matching:

Listing 14.2: Mapping numbers with days of the week

```
def dayOfWeek(n: Int): String = n match {
  case 1 => "Sunday"
  case 2 => "Monday"
  case 3 => "Tuesday"
  case 4 => "Wednesday"
  case 5 => "Thursday"
  case 6 => "Friday"
  case 7 => "Saturday"
  case _ => "Unknown" ①
}
```

① the default case

You can visualize pattern matching as an expression with a collection of cases: you can identify it by an expression with a `match` keyword and one or more `case` keyword clauses. It considers each case in order of declaration: once it can successfully match a case, it continues the computation by evaluating the corresponding expression.

At first, pattern matching may seem similar to the `switch/case` statement of other languages like Java or C++. However, it is a lot more powerful because of its expressive syntax and dedicated support for several types, such as sealed traits and case classes.

Imagine you want to write a function that takes a parameter of any type and returns a string depending on its type:

- if its argument is a positive integer or a double, you should return a descriptive string of the represented number;
- if it’s the string “pong”, you should reply with the string “pong”;
- if it’s a generic string, you should return a default message;
- if none of the previous cases matches, you should return its default standard string representation using the function `toString` – any class instance has an implementation for it!

Listing 14.3 illustrate how you can translate this with pattern matching:

Listing 14.3: Implementing objInfo

```
def objInfo(param: Any) = param match { ①
    case n: Int if n > 0 => s"$n is a positive integer" ②
    case d: Double => s"$d is a double" ③
    case "ping" => "pong" ④
    case _: String => "you gave me a string" ⑤
    case obj => obj.toString ⑥
}
```

- ① Any is the root of the class hierarchy in Scala: param can be of whatever type.
- ② it matches an integer bigger than zero
- ③ it matches any double
- ④ it matches the string “ping”
- ⑤ it matches any string. You can use an underscore when you do not need to bind the value to a name.
- ⑥ it matches any instance: as per Java tradition, any instance always has an implementation for the `toString` function.

The function `objInfo` may not seem extremely useful, but it gives you an example of all the matches that pattern matching can support. Have a look at figure 14.1 for a summary of the different use pattern matching cases you have seen so far.

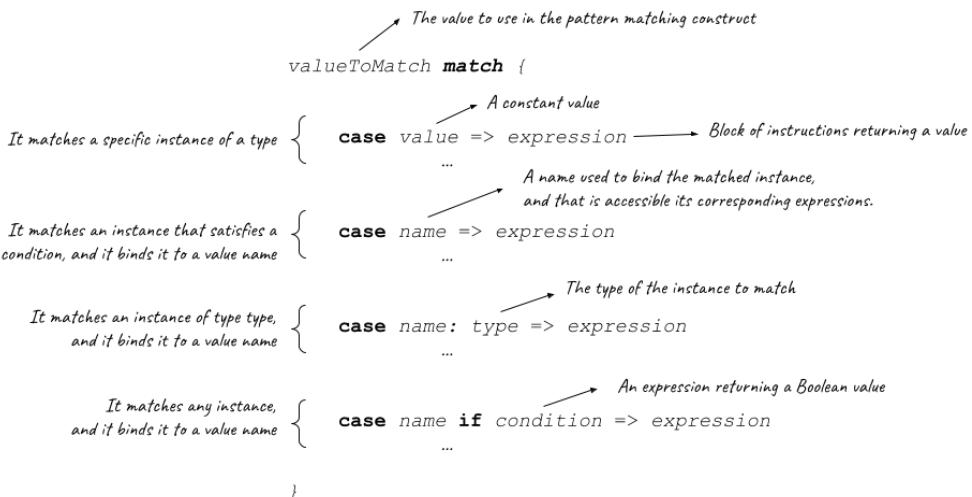


Figure 14.1: Syntax diagram on pattern matching in Scala. When pattern matching, you need to declare at least one case condition.

The many uses of underscore

In lesson 9, you have learned about imports: you have used the underscore symbol to import all the package components or a specific code element.

In this lesson, you have now encountered another usage of the symbol underscore: you can use it to discard value names. You will soon realize that this symbol has lots of means in Scala! But do not be concerned: you'll discover them gradually and become familiar with them relatively quickly.

As a rule of thumb, you state to the compiler that the details are not relevant every time you use an underscore in your code: it will act accordingly.

In listing 14.2, the underscore indicates that you do not care about the value of the parameter `n`, so the case should always match. In listing 14.3, because you do not refer to the string matched in the case, you can ask the interpreter not to bind it to a value and discard it.

QUICK CHECK 14.1:

Consider the function `objInfo` shown in listing 14.3. Guess the type and value that the function returns for the following inputs. Use the REPL to validate your hypotheses.

- `objInfo(-1)`
- `objInfo(true)`
- `objInfo(200)`
- `objInfo(200.00)`
- `objInfo("ping")`

14.2 Sealed Pattern Matching

In lesson 12, you have discovered how the use of the keyword `sealed` can allow you to define a closed set of possible implementations for a trait: this is particularly useful when combined with pattern matching. Suppose you have the following values to represent several currencies:

```
sealed trait Currency

object USD extends Currency
object GBP extends Currency
object EUR extends Currency
```

You need to define a function, called `exchangeRateUSD`, to return the USD exchange rate for a given currency. For example, you could define a function as follows:

Listing 14.4: USD Exchange Rate

```
def exchangeRateUSD(currency: Currency): Double =
  currency match {
    case USD => 1
    case GBP => 0.744
    case EUR => 0.848
  }
```

What happens if you add a new type of currency and forget to update your `exchangeRateUSD` function accordingly? Let's try it out!

Let's start the REPL and define the existing currencies together with a new one for Canadian Dollars, called CAD:

```
$ scala
// ... output omitted
scala> sealed trait Currency
```

```

| object USD extends Currency
| object GBP extends Currency
| object EUR extends Currency
| object CAD extends Currency
// defined trait Currency
// defined object USD
// defined object GBP
// defined object EUR
// defined object CAD

```

If you define an `exchangeRateUSD` function that doesn't consider Canadian Dollars, the compiler will provide you with a warning to inform you which implementations of `Currency` you haven't considered:

```

scala> def exchangeRateUSD(currency: Currency): Double =
|   currency match {
|     case USD => 1
|     case GBP => 0.744
|     case EUR => 0.848
|   }
<console>:16: warning: match may not be exhaustive.
It would fail on the following input: CAD
      currency match {
      ^
exchangeRateUSD: (currency: Currency)Double

```

If your input doesn't match any of the case clauses, the compiler throws a `MatchError` exception:

```

scala> exchangeRateUSD(CAD)
scala.MatchError: CAD$@30437e9c (of class CAD$)
  at .exchangeRateUSD(<console>:19)
... 28 elided

```

These compiler warnings highlight issues in your code that can potentially be extremely dangerous: do not ignore them! You can ask the compiler to consider warnings as compilation errors by enabling a feature flag. To do so, add the following line to your `build.sbt` file:

```
scalacOptions += "-Xfatal-warnings"
```

QUICK CHECK 14.2:

You now have four possible implementations for the sealed trait `Currency`. Fix the function `exchangeRateUSD` so that you no longer see the match-not-exhaustive warning when compiling your code.

14.3 Summary

In this lesson, my objective was to teach you about pattern matching.

- You have discovered that using if-else is not always the best option, particularly when many condition branches are involved: you should consider using pattern matching instead.
- You have learned that pattern matching is more than just an alternative if-else construct: you can match complex conditions that consider many aspects of an expression, such as its value, type, and custom predicates.

- When pattern matching over sealed values, the compiler will warn you if you forget to consider any of them.

Let's see if you got this!

TRY THIS

In lesson 5, you have written a function to apply the discount to a given price as follows:

- 0% discount if the price is less than \$50
- 10% discount if the price is at least \$50 but less than \$100
- 15% discount if the price is at least \$100

Re-implement it using pattern matching instead of an if-else construct.

14.4 Answers to quick checks

QUICK CHECK 14.1:

All the returned values are of type `String`. The return types and values for the function `objInfo` are following:

- `objInfo(-1)` returns "-1"
- `objInfo(true)` returns "true"
- `objInfo(200)` returns "200 is a positive integer"
- `objInfo(200.00)` returns "200.0 is a double"
- `objInfo("ping")` returns "pong"

QUICK CHECK 14.2:

The warning disappears as soon as the function `exchangeRateUSD` has a case for Canadian Dollars. For example, you could change it as follows:

```
def exchangeRateUSD(currency: Currency): Double =
  currency match {
    case USD => 1
    case GBP => 0.744
    case EUR => 0.848
    case CAD => 1.278
  }
```

15

Anonymous Functions

After reading this lesson, you will be able to:

- Implement anonymous functions
- Code using the concise notation for anonymous functions

In lesson 6, you have learned the basics of functions in Scala. In this lesson, you'll discover a new type of function called "anonymous". Anonymous functions are functions that you can define quickly and concisely. At first, they may seem just an alternative to the standard Scala functions you have seen so far, but you'll soon discover that they are particularly handy when combined with another type of function, called "higher order". The concept of anonymous function is not unique to Scala: other languages, such as Java 8+, and Python, refer to it as "lambda". In the capstone, you will use a particular kind of anonymous function called "partial" to define your HTTP server's routes.

Consider this

Consider the following two functions to sum and subtract two integers:

```
def sum(a: Int, b: Int): Int = a + b
```

```
def subtract(a: Int, b: Int): Int = a - b
```

Which parts these two functions have in common? Which ones are not? Can you think of a more concise way of achieving the same implementation?

15.1 Function versus Anonymous Function

Suppose you have implemented a calculator program to perform the standard operations on integers (i.e., sum, subtraction, multiplication, division) together with negation. Listing 15.1 shows a possible implementation:

Listing 15.1: MyCalculator Program

```
object MyCalculator {

    def sum(a: Int, b: Int): Int = a + b

    def subtract(a: Int, b: Int): Int = a - b

    def multiply(a: Int, b: Int): Int = a * b

    def divide(a: Int, b: Int): Int = a / b

    def negate(a: Int): Int = subtract(0, a) ①

}
```

① Alternatively, you can also multiply by minus one.

You can use your `MyCalculator` program as follows:

```
import MyCalculator._

sum(3, 5)           // returns 8
subtract(4, 4) // returns 0
multiply(5, 3) // returns 15
divide(6, 2)        // returns 3
negate(-5)          // returns 5
```

In Scala, every function has a type: you can represent this by combining its parameters with the arrow "`=>`" and its return type. For example, let's have another look at listing 15.1. The function `sum` has the type `(Int, Int) => Int` – this reads "Int Int to Int" because it takes two integers as parameters and returns an integer. On the other hand, the function `negate` has type `Int => Int` – this reads "Int to Int" – because it takes an integer as the parameter and returns an integer.

The type notation for functions reflects the syntax used to implement anonymous functions. For example, you can implement the equivalent anonymous functions for `sum` and `negate` as follows:

```
def sum(a: Int, b: Int): Int = a + b      // function for sum
{ (a: Int, b: Int) => a + b }             // anonymous function for sum

def negate(a: Int): Int = subtract(0, a) // function for negate
{ (a: Int) => subtract(0, a) }          // anonymous function for negate
```

When implementing anonymous functions, the function name is no longer needed, while its parameters and body stay the same. Its return type also disappears because the compiler infers it from its implementation. Have a look at figure 15.1 for a syntax summary of how to define anonymous functions.

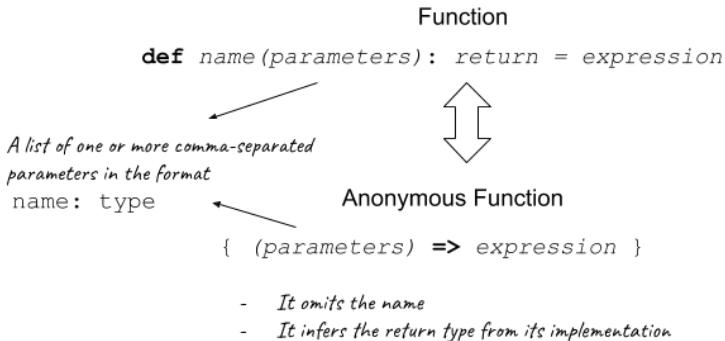


Figure 15.1: Comparison between the syntax for a function and its corresponding anonymous function.

Listing 15.2 shows how to re-implement your calculation program using anonymous functions:

Listing 15.2: MySecondCalculator Program

```
object MySecondCalculator {
    val sum = { (a: Int, b: Int) => a + b }
    val subtract = { (a: Int, b: Int) => a - b }
    val multiply = { (a: Int, b: Int) => a * b }
    val divide = { (a: Int, b: Int) => a / b }
    val negate = { a: Int => subtract(0, a) } ①
}
```

① you can omit the round parenthesis for “a: Int” because you only have one function parameter.

In the function `negate`, you can omit the parenthesis for the parameter `a`: you can do this when an anonymous function accepts only one parameter. Note that you have re-used the function names in listing 15.1 to create values that refer to the corresponding anonymous function: this is an optional step that allows you to call the function later on. You can use your `MySecondCalculator` program by calling the values you have defined and by providing parameters as if they were regular functions:

```
import MySecondCalculator._

sum(3, 5)
subtract(4, 4)
multiply(5, 3)
divide(6, 2)
negate(-5)
```

QUICK CHECK 15.1

What is the type for each of the values defined in listing 15.2? Use the REPL to validate your hypothesis.

QUICK CHECK 15.2

Write an anonymous function equivalent to the following function:

```
def hello(n: String): String = s"Hello, $n!"
```

15.2 Concise notation for Anonymous Functions

Scala offers a more concise notation for anonymous functions: let's see how it works.

When transforming a function to an anonymous function its return type is no longer needed because the compiler infers its type: you can do the same for the parameter type. If you provide the compiler an explicit type for your anonymous function, it will use it to infer the type of your parameters. For example, you can refactor the anonymous functions `sum` and `negate` shown in listing 15.2 as follows:

```
val sum = { (a: Int, b: Int) => a + b }           // before
val sum: (Int, Int) => Int = { (a, b) => a + b } // after

val negate = { a: Int => subtract(0, a) }         // before
val negate: Int => Int = { a => subtract(0, a) } // after
```

If your anonymous function has an implementation that consists of a single instruction and your parameters are used in order of declaration, you can even go a step further by removing the parameters completely and replacing them with an underscore:

```
val sum = { (a: Int, b: Int) => a + b }           // before
val sum: (Int, Int) => Int = { (a, b) => a + b } // first refactoring
val sum: (Int, Int) => Int = { _ + _ }           // second refactoring

val negate = { a: Int => subtract(0, a) }         // before
val negate: Int => Int = { a => subtract(0, a) } // first refactoring
val negate: Int => Int = { subtract(0, _) }       // second refactoring
```

You can now refactor your calculator program using this more concise notation:

Listing 15.3: MyThirdCalculator Program

```
object MyThirdCalculator {

    val sum: (Int, Int) => Int = { _ + _ } ①
    val subtract: (Int, Int) => Int = { _ - _ } ①
    val multiply: (Int, Int) => Int = { _ * _ } ①
    val divide: (Int, Int) => Int = { _ / _ } ①
    val negate: Int => Int = subtract(0, _) ②

}
```

- ① You could omit the curly brackets here (e.g., use “`_ + _`” instead of “`{ _ + _ }`”)
 ② Curly brackets omitted

Readability first!

In future lessons and units, you’ll discover how useful and expressive the concise notation for anonymous functions is, in particular when combined with higher order functions.

Depending on the context you are in, this notation can become quite cryptic and hard to read due to all the information removed and inferred at compile time instead. For example, you could argue that the expression “`_ - _`” is less expressive and more confusing than “`{ (a, b) => a - b }`”.

Do not compromise the readability of your code. When using the concise notation for anonymous functions, use it only when the omitted information is easy to infer for both the compiler and your fellow developers.

QUICK CHECK 15.3

Are functions `funcA` and `funcB` equivalent? In other words, do they return the same output when receiving the same input? Why? Use the REPL to verify your hypotheses.

```
val funcA: (Int, Int) => Int = { (a, b) => b / a }
val funcB: (Int, Int) => Int = { _ / _ }
```

15.3 Summary

In this lesson, my objective was to teach you about anonymous functions.

- You can use them to create functions on the fly and without too much boilerplate: you are going to see their full potential when you’ll learn about higher order functions.
- You have also discovered their concise notation to remove unnecessary information that the compiler infers from the function’s type.

Let’s see if you got this!

TRY THIS

Rewrite each of the following functions as anonymous functions: use your concise notation at your discretion.

```

1. def multiply(s: String, n: Int): Int = s.length * n
2. def toDouble(n: Int): Double = n.toDouble
3. def concat(s1: String, s2: String): String = s1 + s2
4. def inverseConcat(s1: String, s2: String): String = s2 + s1
5. def myLongFunc(s: String): String = {
    val length = s.length
    s.reverse * length
}

```

15.4 Answers to Quick Checks

QUICK CHECK 15.1

The values `sum`, `subtract`, `multiply`, `divide` have the type `(Int, Int) => Int`, while the value `negate` has type `Int => Int`.

QUICK CHECK 15.2

An implementation of an anonymous function equivalent to the function `hello` is the following:

```
{ n: String => s"Hello, $n!" }
```

QUICK CHECK 15.3

Functions `funcA` and `funcB` are not equivalent because of the order of their parameters. Function `funcA` divides its second parameter called `b` by its first parameter called `a`. Function `funcB` does the inverse: it divides its first parameter by its second one because the compiler substitutes them by following their declaration order.

16

Partial Functions

After reading this lesson, you will be able to:

- Implement partial functions to abstract commonalities between functions
- Create new functions by composing partial functions
- Use a try-catch expression to handle exceptions

After learning about pattern matching, you'll discover partial functions and how they relate to pattern matching in this lesson. Partial functions are functions that are defined only for some input. You'll see how they can be useful to abstract commonalities between functions and how you can compose them to create more complex functionalities. Finally, you'll see how you can use partial functions to catch and handle exceptions. In the capstone, you'll use partial functions to define the routes of your HTTP server.

Consider this

Suppose you have two pattern matching constructs: they look the same except the last case clause. How would you refactor them so that you can avoid code repetition?

16.1 Partial Functions

Suppose you need to compute operations on integers. In particular, you want to calculate the square root of an integer. But this operation is defined only for non-negative numbers. When dealing with a negative integer, you need to either return the negative value or return zero, depending on your use case. You could use pattern matching and define two functions as follows:

Listing 16.1: Square root of an integer functions

```
def sqrtOrZero(n: Int): Double = n match {
  case x if x >= 0 => Math.sqrt(x)
  case _ => ①
}

def sqrtOrValue(n: Int): Double = n match {
  case x if x >= 0 => Math.sqrt(x)
  case x => x
}
```

① The compiler converts the integer into a double because the function has `Double` as its return type.

The functions `sqrtOrZero` and `sqrtOrValue` are very similar: their only difference is the last case clause of their pattern matching constructs. Code duplication makes code difficult to maintain and keep consistent: let's see how you can avoid repeating yourself using partial functions.

16.1.1 Implementing a Partial Function

A partial function is a function that you can define only for some instances of a type. You have already encountered examples of partial functions when discussing pattern matching: you can consider each case clause as a partial function. For example, you can define a partial function to compute the square root of a non-negative integer as follows:

Listing 16.2: The `sqrt` partial function

```
val sqrt: PartialFunction[Int, Double] = ①
  { case x if x >= 0 => Math.sqrt(x) }
```

① You can read `PartialFunction[Int, Double]` as "partial function from int to double".

You can view partial functions as a particular anonymous function with one or more case clauses as their body. The compiler cannot infer their type, so you need to specify its argument and return types: in Scala, the type `PartialFunction[A, B]` identifies a partial function that a parameter of type `A` and that returns an instance of type `B`.

Listing 16.3 shows another example of partial function:

Listing 16.3: A `toPrettyString` function

```
val toPrettyString: PartialFunction[Any, String] = { ①
  case x: Int if x > 0 => s"positive number: $x"
  case s: String => s
}
```

① Partial function for any type to string

After you define a partial function, you can perform function calls using the same syntax you have learned for anonymous functions. But do not forget that your function is now partial: if your input doesn't match any of its case clause, you will receive a `MatchError` exception at runtime:

```
scala> val toPrettyString: PartialFunction[Any, String] = {
    |   case x: Int if x > 0 => s"positive number: $x"
    |   case s: String => s
```

```

    |
    }

val toPrettyString: PartialFunction[Any, String] = <function1>

scala> toPrettyString(1)
res0: String = positive number: 1

scala> toPrettyString("hello")
res1: String = hello

scala> toPrettyString(-1)
scala.MatchError: -1 (of class java.lang.Integer)
  at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:255)
  at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:253)
  at $anonfun$1.applyOrElse(<pastie>:14)
  at scala.runtime.AbstractPartialFunction.apply(AbstractPartialFunction.scala:34)
... 28 elided

```

Figure 16.1 summarizes how to implement partial functions in Scala.

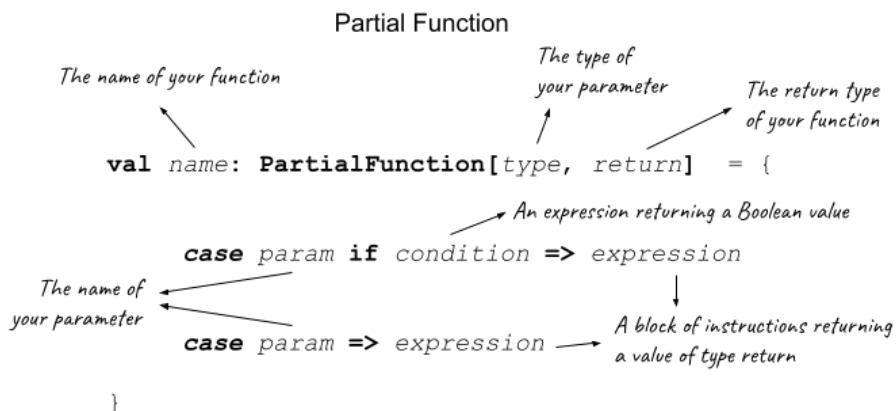


Figure 16.1: Syntax diagram on how to implement partial functions in Scala. Ensure to specify your partial function's type using `PartialFunction[A, B]:A` is the type of your parameter, while `B` is its return type.

QUICK CHECK 16.1

Define a partial function called `transform` that reverses strings starting with an 'a' and converts to uppercase all those beginning with an 's'. Use the `startsWith`, `reverse`, and `toUpperCase` functions of the class `String`.

16.1.2 Function composition

When talking about “composition of functions”, you usually refer to chaining two functions together by passing the result of the first function as the parameter to the second one. For example, consider the following two function:

```
val f: String => Int = _.size
val g: Int => Boolean = _ > 2
```

You can create a new function by calling f and then g:

```
val gof: String => Boolean = { s => g(f(s)) }
```

Alternatively, you can use the `andThen` function as follows:

```
val gof: String => Boolean = f.andThen(g)
```

The concept of composition may have a different meaning in the context of partial functions. Instead of chaining functions together, you may want to combine partial functions as fallbacks if the previous partial function couldn't match the given input. In Scala, you can compose partial functions as fallbacks using the function `orElse`. Listing 16.4 shows you how you can refactor your square root functions to remove the code duplication:

Listing 16.4: Another two square root functions

```
val sqrt: PartialFunction[Int, Double] =
{ case x if x >= 0 => Math.sqrt(x)}

val zero: PartialFunction[Int, Double] = { case _ => 0 }

val value: PartialFunction[Int, Double] = { case x => x }

def sqrtOrZero(n: Int): Double = sqrt.orElse(zero)(n) ①

def sqrtOrValue(n: Int): Double = sqrt.orElse(value)(n) ①
```

① First, you create a new function using `orElse`. Then, you apply the parameter to it.

QUICK CHECK 16.2

Consider the partial functions `sqrt` and `zero` in listing 16.4: is the partial function `sqrt.orElse(zero)` equivalent to `zero.orElse(sqrt)`?

16.2 Use case: Exception Handling

You have first encountered partial functions in lesson 14 when discussing pattern matching. You are now ready to discover another popular use of partial functions: the try-catch expression.

Scala's exception handling comes directly from the Java world. In Scala, any class that extends `java.lang.Exception` is an exception. An exception interrupts your code's execution flow: you need to intercept it, or it will terminate your program.

The `java.lang` package offers a few types of exceptions ready for you to use. A few of them are `RuntimeException`, `NullPointerException`, `IllegalStateException`, `IllegalArgumentException`, `NumberFormatException`. Alternatively, you can define a custom exception by extending the `java.lang.Exception` class:

```
scala> class MyException(msg: String) extends Exception(msg)
defined class MyException
```

You can use an instance of an exception to interrupt your program using the keyword `throw`:

```
scala> throw new MyException("BOOM!")
MyException: BOOM!
... 28 elided
```

After raising an exception, you'll need to catch it before it terminates your program. To do so, you can use a try-catch expression as follows:

Listing 16.5: An Exception Handling Example

```
def n(): Int =
  try {
    throw new Exception("BOOM!")
    42
  } catch {
    case ex: Exception =>
      println(s"Ignoring exception $ex. Returning zero instead")
      0
  }
```

When evaluating `n`, the console will display a message and return the integer zero:

```
scala> n()
Ignoring exception java.lang.Exception: BOOM!. Returning zero instead
val res0: Int = 0
```

The `catch` keyword follows partial functions that identify which exceptions it should handle: if an exception doesn't match, it will not intercept it.

Think in Scala: Avoid Exceptions

Scala reuses Java code that heavily uses exceptions. Some of its standard libraries and functions also throw them: knowing how to deal with them is crucial. However, you should avoid using exceptions in your code.

Exceptions are the equivalent of ticking bombs: they will explode and kill the whole program unless someone is ready to defuse them!

Exceptions are unpredictable. Identifying which exceptions a function could throw is particularly challenging. Its signature doesn't give you this information. You could document and annotate which exceptions it could throw, but this is not controlled or enforced at compile-time, so you have no guarantee that they are still accurate or correct. Often, your only option is to look at its implementation while hunting for exceptions, but this is not an easy task as they can hide in any of its inner function calls.

Exceptions are a drastic solution. If you do not catch an exception, it will terminate your program. Unless you are writing a simple script, your application's crash is probably not the behavior you desire.

In future lessons, you'll see how you can represent possible errors using types (i.e., the absence of a value using the type `Option`). In doing so, you will have specific information on the possible errors just by looking at your function signature. The compiler will also make sure that you handle them correctly by considering both the positive and negative case scenarios.

Partial functions can be dangerous: they will throw an exception if your input doesn't match any case. You should ensure to convert all their possible inputs when composing them. When introducing `Option`, you'll see how to rewrite any partial function as a total function returning an instance of `Option` and avoid the inconvenience of dealing with a `MatchError` exception.

QUICK CHECK 16.3

The following expression throws an `IllegalArgumentException` exception:

```
val b = "hello".toBoolean
```

Write a try-catch expression to default any non-parsable value to `false`.

16.3 Summary

In this lesson, my objective was to teach you about partial functions.

- You have discovered how you can implement partial functions and compose them to abstract commonalities between functions.
- You have also learned how to use partial functions in a try-catch expression to handle exceptions.

Let's see if you got this!

TRY THIS

Implement a function to parse a string into an integer. If you cannot parse it, return its length instead.

HINT: Use the `toInt` function on an instance of `String`.

16.4 Answers to Quick Checks

QUICK CHECK 16.1

A possible implementation for the function transform is following:

```
val transform: PartialFunction[String, String] = {
  case s if s.startsWith("a") => s.reverse
  case s if s.startsWith("s") => s.toUpperCase
}
```

QUICK CHECK 16.2

The two partial functions are not equivalent because of the different composition order: `zero.orElse(sqrt)` returns zero for any input.

```
sqrt.orElse(zero)(4)      // returns 2.0
zero.orElse(sqrt)(4)      // returns 0.0
```

QUICK CHECK 16.3

You could change the expression `val b = "hello".toBoolean` as follows:

```
val b = try {
  "hello".toBoolean
} catch {
  case _: IllegalArgumentException => false
}
```

17

HTTP API with http4s

After reading this lesson, you will be able to:

- Run an HTTP server using sbt
- Implement an API to handle GET requests

After learning about partial functions, you'll use them as part of your implementation of an HTTP server. Building an HTTP server without the help of an external library would require lots of extra time and code. Thankfully, the Scala ecosystem offers a few external libraries to help you handle HTTP communication in an efficient and performant way. In this lesson, you'll learn about `http4s`, a popular library to manage HTTP requests and responses. You'll discover how to implement an HTTP server that replies to a GET "/ping" request with a response with status code "200 - Ok" and the text "pong". Finally, you'll see how to run it using sbt. In the capstone, you'll use `http4s` to create an HTTP server and define its API.

Consider this

Suppose you are building an HTTP server that provides a REST API for retrieving and storing data. What are the main components of your server? How would you structure your code?

17.1 An overview of http4s

Typelevel is a non-profit organization to promote purely functional open-source Scala projects together with an inclusive and welcoming environment: for more information, visit their website at typelevel.org. `http4s` is a Typelevel project and one of the most popular Scala libraries to handle HTTP requests and responses. It is particularly accessible to newcomers to the language thanks to its extensive documentation and numerous examples: you can find them at http4s.org.

Even though `http4s` does offer a `giter8` template (have a look at the sidebar for more information), you'll see how to build an HTTP server from scratch in this lesson.

A giter8 template for http4s

The library `http4s` offers a `giter8` template to generate a simple HTTP server that replies to an `/hello/world` endpoint, an ideal skeleton for any new project that deals with an HTTP API. To apply the template, type the following `sbt` command:

```
$ sbt new http4s/http4s.g8
```

The terminal prompt will ask you to provide some information about your project, such as its name, organization, and package: when in doubt, choose the provided default value.

After applying the template, navigate to the newly created folder and execute the command `sbt run` to start the HTTP server. You can now send HTTP requests to it:

```
$ curl -i http://localhost:8080/hello/scala
```

HTTP/1.1 200 OK

Content-Type: application/json

Date: Tue, 29 Dec 2020 15:24:52 GMT

Content-Length: 26

```
{"message": "Hello, scala"}
```

Have a pick at its code: it should look reasonably similar to the one you'll see in this lesson. It also provides examples of testing an HTTP application and deserialize from JSON – a topic you'll master in unit 8.

Before explaining how to implement an HTTP server using `http4s`, let's overview its architecture. First, you need to link your routes to their business logic through instances of `org.http4s.HttpRoutes`. Each `HttpRoutes` uses partial functions to match an incoming HTTP request, and it produces an HTTP response together with a side effect (e.g., an IO read/write, a connection to a third party). At this point, you may not be familiar with the concept of side effects, but do not worry as you'll learn what they are in the next unit. Most applications use `cats.effect.IO` as a generic representation of both synchronous and asynchronous side effects: in unit 8, you'll learn about side effects and why this is crucial. Finally, you define a singleton object that extends `cats.effect.IOApp` and provide instructions on the port and host to bind and services to mount. Your executable object also uses `Blaze` (see <https://github.com/http4s/blaze>) together with streams as the backend for network IO. Figure 17.1 provides a visual summary of the components of an `http4s` application.

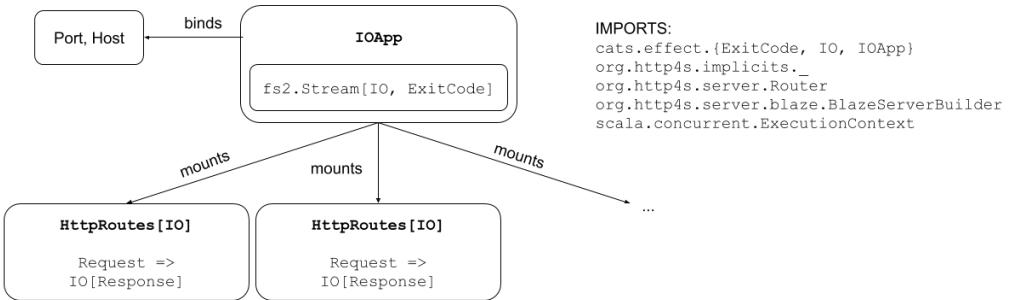


Figure 17.1: Summary of the main components of an http4s application. An `HttpRoutes[IO]` matches a request, and it produces a response wrapped around an `IO` instance to represent possible side effects. Your executable object extends `IOApp` and uses a `BlazeServerBuilder` to bind to a given port and host and to mount multiple instances of `HttpRoutes[IO]` to define the API of an HTTP server using an `f2.Stream` instance.

17.2 A Ping Server using http4s

After giving an overview of `http4s`, let's see how you can use it to code an HTTP server. Your goal is to implement an HTTP server that runs on `localhost:8000`, and that replies to a `GET /ping` request with a response with status code "200 - OK" and containing the string "pong": have a look at figure 17.2 for a visual summary of your server's requirements.



Figure 17.2: Functionalities of your HTTP server: when receiving a `GET /ping` request, it should reply with a response with status code "200 - OK" and body "pong".

17.2.1 Initial Setup

Let's create an empty sbt project: use your IDE or a giter8 template to do so. Alternatively, you can create one manually as follows:

1. Create a `build.sbt` file in the directory of your project: it should contain the name and version of the project together with the Scala version you'd like to use:

```

// file build.sbt
name := "ping-app"
version := "0.1"
scalaVersion := "2.13.4"
  
```

2. Create a project directory and create a `build.properties` file that includes your sbt version:

```
// file project/build.properties
sbt.version = 1.4.4
```

3. Create the directories `src/main/scala` and `src/main/resources` to store your source code and static configuration files.

Next, you need to add the `http4s` modules you'll use for your HTTP server to your `build.sbt` file. You are also going to add the logback library: `http4s` uses it as its logging engine.

```
// append to build.sbt

val Http4sVersion = "0.21.14"

libraryDependencies ++= List(
  "org.http4s"      %% "http4s-blaze-server" % Http4sVersion,
  "org.http4s"      %% "http4s-dsl"        % Http4sVersion,
  "ch.qos.logback"  % "logback-classic"   % "1.2.3"
)
```

`logback` is a popular Java library to customize your logs' behavior and appearance via a configuration file, usually called `logback.xml`: let's create one in the `src/main/resources` folder. Have a look at <https://logback.qos.ch> for more information on `logback` and how to configure it.

```
<!-- file src/main/resources/logback.xml -->

<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%msg %n</pattern>
    </encoder>
  </appender>
  <root level="INFO">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

Finally, let's define a package that will include all the code for your HTTP ping route by creating a folder `src/main/scala/org/example/ping`.

17.2.2 Implementing the API

Let's define a class called `PingApi` to define the routes of your API. `PingApi` should extend `Http4sDsl`, a trait that introduces a more intuitive DSL to match HTTP requests and produce HTTP responses. A possible implementation is the following:

Listing 17.1: The Ping API

```
// file src/main/scala/org/example/ping/PingApi.scala

package org.example.ping

import cats.effect.IO
import org.http4s.HttpRoutes
import org.http4s.dsl.Http4sDsl

class PingApi extends Http4sDsl[IO] { ①

    val routes = HttpRoutes.of[IO] { ②
        case GET -> Root / "ping" => Ok("pong") ③
    }
}
```

① `org.http4s.dsl.Http4sDsl` provides an intuitive DSL for matching HTTP requests

② You need to provide a function from a request to a response wrapped in IO to define an instance of `HttpRoutes[IO]`.

③ `case GET -> Root / "ping"` matches a GET request with path `/ping`. `Ok("pong")` returns a response with status code “200–OK” and body “pong”. You do not need to wrap the response as an IO type: the compiler does it for you at compile time automatically. QUICK CHECK 17.1

QUICK CHECK 17.1

Suppose you’d like your `PingApi` to match a request with path `/ping`, but with any HTTP method (i.e., GET, POST, PUT, DELETE, PATCH). How would you change your code?

QUICK CHECK 17.2

Add a new endpoint to your `PingApi` to match a GET request with a path `/ping/<name>`: it should return a string containing “pong” followed by the value passed in the path.

17.2.3 Building a Server

The only missing component for your HTTP server is your executable application that extends `IOApp`. It defines your server, so you should ensure to instantiate it only once by declaring it as an object.

```
// file src/main/scala/org/example/ping/PingApp.scala

package org.example.ping

import cats.effect.{ExitCode, IO, IOApp}
import org.http4s.server.Router

import org.http4s.implicits._ ①
import org.http4s.server.blaze.BlazeServerBuilder
import scala.concurrent.ExecutionContext

object PingApp extends IOApp { ②

    private val httpApp = Router(
        "/" -> new PingApi().routes
    ).orNotFound ③

    override def run(args: List[String]): IO[ExitCode] =

```

```

    stream(args).compile.drain.as(ExitCode.Success)

private def stream(args: List[String]): fs2.Stream[IO, ExitCode] =
  BlazeServerBuilder[IO](ExecutionContext.global) ❸
    .bindHttp(8000, "0.0.0.0")
    .withHttpApp(httpApp)
    .serve
}

```

- ❶ The `org.http4s.implicits._` import adds the `orNotFound` function to your scope: it returns a 404 error code if a request doesn't match any route.
- ❷ `IOApp` is an interface that provides requires to implement a function.
- ❸ `BlazeServerBuilder` defines an HTTP server with an address, port, and routes using a stream to process requests. `ExecutionContext.global` indicates the thread pool to use while streaming: you'll learn more about this when discussing concurrency and the type `Future`.

`IOApp` is an interface that requires you to define a `run` function. Use `BlazeServerBuilder[IO]` together with the function `bindHttp` to provide a port and host. The function `withHttpApp` attaches a group of routes with a prefix to your server. Finally, the function `serve` transforms your Blaze definition into a stream that represents your HTTP server.

QUICK CHECK 17.3

Change your code not to provide a host or port for your server by invoking the function `bindHttp` without parameters. What happens when you start your application using the command `sbt run`?

17.2.4 Let's try it out!

The implementation of your HTTP ping server is now complete: it's time to see it in action! Open your terminal, navigate to your project's root directory, and execute the command `sbt run`. `sbt` will download the dependencies, compile your code, and start your application. After a few seconds, you should see a message similar to the following in your console:

```
$ sbt run
[info] Running org.example.ping.PingApp
Service bound to address /0:0:0:0:0:0:8000
[|_|_|_|_|_|_,-,_|_|_|
 | \ -| -|_| -\ -|-<
 | | \ \ \_| -/_|_|/_/|
 | |
http4s v0.21.14 on blaze v0.14.14 started at http://[0:0:0:0:0:0:0]:8000/
```

Your HTTP server is now running and listening on localhost:8000.

If you send a GET `/ping` request, you will get back a response with status code "200 - Ok" and body "pong":

```
$ curl -i localhost:8000/ping
HTTP/1.1 200 OK
Content-Type: text/plain; charset=UTF-8
Date: Wed, 30 Dec 2020 19:16:18 GMT
Content-Length: 4

pong
```

On the other hand, if you send a POST /ping request, rather than a GET /ping request, the server will reply with a "404 – Not Found" response:

```
$ curl -i -X POST localhost:8000/ping
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=UTF-8
Date: Wed, 30 Dec 2020 19:16:35 GMT
Content-Length: 9

Not found
```

17.3 Summary

In this lesson, my objective was to teach you how to build a server that provides an HTTP API to handle GET requests.

- You have discovered `http4s`, a purely functional library to handle HTTP communication.
- You have reviewed its main components, and you have applied them to implement an HTTP ping server.

Let's see if you got this!

TRY THIS

Implement a server that exposes an API that successfully replies to any request with a response with status code "200 – Ok" and its body containing a message that provides the request method and path. For example, when receiving a request POST /this/is/an/example, it should reply with a response with status code "200 – Ok" and a body "method is POST; path is /this/is/an/example".

17.4 Answers to Quick Checks

QUICK CHECK 17.1

You can change your partial function to match any HTTP method of an incoming request by using the underscore symbol as follows:

```
case _ -> Root / "ping" => Ok("pong")
```

QUICK CHECK 17.2

Implement an additional case clause for the partial function that defines your `HttpRoutes` instance:

```
case GET -> Root / "ping" / name => Ok(s"pong $name")
```

QUICK CHECK 17.3

The function `bindHttp` uses predefined defaults when invoked with no parameters. It binds the server to host 127.0.0.1 and port 8080:

```
[info] running org.example.ping.PingApp  
Service bound to address /127.0.0.1:8080
```



```
http4s v0.21.14 on blaze v0.14.14 started at http://127.0.0.1:8080/
```

18

The Time HTTP Server

In this capstone, you will:

- Define partial functions and pattern matching constructs
- Handle exceptions using a try-catch expression
- Implement an HTTP server using http4s

In this lesson, you'll adapt the code you have written in the previous capstone to implement an HTTP server to provide the current date and time for a given country (e.g., "Italy" or "Japan"). In particular, you'll expose an HTTP API to returns a representation of the current date and time or a human-readable error message.

This lesson uses Scala 2

The code examples in this lesson use Scala 2 because the library http4s doesn't support Scala 3 yet. The lesson will be updated to Scala 3 before publication as soon as an http4s version for Scala 3 is available.

18.1 What time is it?

Let's define an API that reflects the business requirements. Your server should be able to handle GET requests to the URI /datetime/<country> and return a string representing the current date and time in the correct timezone for the given country. For example, GET /datetime/italy should respond with a status code "200 – Ok" and a string with the current date and time in Rome using a human-readable format, such as RFC 1123 (e.g., "Fri, 27 Apr 2020 11:44:35 +0200"). If the given country is not valid or not supported, your application should error with an explicative message. For example, GET /datetime/invalid should return a response with status code "404 - Not Found" and the body "Unknown timezone for country invalid". Figure 18.1 provides a summary of the expected behavior of this API.



Figure 18.1: Overview of the API for your Time HTTP Server. When recognizing the country, the application should respond successfully with a string representing its current date and time. Otherwise, it should error with an explicative human-readable message.

18.1.1 Setting your sbt Project up

In the capstone for unit 2, you have created an empty sbt project together with an `org.example.time` package for your application: have a look at section 13.1.1 if you'd like to refresh your memory on how to do this.

You now need to add your external dependencies: you'll use `http4s` and `logback` to handle HTTP requests and log messages in the terminal. You'll also need `java.time` to compute the current date and time, but you do not need to include it as an external library because it is already accessible as an internal module. Listing 18.3 illustrates how to add the dependencies to your `build.sbt` file.

Listing 18.3: Adding dependencies to build.sbt

```
// append to build.sbt

val Http4sVersion = "0.21.14"

libraryDependencies ++= List(
  "org.http4s"      %% "http4s-blaze-server" % Http4sVersion,
  "org.http4s"      %% "http4s-dsl"        % Http4sVersion,
  "ch.qos.logback"  % "logback-classic"   % "1.2.3"
)
```

You also need to create a directory `src/main/resources` for your `logback.xml` file and any other static configuration. The `logback.xml` file contains the settings and formats for your application's logger: listing 18.4 provides an example of a well-structured `logback.xml` file.

Listing 18.4: An example of a logback.xml file

```
<!-- file src/main/resources/logback.xml -->

<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <withJansi>true</withJansi>
        <encoder>
            <pattern>
                %d{HH:mm:ss.SSS} %highlight(%-5level) - %msg %
            </pattern>
        </encoder>
    </appender>
    <root level="INFO">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

If you have done everything right so far, you should be about to execute the command `sbt compile`, or `sbt update`, from the root directory of your project: sbt will download all the external dependencies you requested.

18.1.2 The TimePrinter class

In capstone 2, you have implemented the `TimePrinter` class to get the current date and time in a given timezone (see section 13.1.2). Originally, you used the class to compute the date and time in a given timezone, but now the business requirements demand that you do this using a country instead. Let's modify the existing code of the `TimePrinter` class so that the function `now` takes a country as the parameter, and let's define a function to match a country to its timezone:

Listing 18.5: The TimePrinter class

```
// file src/main/scala/org/example/time/TimePrinter.scala
package org.example.time

import java.time.format.DateTimeFormatter
import java.time.{ZoneId, ZonedDateTime}

class TimePrinter(formatter: DateTimeFormatter) {

    def now(country: String): String = {
        val timezone = countryToTimezone(country)
        val dateTime = currentDateTime(timezone)
        dateTimeToString(dateTime)
    }

    private def countryToTimezone(country: String): String = ①
        country.toLowerCase match {
            case "italy" => "Europe/Rome"
            case "uk" => "Europe/London"
            case "germany" => "Europe/Berlin"
            case "japan" => "Asia/Tokyo"
            case _ =>
                val msg = s"Unknown timezone for country $country" ②
                throw new IllegalArgumentException(msg)
        }
    }
}
```

```

private def currentDateTime(timezone: String): ZonedDateTime = {
  val zoneId = ZoneId.of(timezone)
  ZonedDateTime.now(zoneId)
}

private def dateTimeToString(dateTime: ZonedDateTime): String =
  formatter.format(dateTime)
}

```

- ① It matches a country with its timezone.
- ② If the country is not recognized, it throws an `IllegalArgumentException`: you'll need to handle it later.

Now that the business logic is ready, you can define the routes of your API.

18.1.3 The API Routes

After implementing your business logic, you need to define your API. Let's create a class, called `TimeApi`, with a service to handle requests in the shape of `GET /datetime/<timezone>`. If successful, you should reply with a "200 - Ok" status code and a body containing the current date and time in format RFC 1123. Otherwise, you should return a response with the status code "404 - Not Found" and a body containing a human-readable error message. Have a look at listing 18.6 for a possible implementation of the class `TimeApi`:

Listing 18.6: The TimeApi class

```

// file src/main/scala/org/example/time/TimeApi.scala

package org.example.time

import java.time.format.DateTimeFormatter

import cats.effect.IO
import org.http4s.HttpRoutes
import org.http4s.dsl.Http4sDsl

class TimeApi extends Http4sDsl[IO] {

  private val printer = ①
    new TimePrinter(DateTimeFormatter.RFC_1123_DATE_TIME)

  val service = HttpRoutes.of[IO] {
    case GET -> Root / "datetime" / country =>
      try {
        Ok(printer.now(country)) ②
      } catch {
        case ex: IllegalArgumentException => ③
          NotFound(ex.getMessage)
      }
  }
}

```

- ① No need to expose the printer value.
- ② The function now throws an exception if the country is invalid or not supported.

- ③ It handles an `IllegalArgumentException` by returning a response with status code “404 – Not Found” and a body containing its error message.

The function call `printer.now(timezone)` throws an exception if it doesn’t recognize the country as valid: it inherits this behavior from the `countryToTimezone` function of the `TimePrinter` class. You need to remember to handle this implementation detail correctly, or your server will error with a nasty “500 – Internal Server Error”.

18.1.4 The HTTP server

The final step is to define a server using an executable object called `TimeApp`. You need to bind it to a port and host and mount the HTTP service you have implemented in the previous section. Listing 18.7 shows how you can do this:

Listing 18.7: The `TimeApp` object

```
// file src/main/scala/org/example/time/TimeApp.scala

package org.example.time

import cats.effect.{ExitCode, IO, IOApp}
import org.http4s.implicits._
import org.http4s.server.Router
import org.http4s.server.blaze.BlazeServerBuilder

import scala.concurrent.ExecutionContext

object TimeApp extends IOApp {

    private val httpApp = Router(
        "/" -> new TimeApi().routes
    ).orNotFound

    override def run(args: List[String]): IO[ExitCode] =
        stream(args).compile.drain.as(ExitCode.Success)

    private def stream(args: List[String]) =
        BlazeServerBuilder[IO](ExecutionContext.global)
            .bindHttp(8000, "0.0.0.0") ①
            .withHttpApp(httpApp) ②
            .serve
}
```

- ① It binds the server to `http://localhost:8000`
 ② It mounts the API routes to the server

Your HTTP server is now fully implemented and ready to run.

18.1.5 Let’s try it out!

Your HTTP server is now complete and ready to run. Navigate to your project’s root folder to compile and run your application using the command `sbt run`. After a few seconds, you should see that the server is up and ready to process HTTP requests:

```
[info] Running org.example.time.TimeApp
20:36:31.979 INFO - Service bound to address /0:0:0:0:0:0:0:8000
20:36:31.982 INFO - 
20:36:31.983 INFO -   [ _ _ _ _ | _ _ | _ _ ] 
20:36:31.983 INFO -   [ _ \ _ | _ | _ \ _ | ( _ <
20:36:31.983 INFO -   [ _ | _ \ _ | \ _ | . _ / | _ | / _ / 
20:36:31.983 INFO -   [ _ | 
20:36:32.047 INFO - http4s v0.21.14 on blaze v0.14.14 started at http://[0:0:0:0:0:0:0:0]:8000/
```

If you perform a GET request to `http://localhost:8000/datetime/italy`, you should get the current date and time in the correct timezone and the expected format:

```
$ curl -i http://localhost:8000/datetime/Italy
HTTP/1.1 200 OK
Content-Type: text/plain; charset=UTF-8
Date: Sun, 29 Apr 2020 20:45:59 GMT
Content-Length: 29

Sun, 29 Apr 2020 22:45:59 +0200
```

On the other hand, if you call <http://localhost:8000/datetime/invalid>, you should receive a “404 – Not Found” response with an error message:

```
$ curl -i http://localhost:8000/datetime/invalid
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=UTF-8
Date: Sun, 29 Apr 2020 20:52:33 GMT
Content-Length: 36

Unknown timezone for country invalid
```

Play around with your server and add support for more countries: can you spot any bugs or implementation issues in your HTTP server? You’ll discover some of them in the next section.

18.2 The ugly bits of our solution

Congratulations on completing your HTTP server! Your implementation respects the requirements, but you could improve a few of its aspects: let’s see some of them and what techniques you’ll learn to overcome them.

EXCEPTIONS ARE TERRIBLE

Exceptions are equivalent to ticking bombs: you throw them and hope that someone will catch them before your program explodes. When coding, you cannot identify which functions throws which exceptions by looking at their signature: you need to look at its (reliable?) documentation or its full implementation most of the time. Forgetting about them is extremely easy, and often you end up writing code wrapped around a preventing try-catch expression. In future lessons, you will learn how to represent error using types rather than exceptions. For example, you’ll learn about the type `Option` as the representation of a possible missing value in the next unit.

COUNTRY-TIMEZONE MAPPING

According to Google, there are 195 countries in the world: your server supports only four of them. You also have countries that people identify with multiple names: for example, “UK”, “United

Kingdom”, “United Kingdom of Great Britain and Northern Ireland” are different names for the same entity. Having a function with 195+ case clauses is not realistic. Your API also has another serious limitation: it assumes that a country has only one timezone, but this is not true for many of them, such as the USA, India, Australia, and Russia. A possible solution would be to change your API to return a list of current dates and times for a given country. Later on in the book, you’ll learn about the `List` and `Map` data structures, and you’ll discover how to generalize this country-timezone relation more efficiently.

DATA REPRESENTATION

Your server responds with plain text, which is the way humans prefer to process data. On the other hand, machines struggle to process data in plain text: they need to tokenize the string correctly to understand it correctly. When exchanging data over HTTP, you can more efficiently parse and interpret structured data formats, such as JSON and XML. In unit 5, you will learn how to serialize/deserialize JSON objects from/to Scala classes so that a machine can quickly and reliably process the exchanged data.

18.3 Summary

In this capstone, you have implemented a fully working HTTP server to determine the current date and time in a given country.

- You have created an sbt project, added some external dependencies, and set a `logback` logging configuration up.
- You have implemented an HTTP API for your server using `http4s`, and you have used the `java.time` module and pattern matching to define your business logic.
- Finally, you have also discovered some of the non-so-ideal aspects of your implementation and seen which techniques will help you overcome them.

Unit 4

Immutable Data and Structures

In Unit 3, you learned how to create an HTTP server that consumes GET requests. In this unit, you'll discover how to implement an HTTP server with POST requests to play the game "Paper, Rock, Scissors, Lizard, Spock!". In particular, we will discuss the following subjects:

- Lesson 19 illustrates what case classes are and how to use them to make your data representations immutable. You'll also learn how pattern matching supports case classes thanks to their `unapply` function.
- Lesson 20 teaches you about higher order functions, one of the most useful types of functions in Scala. You'll see how you can use functions both as parameters and return values of other functions.
- Lesson 21 introduces you to the notion of purity, one of the fundamental concepts of functional programming. You'll also learn about referential transparency and how to differentiate between pure and impure functions.
- Lesson 22 shows you how to model nullable values using the type `Option`. You'll learn how to represent an optional value and pattern match on it.
- Lesson 23 teaches you about the `map`, `flatten`, and `flatMap` functions: they are fundamental operations that you can perform on `Option` to transform and combine optional values.
- Lesson 24 introduces you to for-comprehension as a tool to reduce boilerplate code when combining instances of `Option`. I'll also give you an overview of what other operations you can perform on nullable values.
- Lesson 25 teaches you about the structure `tuple` as a way to quickly group elements together. You'll also learn how to implement an `unapply` function for a generic class.
- Finally, you'll apply everything you have learned so far to implement an HTTP server to play the game "Paper, Rock, Scissors, Lizard, Spock!" in lesson 26.

After learning how to represent data in an immutable manner, you'll continue with Unit 5, in which you'll discover the `List` collection.

19

Case Classes to structure your data

After reading this lesson, you'll be able to:

- Represent immutable structured data using case classes
- Decide when to use case objects rather than regular objects
- Use case classes together with pattern matching

In the previous unit, you have mastered how to create a simple HTTP server. In this lesson, you'll discover an essential tool in your Scala's toolbox, called "case class". When coding, dealing with data is a fundamental and recurring task. A case class provides a convenient and efficient way to represent your data in an immutable way: this allows you to share data between multiple threads safely. Being able to express data efficiently and conveniently is essential to make sure our program works correctly. You'll also learn about case objects and how they can be useful when serialization is involved. Finally, you'll see how pattern matching provides dedicated support for case classes, thanks to the `unapply` function. In the capstone, you'll use case classes and case objects to represent the core elements of the game "Paper, Rock, Scissors, Lizard, Spock!".

Consider this

Think of the languages that you have encountered so far in your coding experience. How do you represent data? Are there methods that you must implement? Do you create both setters and getters? Do you have support from either the language itself or your IDE to reduce any potential boilerplate?

19.1 Case class

Representing data is a crucial part of writing programs, but it is also often mechanical: you need to define your fields, setters, getters, etc. When coding in languages a bit more verbose such as Java, you usually take advantage of tools, such as your IDE, to generate code automatically for you. What if the compiler could do this for you instead of relying on your IDE?

A case class is a class with an arbitrary number of parameters for which the compiler automatically adds ad-hoc code. In Scala's early versions, you couldn't specify a case class with more than 22 parameters, but this limitation no longer exists since Scala 2.11. Case classes are the ideal data containers because they encourage the use of immutability. The implementation of a case class looks very similar to a regular class, but it has an additional `case` keyword. Listing 19.1 shows an example of a case class representing a person with a name and age:

Listing 19.1: The case class Person

```
case class Person(name: String, age: Int)
```

You can also think of a case class as a class that is characterized by its parameters. For example, you could identify an instance of `Person` by its name *and* its age. For this reason, you can also refer to this as "product type". As a result of the keyword `case`, the compiler automatically adds a few convenience methods to it. Let's have an overview of what these are by considering the following instance of the class `Person`:

```
val personA = new Person("Tom", 25)
```

GETTERS

For each parameter, the compiler adds a getter function so that you can easily access it. For example, you can access the name and age of `personA` as follows:

```
personA.name // returns "Tom"
personA.age // returns 25
```

COPY FUNCTION

You do not have setter functions for its parameters because a case class represents data in an immutable way. When changing one of its values, you should use the `copy` function to create a new data representation. Suppose you want to change the age of you instance `personA` to be 35:

```
val personB: Person = personA.copy(age = 35)
personA.age // returns 25
personB.age // returns 35
```

You can also change more parameters at the same time. For example, if you want to change both its name and age you can just provide more parameters to the `copy` function:

```
val mark = personA.copy(name = "Mark", age = personA.age + 1)
mark.name // returns "Mark"
mark.age // returns 26
```

The `copy` function is effectively equivalent to initializing a new class. Using the `copy` function is particularly convenient when a case class has lots of fields, but you only need to change one. Consider the following snippet of code:

```
case class Test(a: Int, b: Int, c: Int, d: Int)
val test = Test(1,2,3,4)

val testA = Test(a = 0, b = test.b, c = test.c, d = test.d)
val testB = test.copy(a = 0)
```

The instances `testA` and `testB` are equivalent: you have initialized `testA` using the `apply` function, while you have initialized `testB` using the `copy` function. Hopefully, you'll agree that the `copy` function approach is more readable than using the `apply` one in this case.

TO STRING, HASHCODE, EQUALS FUNCTIONS

Every class has the functions `toString`, `hashCode`, and `equals`: their implementation comes directly from the Java world. In Java, `java.lang.Object` is the superclass of all the classes, and it provides an implementation for several functions inherited by all the other classes. A case class redefines the implementation for some of these functions inherited from `java.lang.Object`. Let's see how:

- `toString` – By default, a `toString` function returns a string representing the class's name followed by the hexadecimal representation of the instance's memory address (e.g., "Person@1e04fa0a"). A case class redefines this method to return a string that is descriptive of the data it contains:

```
personA.toString() // returns "Person(Tom,25)"
```

- `hashCode` – The `hashCode` function returns an integer that represents an instance of a class. The JVM uses this number in data structures and hash tables when storing objects in a more performant way. While a hash code of an instance usually considers both its internal structure and memory allocation, a case class overrides its hash code so that it considers only its internal structure. The compiler makes sure that two case classes with the same structure have the same hash code.

```
class C(x: Int)
new C(5).hashCode == new C(5).hashCode // returns false

case class A(x: Int)
case class B(n: Int)
new A(5).hashCode == new B(5).hashCode // returns true
```

- `equals` – According to the implementation of `equals` defined in `java.lang.Object`, equality holds if two instances are the *same*. In other words, they are equal if they point to the same memory allocation. When working with case classes, the compiler provides a different implementation for `equals` in which case classes that belong to the same type and structure are considered equal.

```

class C(x: Int)
new C(5).equals(new C(5)) // returns false

case class A(x: Int)
case class B(n: Int)
new A(5).equals(new B(5)) // returns false
new A(5).equals(new A(5)) // returns true

```

Scala 3: Strict Equality

Scala 2 uses universal equality: you can always compare two instances for equality, even if they have different types. For example, the following expression comparing a Boolean and a String is considered valid (with a compiler warning):

```

scala> true == "true"
^
warning: comparing values of types Boolean and String using
`==' will always yield false
val res0: Boolean = false
// it compiles in Scala 2

Scala 3 adopts strict equality: you can only compare instances that have the same type. The previous expression no longer
compiles:
scala> "true" == true
1 |"true" == true
|^^^^^^^^^^^^^^^
|Values of types String and Boolean cannot be compared
with == or !=
// it doesn't compile in Scala 3

```

COMPANION OBJECT: APPLY AND UNAPPLY FUNCTIONS

When declaring a case class, the compiler generates its companion object with implementations for the `apply` and `unapply` functions. You can use such methods to construct and deconstruct an instance of a case class, respectively. Let's see how they work:

- `apply` – thanks to the generated `apply` method, you can create an instance of your case class by providing parameters for it. You have already encountered the `apply` function when discussing singleton objects: let's quickly recap how it works. For example, to create an instance of `Person`, you can use the `apply` method rather than directly invoking its constructor. All the following expressions are equivalent:

```

new Person("Tom", 25)
Person.apply("Tom", 25)
Person("Tom", 25)
Person(age = 25, name = "Tom")

```

- `unapply` – you can use the `unapply` method to decompose a class. In a case class, the compiler implements the `unapply` to return the class fields. For example, you can decompose a `Person` to obtain an optional grouping containing a name and an age:

```
Person.unapply(Person("Tom", 25))
// returns Some((Tom,25)), which has type Option[(String, Int)]
```

You may not be able to fully understand its return type and implementation yet: you'll learn about the `unapply` method in detail when discussing optional values and tuples. In the next section, you'll see how having an implementation for the `unapply` function allows you to pattern match over the fields of a case class.

Through the use of case classes, the compiler saves us from writing lots of potentially buggy code! Listing 19.2 shows the amount of boilerplate that you would have to define to implement a class that is equivalent to a case class:

Listing 19.2: Class versus Case Class

```
class Person(n: String, a: Int) {

    val name: String = n
    val age: Int = a

    def copy(name: String, age: Int) =
        new Person(name, age)

    override def toString(): String = s"Person($n,$a)"

    override def hashCode(): Int = ??? ①

    override def equals(obj: Any): Boolean = ??? ①
}

object Person {

    def apply(name: String, age: Int): Person =
        new Person(name, age)

    def unapply(p: Person): Option[(String, Int)] =
        Some((p.name, p.age))
}
```

① Implementation omitted

Figure 19.1 shows a summary of how to declare a case class and the functionalities the compiler generates for it.

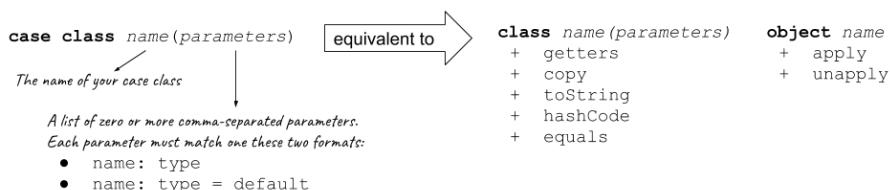


Figure 19.1: A syntax diagram for a case class. A case class is equivalent to a regular class with its companion object that the compiler has enriched with a set of useful functions.

QUICK CHECK 19.1

Define case classes to express the following relations:

- a brewery has a name
- a beer has a name and a brewery

19.2 Pattern Matching and Case Classes

In the previous section, you have discovered that the compiler provides an implementation for the `unapply` function for a case class. This function decomposes a case class into its parameters, and it enables pattern matching to analyze them. This process is entirely transparent: you do not need to invoke the `unapply` function explicitly. The compiler will look for an `unapply` function in the companion object for the class and the number of parameters you use in your pattern matching construct.

Consider again the case class `Person` you have seen in listing 19.1:

```
case class Person(name: String, age: Int)
```

Suppose that you want to write a function, called `welcome`, that returns a different message depending on the name and age of a `Person`. Listing 19.3 shows you how to achieve this using pattern matching on parameters of a class:

Listing 19.3: Pattern Matching of a case class `Person`

```
def welcome(person: Person): String = person match {
    case Person("Tom", _) => "Hello Mr Tom!" ①
    case Person(name, age) if age > 18 => s"Good to see you $name" ②
    case p @ Person(_, 18) => s"${p.name} you look older!" ③
    case Person(_, _) => "Hi bro!" ④
}
```

- ① It matches a person with the name "Tom"
- ② It matches a person with age bigger than 18
- ③ It matches a person with age 18, and it binds it to a value `p`
- ④ It matches a person with any name and any age

You have seen a new way of patterning using value binding in listing 19.3:

```
case p @ Person(_, 18) => s"${p.name}, you look older!"
```

When pattern matching a class, you can also bind the entire class instance to a value by providing a name and the symbol `@`.

QUICK CHECK 19.2

What happens to the implementation of the function `welcome` if you declare `Person` as a regular class rather than a case class – that is, you remove the `case` keyword from its declaration?

19.3 Case object

Now that you have understood what case classes are, you may wonder if an equivalent scenario exists for singleton objects: you refer to them as "case objects". Have a look at listing 19.4 for an example of a case object for the currency USD:

Listing 19.4: The USD currency

```
case object USD
```

A case object is a regular singleton object for which the compiler automatically overrides some useful methods: it redefines the implementation of `toString` to produce a human-readable string representation. For a regular object, `toString` returns its name followed by the hexadecimal encoding of its memory address: this looks similar to "USD@7b36aa0c". When dealing with a case object, the compiler changes the definition for `toString` to return only the object name:

```
USD.toString // returns "USD"
```

Have a look at figure 19.2 for a summary of the syntax for case objects.

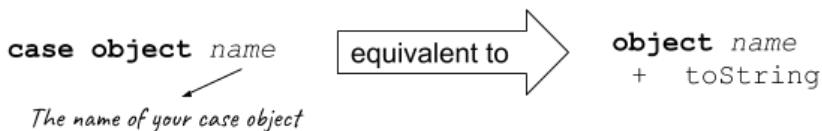


Figure 19.2: A syntax diagram for a case object in Scala. A case object is equivalent to an object with a redefined `toString` function.

QUICK CHECK 19.3

Consider the code you have seen in listing 19.4. Modify the implementation of `USD` to define three more currencies using case objects: GBP, CAD, and EUR. Use a sealed trait to group them all as currency.

19.4 Summary

In this lesson, my objective was to teach you about case classes, how they differ from regular classes, and why they are beneficial when representing data.

- You have learned how to use pattern matching with a case class and use the symbol `@` to bind an entire instance to a value.
- We have also discussed case objects and why they are ideal for representing singleton objects using strings.

Let's see if you got this!

TRY THIS

Use case classes and case objects to represent the following data:

- An author has a forename and a surname.
- A genre has only three possible values: Drama, Horror, Romantic.
- A book has a title, an author, and a genre.

19.5 Answers to Quick Checks

QUICK CHECK 19.1

The following code describes the relations between brewery and beer:

```
case class Brewery(name: String)
case class Beer(name: String, brewery: Brewery)
```

QUICK CHECK 19.2

The function `welcome` no longer compiles. The compiler complains that it cannot find a value `Person`: a companion object with the name `Person` containing an implementation for the `unapply` function no longer exists because you have declared `Person` as a regular class rather than a case class. Also, its fields `name` and `age` are no longer accessible.

QUICK CHECK 19.3

A representation for currencies using case objects and a sealed trait is the following:

```
sealed trait Currency
case object USD extends Currency
case object GBP extends Currency
case object CAD extends Currency
case object EUR extends Currency
```

20

Higher Order Functions

After reading this lesson, you will be able to:

- Define functions that have functions as parameters
- Implement functions that return functions
- Build powerful abstractions that remove code duplication

Functions are first-class citizens in Scala: you can use them as parameters or return them as the result of some computation. “Higher order functions” are functions that accept other functions as parameters, or that return functions, or both. Their use allows you to create powerful abstractions that reduce duplication and increase the reusability of your code. In the capstone, you’ll use higher order functions to extract information on the winner of the game “Paper, Rock, Scissors, Lizard, Spock!”.

Consider this

Suppose you want to compute some calculations and perform some input/output operation on them. For example, you may need to express two case scenarios: you should display the result of your computation into the console in the first, while you should append it to an existing file in the second. How would you design your functions to avoid code duplication and ensure their implementation is consistent everywhere in your program? How efficiently can you add support for other IO operations, such as querying a database?

20.1 Functions as Parameters

Suppose you want to perform some simple statistics on a string to calculate its length, how many letters, and how many digits it has. Have a look at listing 20.1 for a possible solution in which you create a function for each of the statistics you need to implement:

Listing 20.1: Stats on a string

```
def size(s: String): Int =
  s.length

def countLetters(s: String): Int =
  s.count(_.isLetter) ①

def countDigits(s: String): Int =
  s.count(_.isDigit) ①
```

① `count` is a function defined in the class `String` to calculate how many chars of a string respect a specific property.

The solution shown in listing 20.1 works, but it is not ideal. Adding new types of statistics to your program drastically increases the number of functions you need to write and maintain. The functions `countLetters` and `countDigits` are very similar: the only difference is the predicate used to determine if you should count a char. How can you design functions so that you can avoid code duplication? How can you also ensure that Can you define them so that you can support new operations easily?

Let's look again at the statistics you need to perform: you need to compute its length, count its letters, and count its digits. In other words, you need to determine if you should consider a character. You should include every one of them when calculating the length of a string. At the same time, you should consider only those with a particular property (i.e., being a letter, being a digit, etc.) for the other two operations. Listing 20.2 shows how you can implement this logic using a higher order function that takes a function as a parameter:

Listing 20.2: Another stats on a string

```
def stats(s: String, predicate: Char => Boolean): Int = ①
  s.count(predicate)
```

① `predicate` is a function that takes a char as its parameter and returns a boolean

After defining the function `stats`, you can compute the requested statistics needed as follows by removing any code duplication:

```
def size(s: String): Int = stats(s, _ => true)

def countLetters(s: String): Int = stats(s, _.isLetter)

def countDigits(s: String): Int = stats(s, _.isDigit)
```

You can also support custom statistics by calling the function `stats` directly. For example, you can count the number of uppercase letters or how many chars "x" it contains as follows:

```
val text = "This is my Text Example"
stats(text, _.isUpper) // count of upper letters
stats(text, _ == 'x') // count of chars equal to "x"
```

QUICK CHECK 20.1

Write a function called `foo` that takes a function `f` of type `Int => Double` as its parameter and returns a `Double`: apply 42 to the function parameter and then add 2 to its result.

You can specify defaults for function parameters. For example, you can change the implementation of your `stats` function so that it counts all the chars by default – in other words, to be equivalent to the length of a string:

Listing 20.3: Stats on a string with default

```
def stats(s: String,
         predicate: Char => Boolean = { _ => true }): Int = ①
  s.count(predicate)
```

① `predicate` is the name of the function parameter. `Char => Boolean` is the type of the function parameter. `{ _ => true }` is an anonymous function used as default value

Once you have provided a default for the function parameter, you can call the function without providing the parameter `predicate`:

```
stats(s) // count of all the chars
```

Have a look at figure 20.1 for a summary of the Scala syntax for function parameters.

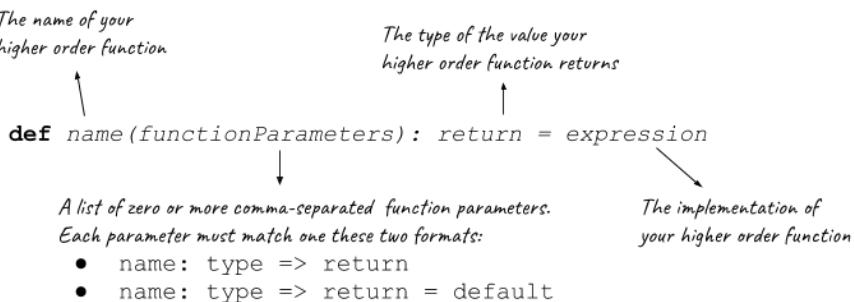


Figure 20.1: A syntax diagram for function parameters in Scala and their use of defaults.

QUICK CHECK 20.2

In the previous quick check, you have implemented a function called `foo`. Modify your implementation to add the function `toDouble` from the class `Int` as the default of its parameter `f`.

20.2 Functions as return values

In the previous section, you have implemented a function called `stats` to perform statistics on a string (see listing 20.3). In particular, you have represented each of the predefined options with three functions called `size`, `countLetters`, `countDigits`. This solution works for a small set of alternatives, but it will quickly become quite verbose once your program needs to provide more and more preferences: you will need to define a function with shape `String => Int` for each possible alternative. This approach can cause the number of functions in your program to explode: if you need to express 20 different alternatives, you will have 20 functions with a very similar structure.

Another solution is to represent all the available selections with a set of well-defined values (or “union types”) and write a function to select the predicate filter to use for your `stats` function:

Listing 20.4: PredicateSelector function

```
sealed trait Mode ①
case object Length extends Mode
case object Letters extends Mode
case object Digits extends Mode

def predicateSelector(mode: Mode): Char => Boolean = ②
  mode match {
    case Length => _ => true
    case Letters => _.isLetter
    case Digits => _.isDigit
  }
```

① A finite set of values to define all the options

② It selects a predicate based on a given mode

Now that you have defined the `predicateSelector` function you can use it to call the `stats` function as follows:

```
val text = "This is my Text Example"
stats(text, predicateSelector(Length)) // count of all chars
stats(text, predicateSelector(Letters)) // count of upper letters
```

Thanks to this solution, you can support additional predefined statistics by adding a case object for the trait `Mode` and a case clause in the `predicateSelector` function, rather than defining a new function from scratch.

Figure 20.2 provides a summary of how to implement higher order functions that return functions in Scala.

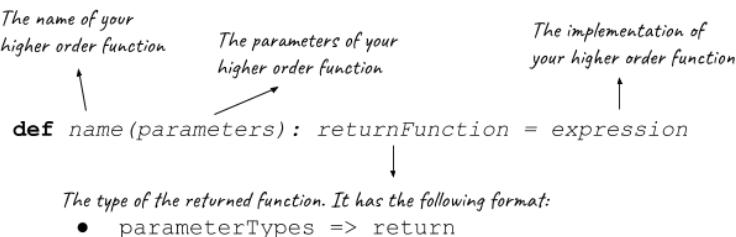


Figure 20.2: A syntax diagram of higher order functions that return functions as result type.

QUICK CHECK 20.3

Change the code shown in listing 20.4 to support a new kind of statistics to count the whitespaces in a string: use the function `isWhitespace` defined in the class `Char`.

20.3 Summary

In this lesson, my objective was to teach you about higher order functions and how to use them to build powerful abstractions.

- You have seen how to create functions that accept other functions as parameters.
- You have also discovered how to implement functions that return functions.

Let's see if you got this!

TRY THIS

Write a function, called `operationWithFallback`, that returns an `Int` and has three parameters:

- `n` is an integer
- `operation` is a function from `Int` to `Int`
- `fallback` is an integer.

The function `operationWithFallback` should implement as follows. Compute the value of `operation` applied to `n`: return it if more than zero, otherwise compute the fallback. Make sure to evaluate `fallback` only if needed.

20.4 Answers to Quick Checks

QUICK CHECK 20.1

An implementation for the function `foo` is the following:

```
def foo(f: Int => Double): Double = f(42) + 2
```

QUICK CHECK 20.2

The implementation is modified as follows:

```
def foo(f: Int => Double = _.toDouble): Double = f(42) + 2
```

QUICK CHECK 20.3

The implementation shown in listing 20.4 should be modified as follows:

```
sealed trait Mode
  case object Length extends Mode
  case object Letters extends Mode
  case object Digits extends Mode
  case object Whitespaces extends Mode

  def predicateSelector(mode: Mode): Char => Boolean =
    mode match {
      case Length => _ => true
      case Letters => _.isLetter
      case Digits => _.isDigit
      case Whitespaces => _.isWhitespace
    }
```

21

What is Purity?

After reading this lesson, you will be able to:

- Differentiate between pure and impure functions
- Provide code examples in which impure functions cause unpredicted code behavior.

In this lesson, you'll learn about purity, a fundamental principle of functional programming. In particular, you'll see that a pure function is total and it has no side effects: you'll discover what these terms mean in detail in this lesson. Distinguishing between pure and impure functions can help you identify and prevent bugs in your code. For example, consider the following scenario:

Suppose you are developing the software for a smart thermostat. Your business requirements dictate your thermostat never to reach temperatures below the freezing point of 0 °C (equivalent to 32 °F) because it could damage its mechanical parts. If this happens, your program should trigger an emergency recovery plan which changes the target temperature to a default value the user can configure. You could translate this with the following function:

```
def monitorTemperature(current: Double, recovery: Double): Double =
  if (current >= 0) current else recovery
```

This function `monitorTemperature` behaves in different ways depending on the “purity” of its parameters. Consider the following function invocations:

```
scala> monitorTemperature(current = 5, recovery = 10)
res0: Double = 5.0

scala> monitorTemperature(
  current = 5,
  recovery = {println("EMERGENCY! triggering recovery"); 10})
EMERGENCY! triggering recovery
res1: Double = 5.0
```

Both function calls are valid, but the second one behaves unpredictably: even when the current temperature is above the freezing threshold, an unexpected (and confusing!) message appears in the console.

Later on in the book, you'll learn about lazy evaluation and how to handle these use cases in which you'd like to evaluate a given function parameter only if needed. In the meanwhile, let's discuss purity and how it differs from impurity. In the capstone, you will use pure functions to determine the winner of the game "Paper, Rock, Scissors, Lizard, Spock!".

Consider this

Can you think of another example in which your function may suffer from some unexpected behavior because of the possibly impure values you have passed as parameters?

21.1 A definition of Purity

A pure function is total, and it has no side effects. Let's see what each of these terms mean.

TOTALITY

A function is total if it is well-defined for every input: it must terminate for every parameter value and return an instance that matches its return type.

Let's consider the following functions:

```
def plus2(n: Int): Int = n + 2 // total
def div(n: Int): Int = 42 / n // non-total
def rec(n: Int): Int = if (n > 0) n else rec(n - 1) // non-total
```

The function `plus2` is total because for every possible integer passed as its parameter, it terminates, and it returns an integer value as the result of its evaluation. The `div` function is not total because even if it always ends, it doesn't always return a value of type `Int`: it throws an `ArithmaticException` if its parameter `n` equal to zero. A thrown exception is an unexpected value not represented in its return type. Finally, the `rec` function is not total because it never terminates for any integer less or equal to zero.

QUICK CHECK 21.1

Which ones of the following functions are total? Why?

1. def opsA(n: Int): Int = if(n <= 0) n else n + 1
2. def opsB(n: Int): Int = if(n <= 0) n else opsB(n + 1)
3. def selectException(predicate: Boolean): Exception =


```
if (predicate) new IllegalStateException("msg here")
      else new ArithmaticException("another msg here")
```
4. def anotherToString(obj: AnyRef): String = {

```

    Thread.sleep(1000) // measured in millis
    obj.toString
}
5. def validateDistance(dist: Double): Double =
  if (dist < 0) {
    throw new IllegalStateException("Distance cannot be negative")
  } else dist

```

SIDE EFFECTS

A side effect is an operation that has an observable interchange with elements outside its local scope: it affects (i.e., write side effect) or is affected by (i.e., read side effect) the state of your application by interacting with the outside world. A few examples are the following:

```

def negate(predicate: Boolean): Boolean = !predicate// no side effect

class Counter {
  private var counter = 0

  def incr(): Unit = counter += 1           // (write) side effect
  def get(): Int = counter                 // (read) side effect
}

def hello(name: String): String = {
  val msg = s"Hello $name"
  println(msg)                           // (write) side effect
  msg
}

```

The function `negate` has no side effects: its only instruction acts on its parameter to produce a return value. The function `Counter.incr` contains a (write) side effect: every time you invoke the function, it changes the assignment for the variable `counter`, which is a code element that lives outside of its local scope. `Counter.get` also has a (read) side effect: given the same input, it returns a different integer depending on the variable `counter`'s current assignment. The function `hello` has a (write) side effect because its `println` instruction produces a message into the console, a component shared across your application that lives independently from its local scope.

QUICK CHECK 21.2

Which ones of the following functions have side effects? Why?

1. def div(a: Int, b: Int): Int = {
 if (b == 0) throw new Exception("Cannot divide by zero")
 else a / b
 }
2. def getUserAge(id: Int): Int = {
 val user = getUser(id) // gets data for a database
 }

```

    user.id
}

3. def powerOf2(d: Double): Double = Math.pow(2, d)
4. def anotherPowerOf2(d: Double): Double = {
    println(s"Computing 2^$d...")
    Math.pow(2, d)
}
5. def getCurrentTime(): Long = System.currentTimeMillis()

```

QUICK CHECK 21.3

A pure function is total and has no side effects. Consider the code snippets provided in Quick Check 21.1 and Quick Check 21.2: which ones of them are pure?

21.2 Differentiating between pure and impure functions

In the previous section, you discovered that a function is pure if total, and without side effects. You can describe this concept in a less formal way as follows: a function is pure if nothing else but its parameters determine its behavior, which its return type entirely describes (see figure 21.1).

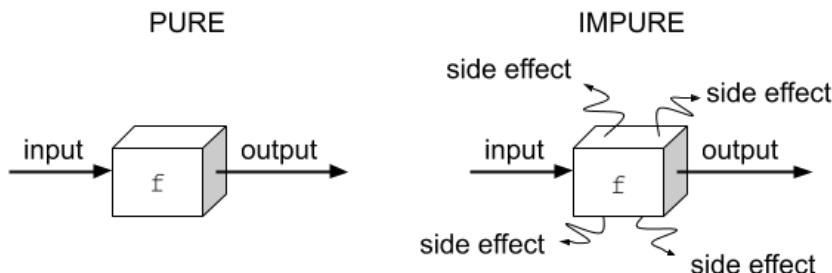


Figure 21.1: A visual representation of the differences between pure and impure functions. Given an input, a pure function returns an output. On the other hand, an impure one also produces additional effects not represented in its return value.

A pure function guarantees that it always returns the same output given the same input parameters. In other words, you can replace its invocation with its return value and obtain the same outcome: This concept is called “referential transparency”, and it has several practical implications.

Suppose you have the following two functions, called `pureF` and `impureF`, that take a string as their parameter and return another string as the result of some computation:

```
def pureF(name: String): String = s"Hi $name!"

def impureF(name: String): String = {
    println("...doing something here...")
    s"Hi $name!"
}
```

The function `pureF` is pure, while the function `impureF` is impure.

You can substitute the function call `pureF("Bob")` with the string "Hi Bob!". On the other side, swapping the function call `impureF("Bob")` with the string "Hi Bob!" would not produce the same result because the print instruction in the console would be missing.

Functions with no parameters: parentheses or no parentheses?

When declaring a function with no parameters, you should omit the parentheses if the function is pure (i.e., `def f = ???`). Vice versa, you should specify them if the function is impure (i.e., `def f() = ???`).

This rule is a style suggestion rather than a law imposed by the compiler.

QUICK CHECK 21.4

Which ones of the following statements are true?

1. Pure functions do more than just computing a value
2. You can replace calls to impure functions with their return value without losing functionalities.
3. Pure functions are total
4. A function that throws exceptions is pure
5. A function with side effects is impure

21.3 Summary

In this lesson, my objective was to teach you about the functional concept of purity.

- You have learned that pure functions are total and have no side effects.
- You have also discovered referential transparency and how you can use it to differentiate between pure and impure functions.

Let's see if you got this!

TRY THIS

Which ones of the following functions are pure? Which ones are impure?

1. `def welcome(n: String): String = s"Welcome $n!"`
2. `def printWelcome(n: String): Unit =`
 `println(s"Welcome $n!")`
3. `def slowMultiplication(a: Int, b: Int): Int = {`

```

    Thread.sleep(1000) // 1 second

    a * b
}

4. def saveUser(user: User): User = {
    insertUser(user) // inserts in a database
    user
}

5. def getUser(id: Int): User = {
    selectUser(id) // searches in a database
}

```

21.4 Answers to Quick Checks

QUICK CHECK 21.1

The answers are as follows:

1. The function `opsA` is total because it always terminates and returns an integer for every integer passed as its parameter.
2. The function `opsB` is not total: it calls itself recursively and never terminates for positive integers.
3. The function `selectException` is total because it returns an instance of exception: it computes a value that matches its return type for every input. The keyword `throw` is missing, so the function does not throw the exception, but it returns it as a class instance.
4. The function `anotherToString` is total: for every input, it eventually terminates after sleeping for 1 second (or 1000 milliseconds) and returning a string. What if the function was to block for a much more extended period (e.g., ten years), would you still consider it total?
5. The function `validateDistance` is not total because it throws an exception for any negative double number.

QUICK CHECK 21.2

The answers are the following:

1. The function `div` has no side effects: throwing exceptions is not a side effect because it does not change the state of components external to the function.
2. The function `getUserAge` returns different results depending on which objects are in the database, which is a (read) side effect.
3. The function `powerOf2` has no side effects as its return value depends entirely on its input.
4. The function `anotherPowerOf2` has a (write) side effects: every time you call it, it produces a new message to the console, changing its state.

5. The function `getCurrentTime` returns a value that depends on your machine's internal clock, which is a (read) side effect.

QUICK CHECK 21.3

In Quick Check 21.1, the functions `opsA`, `selectException`, `anotherToString` are pure. In Quick Check 21.2, there is only one pure function, which is `powerOf2`.

QUICK CHECK 21.4

The answers are as follows:

1. False
2. False
3. True
4. False
5. True

22

Option

After reading this lesson, you will be able to:

- Represent a nullable value using `Option`
- Use pattern matching on instances of the type `Option`

After mastering higher order functions, you'll learn about the type `Option`. Using `null` to represent nullable or missing values in Scala is an anti-pattern: use the type `Option` instead. The type `Option` ensures that you deal with both the presence or the absence of an element. Thanks to the `Option` type, you can make your system safer by avoiding nasty `NullPointerExceptions` at runtime. Your code will also be cleaner as you will no longer preventively check for null values: you will be able to clearly mark nullable values and act accordingly only when effectively needed. The concept of an optional type is not exclusive to Scala: if you are familiar with another language's `Option` type, such as Java, you will recognize a few similarities between them. You'll analyze the structure of the `Option` type. You'll also discover how to create optional values and analyze them using pattern matching. In the capstone, you will use `Option` to represent that a winner for the game "Paper, Rock, Scissors, Lizard, Spock!" may be missing in case of a tie.

Consider this

Suppose you need to design a structure to represent nullable values: how would you indicate that an element may or may not be there?

22.1 Why Option?

Suppose you have defined the following function to calculate the square root of an integer:

```
def sqrt(n: Int): Double =
  if (n >= 0) Math.sqrt(n) else null
```

This function has a fundamental problem. Its signature (i.e., *what* a function does) does not provide any information about its return value being nullable: you will need to look at its implementation (i.e., *how* a function computes a value) and remember to deal with a potentially null value. This approach is particularly prone to errors as you can easily forget to handle the null case, causing a `NullPointerException` at runtime, and it forces you to write a lot of defensive code to protect your code against null:

```
val x: Int = ???  
val result = sqrt(x)  
if (result == null) {  
    //protect from null here  
} else {  
    // do things here  
}
```

The type `Option` is equivalent to a wrapper around your value to provide the essential information that it may be missing. Thanks to the use of `Option`, you no longer need to look at the specific implementation of a function to discover if its return value is nullable because this information will be in its signature. The compiler will also make sure that you handle both the cases when it is present and when it is absent, making your application safer at runtime.

22.2 Creating an Option

After discussing how the use of the type `Option` can improve your code's quality, let's see how you can create instances for it. A nullable value either exists or is missing: the (simplified) definition of the Scala's `Option` shown in listing 22.1 reflects this structure.

Listing 22.1: The Option Type

```
package scala  
  
sealed abstract class Option[A]  
  
case class Some[A](a: A) extends Option[A]  
case object None extends Option[Nothing]
```

Let's analyze its definition line by line and see what each of them mean:

- package `scala`

The `Option` type lives in the `scala` package, so it is already available into your scope without the need for an explicit import.

- sealed abstract class `Option[A]`

`Option` is an abstract class, so you cannot initialize it directly. It is sealed: it has a well-defined set of possible implementations (i.e., `Some` of a given value and `None`). For the first time, you have also have encountered the Scala notation for "generics", which is `Option[A]`. An optional type works independently from the actual instance it contains: you would expect an optional value to behave in the same way as an optional integer, an optional string, or any other optional value. With the notation `Option[A]`, you tell the compiler that you will

associate `Option` with a type that you will provide during initialization: in other words, `Option` has a “type parameter”. Scala has a convention of using upper case letters of the alphabet for type parameters, the reason why `Option` uses `A`, but you could have provided any other name for it.

- `case class Some[A](a: A) extends Option[A]`

`Some` is case class and a valid implementation of `Option` that represents the presence of a value. It has a type parameter `A` that defines the type of the value it contains. `Some[A]` is an implementation of `Option[A]`, which implies that `Some[Int]` is a valid implementation for `Option[Int]`, but not for `Option[String]`:

```
scala> val optInt: Option[Int] = Some(1)
optInt: Option[Int] = Some(1)

scala> val optString: Option[String] = Some(1)
<console>:11: error: type mismatch;
  found   : Int(1)
  required: String
          val optString: Option[String] = Some(1)
```

Scala’s type inference is about to infer type parameters. For this reason, you can write `Some(1)` instead of `Some[Int](1)`.

- `case object None extends Option[Nothing]`

`None` is the other possible implementation for `Option`, and it represents the absence of a value. It is a case object, which means it is a serializable singleton object. You can apply the concept of a missing value to any instance independently from its type: for this reason, `None` doesn’t have a type parameter, but it extends `Option[Nothing]`. `Nothing` has a special meaning in Scala: `Nothing` is the subclass of every other class, and it is at the bottom of the class hierarchy (see figure 22.1):

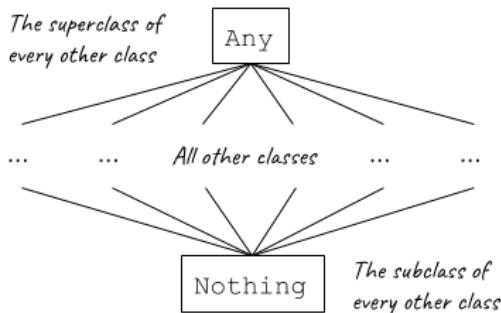


Figure 22.1: In Scala, the types `Any` and `Nothing` have a special meaning. `Any` is the superclass of every other class – in other words, it is the root of the class hierarchy. `Nothing` is at the bottom of the class hierarchy: it is the subclass of every other class.

Because `None` extends `Option[Nothing]`, you can use `None` as a valid implementation for `Option` independently of its type parameter. Thanks to its special meaning, `Nothing` will always be compatible with the type parameter you provided:

```
scala> val optInt: Option[Int] = None
optInt: Option[Int] = None

scala> val optString: Option[String] = None
optString: Option[String] = None
```

The terms `None`, `Nothing`, and `null` can confuse here, so let's recap what each of them means. `None` is an instance of the class `Option`, and it represents a missing nullable value. `Nothing` is a type that you can associate with every other Scala types. The term `null` is a keyword of the language to indicate a missing reference to an object.

Think in Scala: Class versus Type

This lesson considers the terms class and type as synonymous for simplicity; however, this is not always the case.

A class represents a code element with a particular behavior and that you can instantiate through its constructor. A type uniquely identifies a much broader category of items you can use in your programs. Let's use the Scala REPL to see a few examples in action.

String has same class and type:

```
scala> import scala.reflect.runtime.universe._

import scala.reflect.runtime.universe._
```

```
scala> classOf[String]
res0: Class[String] = class java.lang.String
```

```
scala> typeOf[String]
res1: reflect.runtime.universe.Type = String
```

This is not the case when comparing `Option[Int]` with `Option[String]`: they both belong to the class `Option` but they have different types:

```
scala> classOf[Option[Int]]
res2: Class[Option[Int]] = class scala.Option
```

```
scala> classOf[Option[Int]] == classOf[Option[String]]
res3: Boolean = true
```

```
scala> typeOf[Option[Int]]
res4: reflect.runtime.universe.Type = scala.Option[Int]
```

```
scala> typeOf[Option[String]]
res5: reflect.runtime.universe.Type = scala.Option[String]
```

```
scala> typeOf[Option[Int]] == typeOf[Option[String]]
res6: Boolean = false
```

You can now re-implement your `sqrt` function to use the type `Option` as shown in listing 22.2:

Listing 22.2: The `sqrt` function with Option

```
def sqrt(n: Int): Option[Double] =
  if (n >= 0) Some(Math.sqrt(n)) else None
```

Figure 22.2 provides a visual summary of Scala's `Option` type.

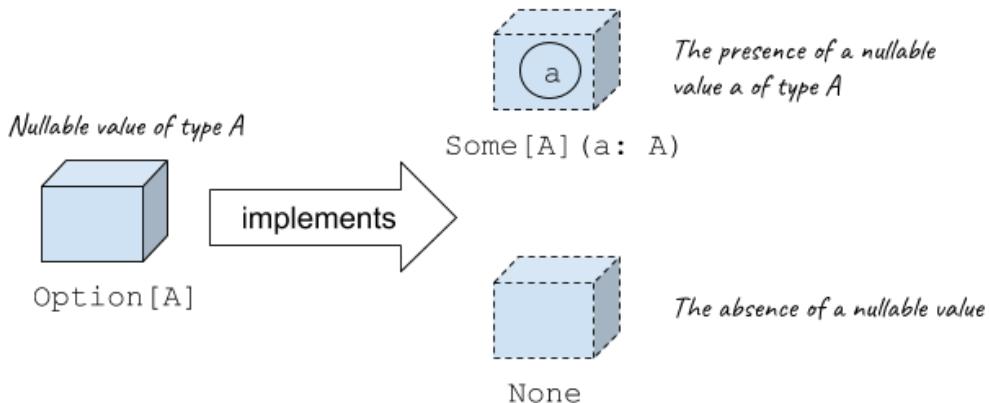


Figure 22.2: Visual summary of the structure of the Scala's `Option` type. An `Option[A]` is either a `Some[A]` of a value `a` or `None`, which represents its absence.

QUICK CHECK 22.1

Write a function called `filter` that takes two parameters of type `String`, called `text` and `word`, and it returns an `Option[String]`. It should return either the original string if `text` contains `word` or no value.

Convert Partial Functions to Total Functions returning Option

In lesson 16, you have learned about partial functions as a tool to abstract commonalities between functions. An example of a partial function is the following:

```
val log: PartialFunction[Int, Double] =
{ case x if x > 0 => Math.log(x) }
```

When you call a partial function on an input that is not defined, it will throw a `MatchError` exception, which is not ideal as exceptions are unpredictable. Instead of using a partial function, you can define a total function that returns an optional value: this approach will protect your code from an unexpected `MatchError` exception at runtime. For example, you could re-implement the function `log` as a total function as follows:

```
def log(x: Int): Option[Double] = x match {
  case x if x > 0 => Some(Math.log(x))
```

```
case _ => None
}
```

Unless you are using an API or library that is explicitly requesting you to use partial functions, consider re-defining your partial functions as total functions to make your code safer at runtime by avoiding possible unpredicted exceptions.

22.3 Pattern Matching on Option

After looking at the structure of an `Option`, let's see how you can handle it.

In lesson 13, you have learned that when pattern matching on a sealed item, the compiler will warn you if you haven't considered all its possible implementations as this could cause a `MatchError` exception. In lesson 19, you have also seen how to pattern matching on case classes and case objects. Let's put everything together and see how you can pattern match on an optional value.

When handling an optional value, one possibility is to use pattern matching to consider both the presence and the absence of a value:

Listing 22.3: Pattern Matching over Option

```
def sqrt(n: Int): Option[Double] =
  if (n >= 0) Some(Math.sqrt(n)) else None

def sqrtOrZero(n: Int): Double =
  sqrt(n) match { ①
    case Some(result) => result ②
    case None => 0 ③
  }
```

- ① `sqrt(n)` returns a value with type `Option[Double]`
- ② if the value is present, return it
- ③ if the value is missing, return 0

Note that pattern matching is not the only way to handle an optional value: in the next lesson, you'll discover how to achieve the same using predefined higher order functions such as `map` and `flatMap`.

QUICK CHECK 22.2

Write a function called `greetings` that takes an optional custom message as its parameter and returns a string. Use its optional parameter as its greeting message when defined (i.e., it contains a value), use the predefined message "Greetings, Human!" when missing. For example, `greetings(Some("Hello Scala"))` should return the string "Hello Scala", while `greetings(None)` should return "Greetings, Human!".

22.4 Summary

In this lesson, my objective was to teach you about Scala's `Option` type:

- You have discovered how you can use it to represent nullable values and improve the quality

of your code.

- You have learned how to create instances of `Option` and see how to handle them using pattern matching.

Let's see if you got this!

TRY THIS

Define a case class `Person` to represent a person with a first name, an optional middle name, and a last name. Write a function that takes an instance of `Person` as its parameter, and it returns a string describing its full name. For example, when representing a person with first name George, middle name Watson and last name Lucas, it should return the string "George Watson Lucas". On the other hand, when representing a person with first name Martin, no middle name, and last name Odersky, it should return "Martin Odersky".

22.5 Answers to Quick Checks

QUICK CHECK 22.1

A possible implementation for the function `filter` is the following:

```
def filter(text: String, word: String): Option[String] =
  if (text.contains(word)) Some(text) else None
```

QUICK CHECK 22.2

You could implement the function `greetings` as follows:

```
def greetings(customMessage: Option[String]): String =
  customMessage match {
    case Some(message) => message
    case None => "Greetings, Human!"
  }
```

23

Working with Option: map and flatMap

After reading this lesson, you will be able to:

- Transform an element contained in an `Option` using the `map` operation
- Simplify a nested optional structure using `flatten`
- Chain optional values together using `flatMap`

In the previous lesson, you have discovered the type `Option` and how to pattern match on it. After working with optional types for some time, you will realize that some of its operations are particularly recurrent. The class `Option` offers you a set of higher order functions for them to be more productive, and you do not have to use pattern matching every time. This lesson will introduce you to some of the most common and useful predefined functions on `Option`. You'll discover how to transform an optional value using `map`. You'll see how to simplify a nested optional structure using `flatten`. Finally, you'll learn how to combine optional values in an order sequence using `flatMap`. These functions describe patterns common to many Scala types other than `Option`: understanding them is crucial as you'll encounter them in many different contexts. In the capstone, you will use the functions `map` on `Option` to extract data from the winner (if any) of the game "Paper, Rock, Scissors, Lizard, Spock!".

Consider this

What are the fundamental operations that you can perform on an optional value? What if you have more optional values that need to be combined?

23.1 Transforming an Option

Let's start by showing you how you can transform the content of an optional value using the `map`, `flatten`, and `flatMap` methods, which are among the most fundamental and recurrent helper functions defined on `Option`.

Suppose you need to represent the following scenario involving a car and its owner as follows:

- A car has a model, and it may have an owner and a registration plate.
- A person has a name and age, and it may have a driving license.
- A car may have no owner (e.g., when it is brand new).
- A car may have no registration plate if its owner hasn't registered the vehicle with the local authorities yet.
- A person without a driving license is still entitled to purchase a car.

You can translate the above requirements using two case classes, called `Car` and `Person`:

Listing 23.1: The Car and Person case classes

```
case class Car(model: String,
              owner: Option[Person],
              registrationPlate: Option[String])

case class Person(name: String,
                  age: Int,
                  drivingLicense: Option[String])
```

Let's see how the functions `map`, `flatten`, and `flatMap` can help you extracting information on your data set.

23.1.1 The map function

Suppose that you'd like to find the name of the owner of a particular car. You could use pattern matching as follows:

```
def ownerName(car: Car): Option[String] =
  car.owner match {
    case Some(p) => Some(p.name)
    case None => None
  }
```

You can rewrite this function using the function `map` rather than pattern matching as follows:

Listing 23.2: Example usage of map on Option

```
def ownerName(car: Car): Option[String] =
  car.owner.map(p => p.name)
```

In Scala, you refer to the operation of applying a function to the content of an optional value as `map`. The function `map` is a higher order function defined on `Option` that takes a function `f` as its parameter:

- If the optional value is present, it will apply the function `f` to it and return it wrapped as optional value;

- If the optional instance is `None`, it will return it without applying any function.

A possible implementation of `map` for a given `Option[A]` is shown in listing 23.3:

Listing 23.3: The function map on Option

```
def map[B](f: A => B): Option[B] =
  this match { ①
    case Some(a) => Some(f(a))
    case None => None
  }
```

① this is a keyword that refers to the current instance of the class.

Compare the implementation of `map` with your implementation of the function `ownerName` using pattern matching: they look very similar and have the same structure! Let's have another look at the signature of the `map` function for the abstract class `Option[A]`:

```
def map[B](f: A => B): Option[B] = ???
```

You do not need to remember what a `map` function does: its signature informs you of what it does. If you have an instance of `Option[A]` and you have a function `f` that transforms an `A` into a `B`, you can apply the `map` operation to obtain a value of type `Option[B]`. Figure 23.1 provides a summary of how the higher order function `map` operates on an instance of `Option`.

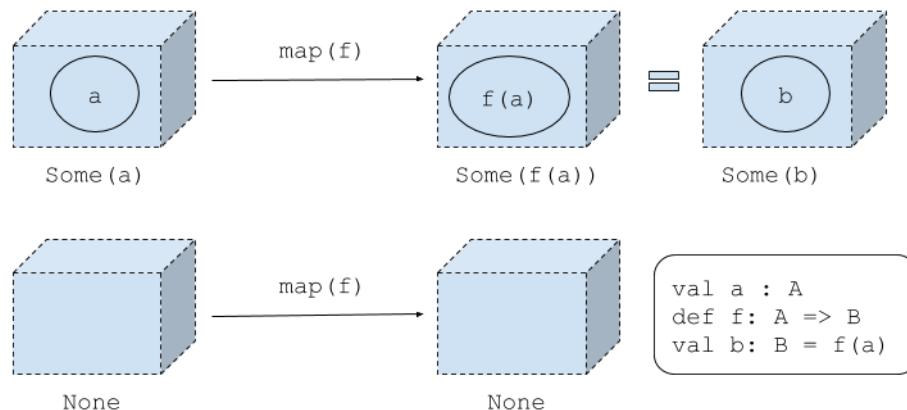


Figure 23.1: Visual representation of the map operation on Option. If the optional value is present, `map` will apply the function `f` to it and wrap its result in a `Some` instance. It will return `None` without applying the function `f` if the optional value is absent.

QUICK CHECK 23.1

Consider the case class `Car` shown in listing 23.1. Write a function called `extractRegistrationPlate` that takes an instance of `Car` and returns an optional registration plate with its text all upper case. Use the function `map` on `Option`.

```
def extractRegistrationPlate(car: Car): Option[String] = ???
```

23.1.2 The flatten function

Suppose you need to retrieve the driving license of an owner of a car, if any. You could achieve this using map as follow:

```
def ownerDrivingLicense(car: Car): Option[Option[String]] =
  car.owner.map(_.drivingLicense)
```

The `ownerDrivingLicense` returns a value of type `Option[Option[String]]`: the first optional type indicates if an owner exists, the second if the owner has a driving license. You may want to keep this distinction in a particular business context, but it doesn't seem natural in a general case. As a human, you expect the value either to be there or not: you do not say that a value "may be there", but simply that a value "may be there".

You can use the `flatten` function to combine two nested optional values as follows:

Listing 23.4: Example usage of flatten on Option

```
def ownerDrivingLicense(car: Car): Option[String] =
  car.owner.map(_.drivingLicense).flatten
```

The function `flatten` acts on an instance of `Option[Option[A]]`, and it returns an `Option[A]`:

- If the outer optional value is a `Some` of a value, return the inner instance of `Option`
- If the outer optional value is `None`, return it

A possible implementation of the function `flatten` on an instance of `Option[Option[A]]` is the following:

Listing 23.5: The function flatten on Option

```
def flatten: Option[A] =
  this match {
    case Some(opt) => opt
    case None => None
  }
```

The function `flatten` merges two optional values into one: if you need to flatten more than two values, you can re-apply it multiple times.

QUICK CHECK 23.2

Write a function called `superFlatten` that takes an instance of `Option[Option[Option[String]]]`, and it returns a value of type `Option[String]` using the function `flatten`.

```
def superFlatten(opt: Option[Option[Option[String]]]): Option[String] = ???
```

23.1.3 The flatMap function

In listing 23.5, you have implemented a function to extract the driving license of a car owner using the function `map` together with the `flatten` function:

```
def ownerDrivingLicense(car: Car): Option[String] =
  car.owner.map(_.drivingLicense).flatten
```

These two operations combined are so common that Scala has created an ad-hoc function for it, called `flatMap`. Listing 23.6 shows you how to rewrite the `ownerDrivingLicense` function using `flatMap` rather than combining `map` and `flatten` operations.

Listing 23.6: Example usage of flatMap on Option

```
def ownerDrivingLicense(car: Car): Option[String] =
  car.owner.flatMap(_.drivingLicense)
```

In Scala, the function `flatMap` is a higher order function on `Option[A]` that applies a function `f`, which returns an optional value itself – in other words, `f` has type `A => Option[B]`:

- It will apply the function `f` to it if the optional value is present. The function `f` returns an instance of `Option`, so you do not need to wrap the result in an optional value.
- It will return it without applying any function if the optional instance is `None`.

A possible implementation of `flatMap` using pattern matching is the following:

Listing 23.7: The function flatMap on Option

```
def flatMap[B](f: A => Option[B]): Option[B] =
  this match {
    case Some(a) => f(a)
    case None => None
  }
```

Have a look at figure 23.2 for a visualization representation of the `flatMap` function on `Option`.

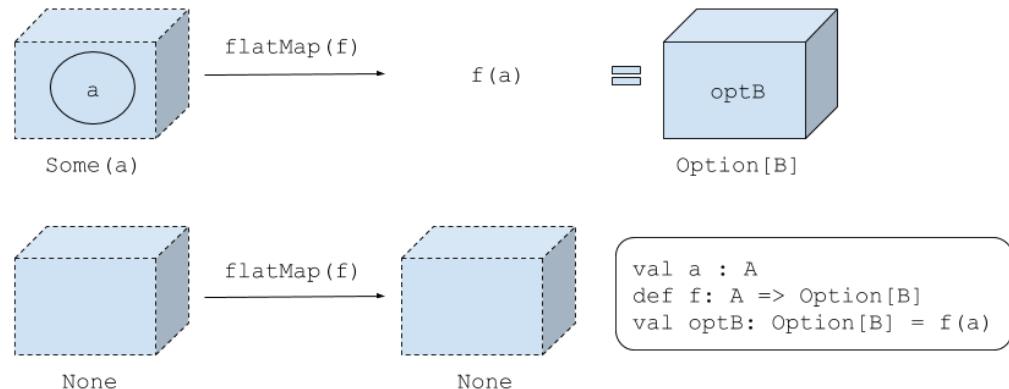


Figure 23.2: Visual representation of the flatMap operation on Option. If the value is present, flatMap will apply the function f to it, which will produce a new optional value. It will return None without applying any function f if absent.

QUICK CHECK 23.3

Write a function called `ownerBelowAge` that takes two parameters: an instance of `Car` and an age parameter of type `Int`. It returns an optional string containing the car owner's name if younger than the given age. Use the `flatMap` function on `Option`.

```
def ownerBelowAge(car: Car, age: Int): Option[String] = ???
```

Let's have another look at the `flatMap` function to understand why it is so powerful. Consider you have an instance of an optional car, and you'd like to extract the driving license of its owner, if any. Its implementation using pattern matching could be similar to the following:

```
def ownerDrivingLicense(optCar: Option[Car]): Option[String] =
  optCar match {
    case None => None
    case Some(car) =>
      car.owner match {
        case None => None
        case Some(person) =>
          person.drivingLicense
      }
  }
```

Having two nested pattern matching constructs makes your code extremely difficult to read. You could get rid of the nested pattern matching by taking advantage of the fact that both `Car` and `Person` are case classes, so you can easily decompose them when pattern matching:

```
def ownerDrivingLicense(optCar: Option[Car]): Option[String] =
  optCar match {
    case Some(Car(_, Some(Person(_, _, drivingLicense)), _)) =>
      drivingLicense
    case None => None
  }
```

This second version of your code is a bit more readable but strictly coupled with your case classes' structure. You need to remember the order of their fields when implementing the function, and you'll need to modify it every time you add, remove, or reorder any of them. Finally, let try to use the `flatMap` function:

Listing 23.8: Chaining optional values with flatMap

```
def ownerDrivingLicense(optCar: Option[Car]): Option[String] =
  optCar.flatMap { car =>
    car.owner.flatMap { person =>
      person.drivingLicense
    }
  }
```

The code is more readable and independent from the specific structure of the involved case classes. In the next lesson, you'll discover another and even more readable way of implementing this function using "for-comprehension".

Because of its unique structure, the `flatMap` function allows you to chain multiple optional operations together. Figure 23.3 provides a visual summary of how you can `flatMap` to combine

multiple optional values in an ordered sequence. Consider you have an instance of Option[A] and due functions `f: A => Option[B]` and `g: B => Option[C]`: by calling `flatMap(f)` you can transform your instance into an Option[B], and then apply `flatMap(g)` to it and obtain an instance of Option[C].

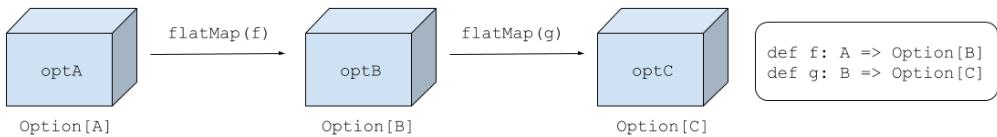


Figure 23.3: Example of chaining optional operations using flatMap. First, you apply `flatMap(f)` to an instance of `Option[A]` to produce an `Option[B]`. Then, you apply `flatMap(g)` to it and obtain an `Option[C]`.

23.2 Summary

In this lesson, my objective was to teach you about transforming an optional value.

- You have seen that the `map` function applies a function to the element of an `Option`.
- You have discovered that you can use the `flatten` function to unify nested optional structures.
- You have learned that the `flatMap` operation is the composition of a `map` followed by a `flatten` function to combine optional values in an ordered sequence.

Let's see if you got this!

TRY THIS

Consider the following scenario:

- A student has an id and a name.
- A student may have a professor assigned as a tutor.
- A professor has an id and a name.
- A professor may have the help of an assistant.
- An assistant has an id and a name.

You can translate the above to code with the following case classes:

```

case class Student(id: Long, name: String, tutor: Option[Professor])
case class Professor(id: Long, name: String, assistant: Option[Assistant])
case class Assistant(id: Long, name: String)

```

Write functions to extract the following information:

1. Retrieve the name of the tutor of a given student
2. Find the id of the assistant of a professor tutoring a given student
3. Return a given student only if having a tutor with a given id

23.3 Answers to Quick Checks

QUICK CHECK 23.1

A possible implementation for the function `extractRegistrationPlate` is the following:

```
def extractRegistrationPlate(car: Car): Option[String] =  
  car.registrationPlate.map(_.toUpperCase)
```

QUICK CHECK 23.2

You can implement the `superFlatten` function by applying the `flatten` function twice as follow:

```
def superFlatten(opt: Option[Option[Option[String]]]): Option[String] =  
  opt.flatten.flatten
```

Notice how much simpler this implementation is compared to its equivalent using pattern matching.

QUICK CHECK 23.3

A possible implementation for the function `ownerBelowAge` is the following:

```
def ownerBelowAge(car: Car, age: Int): Option[String] =  
  car.owner.flatMap { p =>  
    if (p.age < age) Some(p.name)  
    else None  
  }
```

24

Working with Option: for-comprehension

After reading this lesson, you will be able to:

- Chain optional values together using for-comprehension.
- Introduce conditions within for-comprehension constructs
- Code using the most common operations defined on Option

In the previous lesson, you have learned how to use the functions `map`, `flatten`, and `flatMap` to manipulate optional values. In this lesson, you are also going to discover a more readable and elegant way of combining instances of `Option` thanks to a new type of construct, called "for-comprehension". You'll also see how to integrate Boolean conditions to control further how the values are chained together. Finally, you'll discover other useful operations implemented for `Option`, such as `isDefined`, `getOrElse`, `find`, and `exists`. In the capstone, you will use the function `getOrElse` on `Option` to provide an alternative message for your HTTP request when a draw of the game "Paper, Rock, Scissors, Lizard, Spock!" occurs.

Consider this

Assume you have implemented a function to combine many (e.g., five or more) optional values in an ordered sequence using `flatMap`. Can you think of any aspect of your implementation that could make your code difficult to maintain and read?

24.1 For-comprehension on Option

In the previous lesson, you have discovered the `flatMap` function and how you can use it to concatenate optional operation together. Let's have a look at the code in listing 23.8:

```
def ownerDrivingLicense(optCar: Option[Car]): Option[String] =
  optCar.flatMap { car =>
    car.owner.flatMap { person =>
      person.drivingLicense
    }
  }
```

The code is fairly readable, but it requires you to nest many `flatMap` function calls together. Consider that you have to chain five or more optional operations together: your code would look something like the following:

```
opA().flatMap{ a =>
  opB(a).flatMap { b =>
    opC(b).flatMap { c =>
      opD(c).flatMap { d =>
        opE(d)
      }
    }
  }
}
```

We like to refer to this as "rocket coding": you write so many nested operations that force you to indent your code many times, making it difficult to read. Scala has introduced some syntactic sugar (i.e., an alternative syntax to simplify verbose operations) for the `map` and `flatMap` functions called "for-comprehension".

24.1.1 For-comprehension as syntactic sugar for nested map and flatMap calls

Listing 24.1 shows you how you can re-implement your `ownerDrivingLicense` function using for-comprehension:

Listing 24.1: Example of for-comprehension

```
def ownerDrivingLicense(optCar: Option[Car]): Option[String] =
  for {
    car <- optCar
    person <- car.owner
    drivingLicense <- person.drivingLicense
  } yield drivingLicense
```

The expressions `optCar`, `car.owner`, and `person.drivingLicense` are all producing an optional value: they have type `Option[Car]`, `Option[Person]`, `Option[String]` respectively. The values `car`, `person`, and `drivingLicense` are their corresponding extracted values: type `Car`, `Person`, and `String`. You have now have encountered a new keyword, called `yield`: it returns a value that you can compute using the extracted values wrapped into an `Option`. In this case, it returns the value of `drivingLicense` as an optional value. As soon as the for-comprehension finds an absent optional value (e.g., `car.owner` is `None`), it evaluates the entire expression as `None`.

You can use for-comprehension on every class with a `flatMap` function: you'll discover soon that this applies to other types other than `Option`. Thanks to it, your code can avoid an excessive nested structure, which makes it easier to read and understand. You can rewrite the earlier example of chained five or more operations as follows:

```
for {
  a <- opA()
  b <- opB(a)
  c <- opC(b)
  d <- opD(c)
  e <- opE(d)
} yield e
```

QUICK CHECK 24.1

Consider the following snippet of code:

```
def f(n: Int): Option[Int] =
  if (n < 5) Some(n * 2)
  else None

def foo(optA: Option[Int]) =
  for {
    a <- optA
    b <- f(a)
    c <- Some(5 * b)
  } yield c
```

What is the value returned by each of the following function calls? Verify your hypothesis using the Scala REPL.

1. `foo(Some(1))`
2. `foo(Some(5))`
3. `foo(None)`

24.1.2 Filtering values within For-Comprehension

Assume that you have to modify your `ownerDrivingLicence` function to return the `drivingLicense` all uppercase and only for an owner with a given name. You could implement it using `flatMap` as follows:

```
def ownerDrivingLicense(optCar: Option[Car], ownerName: String): Option[String] =
  optCar.flatMap { car =>
    car.owner.flatMap { person =>
      if (person.name == ownerName) person.drivingLicence.map(_.toUpperCase)
      else None
    }
  }
```

Listing 24.2 shows you how you can achieve the same using for-comprehension:

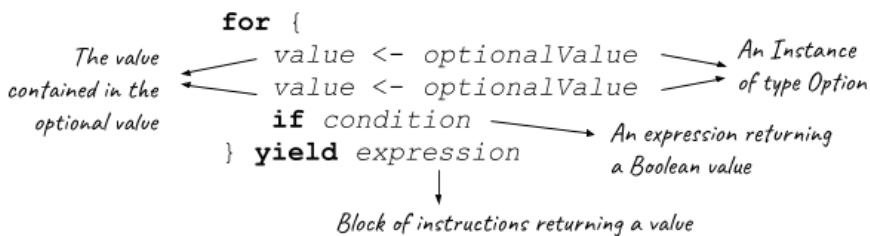
Listing 24.2: Example of for-comprehension with if condition

```
def ownerDrivingLicense(optCar: Option[Car], ownerName: String): Option[String] =
  for {
    car <- optCar
    person <- car.owner
    if person.name == ownerName ①
    drivingLicense <- person.drivingLicense
  } yield drivingLicense.toUpperCase ②
```

① When the condition is false, the chain will stop and cause the expression to return None.

② You can modify the yield value before returning it.

You can modify the yield values in for-comprehension (e.g., `drivingLicense.toUpperCase`) and add conditions to stop the combination of values by adding the keyword `if` followed by a Boolean expression (e.g., `if person.name == ownerName`). Have a look at figure 24.1 for a syntax diagram on for-comprehensions in Scala.



- It returns the expression wrapped into a `Some` instance only if all `Option` instances contain a value

syntactic sugar for

```

optionalValue.flatMap { value =>
  optionalvalue.flatMap { value =>
    if (condition) Some(expression)
    else None
  }
}
  
```

The diagram shows a large downward-pointing arrow with the text "syntactic sugar for" above it. Below the arrow is a block of Scala code. The code uses nested `flatMap` calls to implement the logic of a for-comprehension on an `Option` value. It checks a condition and returns `Some(expression)` if true, or `None` if false.

Figure 24.1: Summary of for-comprehension on `Option`: they allow you to chain optional operation thanks to their syntactic sugar to rewrite nested `map` and `flatMap` function calls.

QUICK CHECK 24.2

In the previous lesson, you have implemented a function `ownerBelowAge` that returns the car owner's name if younger than a given age: re-implement it using for-comprehension.

```
def ownerBelowAge(car: Car, age: Int): Option[String] = ???
```

24.2 Other operations on Option

The functions `map`, `flatten`, and `flatMap` are some of the available operations you can perform on an optional value. Other commonly used functions defined for an instance of `Option[A]` are listed below:

- `isDefined` returns `true` if an optional instance has a value, `false` otherwise.

```
def isDefined: Boolean = ???
Some(1).isDefined // returns true
None.isDefined // returns false
```

- The function `isEmpty` is the opposite of `isDefined`: it returns `true` if an optional instance is absent, `false` otherwise.

```
def isEmpty: Boolean = ???
Some(1).isEmpty // returns false
None.isEmpty // returns true
```

- `getOrElse` returns the optional value if present, otherwise it will execute the provided default operation.

```
def getOrElse(default: A): A = ???
Some(1).getOrElse(-1) // returns 1
None.getOrElse(-1) // returns -1
```

- `find` returns an optional value if its element satisfies a given predicate.

```
def find(predicate: A => Boolean): Option[A] = ???
Some(10).find(_ > 5) // returns Some(10)
Some(1).find(_ > 5) // returns None
None.find(_ > 5) // returns None
```

- The function `exists` combines `find` with `isDefined`: it returns `true` if the value is present and satisfies a given predicate, `false` otherwise.

```
def exists(predicate: A => Boolean): Boolean = ???
Some(10).exists(_ > 5) // returns true
Some(1).exists(_ > 5) // returns false
None.exists(_ > 5) // returns false
```

Think in Scala: do not use the function get on Option

You may have noticed that `Option` has an implementation for a function called `get`: it returns the value if present, it throws a `java.util.NoSuchElementException` if absent. Because it throws an exception, you should consider it unsafe to use and an anti-pattern because the compiler will no longer be able to guarantee that your implementation does not throw exceptions at runtime.

Do not use the function `get` on `Option`. When tempted to do so, you should ask yourself if that type should be optional in the first place. Maybe you should consider resolving the optional wrapper using pattern matching. Perhaps you need to re-evaluate the operations you are performing on an optional value.

For example, consider the following function `foo`:

```
def foo(a: Option[Int]): Int = if (a.isDefined) a.get else 0
```

It would be best if you rewrote it using the `getOrElse` operation on `Option` as follows:

```
def foo(a: Option[Int]): Int = a.getOrElse(0)
```

QUICK CHECK 24.3

Implement a function called `carWithLicensedOwner` that takes and returns an optional car instance if its owner has a driving license.

```
def carWithLicensedOwner(optCar: Option[Car]): Option[Car] = ???
```

24.3 Summary

In this lesson, my objective was to teach you about for-comprehension in Scala.

- You have learned that for-comprehension is an alternative, more readable way to rewrite nested `flatMap` expression to chain optional values together.
- You have also seen how to use `if` conditions to filter the values to consider in a for-comprehension construct.
- You had a quick tour of other useful functions available on an instance of `Option`.

Let's see if you got this!

TRY THIS

Let's reconsider the same scenario you have seen at the end of lesson 23 that describes a Student-Professor-Assistant relation:

You can translate the above to code with the following case classes:

```
case class Student(id: Long, name: String, tutor: Option[Professor])
case class Professor(id: Long, name: String, assistant: Option[Assistant])
case class Assistant(id: Long, name: String)
```

Re-implement these following function using for-comprehension and the other operations on `Option` you have seen in this lesson:

1. Retrieve the name of the tutor of a given student
2. Find the id of the assistant of a professor tutoring a given student

3. Return a given student only if having a tutor with a given id

24.4 Answers to Quick Checks

QUICK CHECK 24.1

The answers are the follows:

1. The expression `foo(Some(1))` evaluates to `Some(10)`: all the optional values are present, so the chain of operations is completed, and its value returned.
2. The function call `foo(Some(5))` returns `None`: `op(5)` returns `None`, causing the chain to break.
3. `foo(None)` returns `None` because the first value of the for-comprehension expression is `None`.

QUICK CHECK 24.2

You can re-implement the function `ownerBelowAge` as follows:

```
def ownerBelowAge(car: Car, age: Int): Option[String] =
  for {
    person <- car.owner
    if person.age < age
  } yield person.name
```

QUICK CHECK 24.3

A possible implementation of the function `carWithLicensedOwner` is the following:

```
def carWithLicensedOwner(optCar: Option[Car]): Option[Car] =
  optCar.find { car =>
    car.owner.flatMap(_.drivingLicense).isDefined
  }
```

25

Tuple and Unapply

After reading this lesson, you'll be able to:

- Group elements using tuples
- Retrieve data from tuples
- Extract information from an instance of a class using the `unapply` method

In the previous lessons, you have discovered how to handle nullable values using the type `Option`. In this lesson, you'll learn about tuples, one of the most basic data structures that Scala offers to quickly group data in a given order. You'll then combine what you have seen about tuples and the type `Option` to discuss the `unapply` method. The function `unapply` is complementary to `apply`: you use the `apply` function to create a class instance and `unapply` to extract information from it. Pattern matching is one of the most powerful tools you can use. So far, you have seen that you can pattern match over raw values (e.g., string, integers, doubles), objects, and case classes. By defining an `unapply` method for a class, you'll also be able to pattern match on them. In the capstone, you'll use tuples to group data together, and you'll define `unapply` methods to pattern match over classes without exposing sensitive information.

Consider this

Suppose you need to represent the concept of a debit card in your application: it must contain the name of its owner, its number, and security code. For security reasons, you must ensure never to expose its owner name and security code. How would you implement it so that you respect its business requirements, but without using the opportunity to pattern match on it?

25.1 Tuples

Suppose you have a website and you'd like to know its number of visits for the last day and month. You can use a third party's API to retrieve several statistics about your website. You call an existing function that looks similar to the following to do so:

```
def get3rdPartyStats(): WebsiteStats = ???  
  
case class WebsiteStats(  
    lastHour: Long,  
    lastDay: Long,  
    lastMonth: Long,  
    lastQuarter: Long,  
    lastYear: Long  
    /* many more fields here! */)
```

The API offers this service for free for a limited number of requests, and it is not particularly performant: for these reasons, you would like to minimize your calls to it.

A first possible implementation could be of calling the function and wrap the information you need in a new case class:

Listing 25.1: Website stats using a case class

```
case class MyStats(lastDay: Long, lastMonth: Long)  
  
def lastDayAndMonthStats(): MyStats = {  
    val allStats = getStats()  
    MyStats(allStats.lastDay, allStats.lastMonth)  
}
```

Suppose you then discover that you need to write another function to extract the stats on your website for the last quarter and year. You could decide to write a similar function returning a new case class, let's call it `MyStats2`, containing the last quarter and last year's data. This approach is not sustainable as it will cause you to create lots of classes to represent the same data many times.

When all you need is grouping some data without creating a specific representation for it, you can use tuples: listing 25.2 shows you how to do this.

Listing 25.2: Website stats using tuples

```
def lastDayAndMonthStats(): (Long, Long) = { ①  
    val allStats = getStats()  
    (allStats.lastDay, allStats.lastMonth) ②  
}
```

① The return type is a tuple containing two elements, both of type `long`

② It builds a tuple of two instances of type `Long`

After defining the function `lastDayAndMonthStats`, you can either refer to its tuple, or decompose it to save its elements:

```
val stats = lastDayAndMonthStats()  
// stats refers to the entire tuple  
  
val (lastDay, lastMonth) = lastDayAndMonthStats()
```

```
// lastDay refers to the first element of the tuple
// lastMonth refers to the second element of the tuple
```

In Scala, a tuple is a tool that allows you to quickly group data that can have a minimum of 1 item and a maximum of 22. You can define it by using round brackets that wrap its items separated by a comma. Tuples have one or more elements. The compiler automatically decomposes a tuple containing only one element, the reason why it evaluates the following expression to true:

```
scala> (1) == 1
res0: Boolean = true
```

The compiler will not simplify tuples with two or more elements. A few examples of tuples are the following:

```
scala> (1,2,3) // tuple with 3 items of type Int
res0: (Int, Int, Int) = (1,2,3)

val a = "hello"
val a: String = hello

scala> (a, 1) // tuple with 2 item of type String and Int respectively
res1: (String, Int) = (hello,1)
```

Scala also has an alternative constructor for tuples containing two elements, called the arrow constructor:

```
scala> 1 -> 2 // tuple with two elements of type Int
res2: (Int, Int) = (1,2)

scala> 1 -> 2 -> 3
res3: ((Int, Int), Int) = ((1,2),3)
// tuple of two elements: a tuple of two Int, and an Int
```

You can use pattern matching to deconstruct a tuple:

```
scala> val t = ("hello", "scala", "!")
t: (String, String, String) = (hello,scala,!)

scala> t match {
    |   case (_, _, c) if c == "!" => "?"
    |   case (a,b,c) => s"$a-$b-$c"
    | }
res4: String = ?
```

You can also use getters for each of its elements based on their order: the function `_1` returns the first one, `_2` the second one, `_3` the third one, and so on.

```
scala> t._1
res5: String = hello

scala> t._2
res6: String = scala

scala> t._3
res7: String = !
```

Using the getter functions defined for tuples is not particularly elegant. It doesn't provide any information about the tuple structure: the item's position in the tuple is usually not indicative of what they represent. A more readable option is to assign the items of tuples to values using value decomposition:

```
scala> val (a, b, c) = ("hello", "scala", "!")
a: String = hello
b: String = scala
c: String = !
```

Suppose you are interested in only extracting some items. In that case, you can use the symbol underscore to indicate the compiler to discard the corresponding value and not to bind it to any value:

```
scala> val (a, b, _) = ("hello", "scala", "!")
a: String = hello
b: String = scala
```

Figure 25.1 summarizes the different ways you can create and decompose a tuple.

2 elements: (a, b) or $a \rightarrow b$
 3+ elements: (a, b, \dots, n)

To extract data from a tuple t:

1) Pattern matching	2) Getters	3) Decomposition
<pre>t match { case (a, b) => ??? case (a, b, ..., n) => ??? }</pre>	<pre>val a = t._1 val b = t._2 ... val n = t._n</pre>	<pre>val (a, b) = t val (a, b, ..., n) = t</pre>

Figure 25.1: A summary of how to define a tuple and extract information from it. You can decompose a tuple using pattern matching or its pre-defined getters or value decomposition.

Think in Scala: When to use Tuples?

Tuples are a great way to group data quickly, but they have several problems related to their lack of expressiveness. Use tuples in functions for temporary grouping of data, but try to avoid using them outside of this context.

Every time you find yourself using tuples as return types of a function, especially if the tuple contains three or more items, you should ask yourself: is this function doing too many things? Try to refactor it by using a more expressive data structure, such as a case class.

Try to use tuples only in short and concise fragments if your program: your code will be more readable and easier to maintain.

QUICK CHECK 25.1

Define a tuple with three elements: the number 5, the string "Jane", the integer 3. Extract the second and third items and multiply them together using the multiplier operator "*".

25.2 Implementing the `unapply` Method

So far, you have discovered the several advantages of having an `unapply` method, but you have never seen how to implement it. Let's recap what it is before seeing how you can code it.

The `unapply` function is a static method defined in the companion object of a class. It is complementary to `apply` to extract information from a class instance. When declaring a case class, the compiler automatically adds an `unapply` method to its companion object. A pattern matching construct uses it to determine which parameters it should consider: you cannot use pattern on a class's parameters if it doesn't have an `unapply` method in its companion object.

Suppose you need to analyze the nutrition facts of a drink. To keep things simple, let's assume that your application needs to use only the metric system: it measures all the liquids in milliliters, other amounts in grams. Your task consists of labeling a drink based on its saturated fat and sugar intake. Somewhere in your code, you have already represented the concept of a drink and its nutrition facts as follows:

```
case class NutritionFacts(
    totalFat: Double, /* grams */
    saturatedFat: Double, /* grams */
    sugars: Double, /* grams */
    salt: Double /* grams */
)

class Drink(
    name: String,
    brand: String,
    size: Double /* milliliter */) {

    def loadNutritionFacts(): NutritionFacts = ???  

        // it retrieves the data from a database or third party
}
```

One possible solution is to define a function to associate a `Label` to an instance of `Drink` similar to the following:

Listing 25.3: Analyzing an instance of `Drink`

```
sealed trait Label ①
case object LowSaturatedFatAndSugar extends Label
case object LowSaturatedFat extends Label
case object LowSugar extends Label
case object HighSaturatedFatAndSugar extends Label

val saturatedFatThreshold: Double = ??? ②
val sugarThreshold: Double = ??? ②

def analyze(drink: Drink): Label = drink.loadNutritionFacts() match {
    case NutritionFacts(_, saturatedFat, sugar, _)
        if saturatedFat < saturatedFatThreshold &&
        sugar < sugarThreshold => LowSaturatedFatAndSugar
    case NutritionFacts(_, saturatedFat, _, _)
        if saturatedFat < saturatedFatThreshold => LowSaturatedFat
    case NutritionFacts(_, _, sugar, _)
        if sugar < sugarThreshold => LowSugar
    case _ => HighSaturatedFatAndSugar
```

```
}
```

- ① The representation of all the possible labels for a drink
- ② The threshold value for saturated fat and sugar

The function `analyze` respects the given requirements, but it is difficult to read due to the numerous amount of `_` to represent the discarded information and the long variable names: you need to distinguish between the concept of total and saturated fat as both data is available during pattern matching.

Let's see how defining an `unapply` method for an instance of `Drink` could improve the readability of your code:

Listing 25.4: Analyzing an instance of Drink using unapply

```
/* Defining a companion object */
object Drink { ①

    def unapply(drink: Drink): Option[(Double, Double)] = { ②
        val nutritionFacts = drink.loadNutritionFacts()
        Some((nutritionFacts.saturatedFat, nutritionFacts.sugars))
    }
}

sealed trait Label
case object LowSaturatedFatAndSugar extends Label
case object LowSaturatedFat extends Label
case object LowSugar extends Label
case object HighSaturatedFatAndSugar extends Label

val fatThreshold: Double = ???
val sugarThreshold: Double = ???

def analyze(drink: Drink): Label = drink match { ③
    case Drink(fat, sugar)
    if fat < fatThreshold &&
    sugar < sugarThreshold => LowSaturatedFatAndSugar
    case Drink(fat, _) if fat < fatThreshold  => LowSaturatedFat
    case Drink(_, sugar) if sugar < sugarThreshold  => LowSugar
    case _  => HighSaturatedFatAndSugar
}
```

- ① Implementing of a companion object for the class `Drink`
- ② Adding an implementation for `unapply` for an instance of `Drink`
- ③ Pattern matching only on the field (saturated) fat and sugar of a drink

The `unapply` method in the companion object `Drink` defines that the compiler only its values for saturated fat and sugar when decomposing a `Drink` instance. This technique allows you to forget many unnecessary details when pattern matching, which simplifies your code greatly. For example, you do not need to remember that your program needs to load the nutrition facts from another source or that your instance contains many other fields. You can also refer to saturated fat as "fat" as the ambiguity between total fat and saturated fat is no longer possible.

You usually define the `unapply` method in the companion object of a class. It always returns a nullable tuple: the tuple represents the information to extract, while `Option` allows you to select no data for specific instances of a type.

The compiler automatically implements an `unapply` method for case classes by extracting all its fields in order of declaration. For example, consider the following case class:

```
case class Person(name: String, age: Int)
```

Have a look at listing 25.5 for the compile-time-generated implementation of its `unapply` method.

Listing 25.5: The `unapply` method of `Person`

```
object Person {

    def unapply(p: Person): Option[(String, Int)] =
        Some((p.name, p.age))
}
```

QUICK CHECK 25.2

Re-implement the `unapply` method for an instance of `Person` so that it is not possible to extract information when pattern matching on any person with the name "James Bond".

25.3 Summary

In this lesson, my objective was to teach you about tuples and how to implement the `unapply` method.

- You have seen how to create a tuple and the different ways you can extract information from it.
- You have also learned how to implement an `unapply` method for a class by combining the concepts of optional values and tuples.

Let's see if you got this!

TRY THIS

Suppose you are writing a program to represent all the books in a public library. Represent a book to have the following information: its title, author, publication date, editor, an ISBN (i.e., a code that uniquely identifies it), and its total page number. You can search a book either by title, author, ISBN, or a combination of the three in the library search engine. Also, all the books in the search engine should have at least ten pages. Implement a book representation for the library and define an `unapply` method to reflect your search engine's requirements.

25.4 Answers to Quick Checks

QUICK CHECK 25.1

The following code multiples the second and third items of a tuple:

```
val (_, name, n) = (5, "Jane", 3)
name * n // returns "JaneJaneJane"
```

QUICK CHECK 25.2

A possible implementation is the following:

```
object Person {  
    def unapply(p: Person): Option[(String, Int)] = {  
        if (p.name.equalsIgnoreCase("James Bond")) None  
        else Some((p.name, p.age))  
    }  
}
```

26

Rock, Paper, Scissors, Lizard, Spock!

In this capstone, you will:

- Implement the main components of your game using case classes and case objects
- Manipulate immutable data using higher order and pure functions
- Represent nullable values using Option
- Use tuples to group elements together
- Define unapply methods to pattern match on classes

In this capstone, you'll implement an HTTP server to play a popular variation of the classic game "Paper, Rock, Scissors!", called "Rock, Paper, Scissors, Lizard, Spock!". Two players play the game by each picking a symbol randomly: the selected symbols determine who the winner is. The allowed moves are Paper, Rock, Scissors, Lizard, Spock, and they interact as follows:

"Scissors cuts paper, paper covers rock, rock crushes lizard, lizard poisons Spock, Spock smashes scissors, scissors decapitates lizard, lizard eats paper, paper disproves Spock, Spock vaporizes rock, rock crushes scissors."

If both players pick the same symbol, your application will not select a winner and declare the game a tie. Have a look at figure 26.1 for a visual representation of the rules for "Rock, Paper, Scissors, Lizard, Spock!".

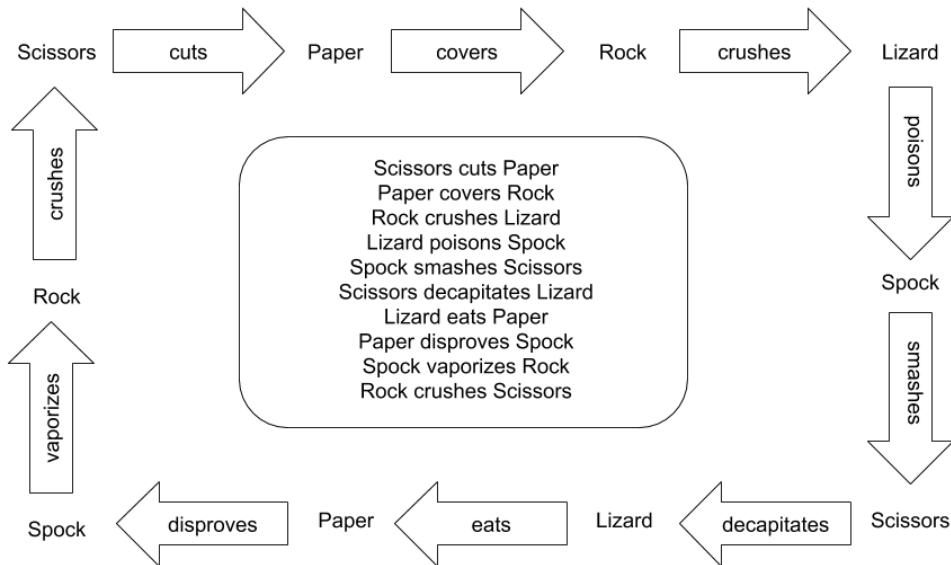


Figure 26.1: “Rock, Paper, Scissors, Lizard, Spock!” has five symbols: Scissors, Paper, Rock, Lizard, Spock. The diagram summarizes the interaction between its symbols: you will use them in your implementation to determine the winner (if any).

This lesson uses Scala 2

The code examples in this lesson use Scala 2 because the library http4s doesn't support Scala 3 yet. The lesson will be updated to Scala 3 before publication as soon as an http4s version for Scala 3 is available.

26.1 Implementing Rock, Paper, Scissors, Lizard, Spock!

Let's define the API's behavior for your HTTP server to play the game “Rock, Paper, Scissors, Lizard, Spock!”. Your application should be able to consume an HTTP POST request to the URI /play. The body of the POST request should provide information about the name and selected symbol of both the two players using the following format:

“nameA: symbolA - nameB: symbolB”

For example, if the player called Daniela selects Spock and the player called Martin picks Paper, the HTTP request should have the body “Daniela: Spock – Martin: Paper”. Your application should parse it and reply with a successful response that provides information about the game's outcome (e.g., “Player Martin with symbol Paper wins!”). Figure 26.2 provides an example of the expected behavior of the API of your HTTP server.

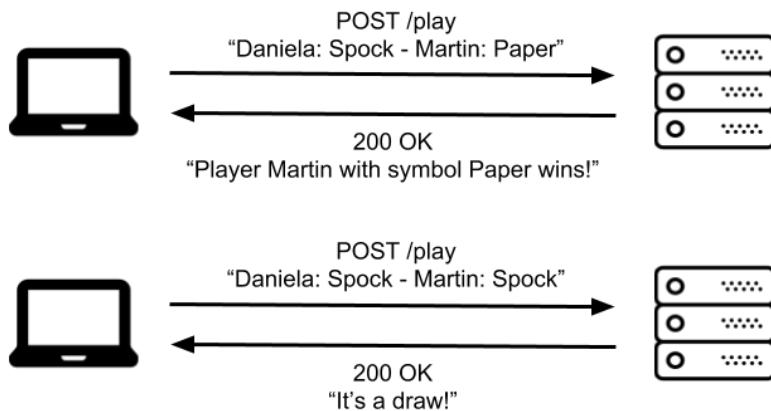


Figure 26.2: Overview of the API for your HTTP server. After parsing the message, the server should respond successfully with a human-readable message containing the game's outcome.

26.1.1 sbt Project Setup and Packages

The first step is to set an empty sbt project up with the external dependencies and logging configurations you need to implement an HTTP server with `http4s`. This process is the same you have seen for capstone 3: if you'd like to refresh your memory on how to do this, please follow the instructions provided in section 18.1.1.

Create a package `org.example.game`: all the files for your application will belong to it. You should also add another package, called `org.example.game.entities` to contain your game's rationale: this allows you to have a clear separation between business logic and server definition.

26.1.2 Defining a Symbol

The first key component of the “Rock, Paper, Scissors, Lizard, Spock!” is the concept of symbol: a player uses it to represent a selected move. You should compare the current symbol to the other to determine if it is the winning one. One possible solution is to represent this with a `trait`:

Listing 26.1: Definition of a Symbol

```
// file src/main/scala/org/example/game/entities/Symbol.scala
package org.example.game.entities

trait Symbol {

    protected def beats: List[Symbol] ①

    def wins(other: Symbol): Boolean =
        beats.contains(other) ②

}
```

① No need to expose the list of symbols that beat the current move: you want to force external users of your code to use the `wins` function.

② The method `contains` belongs to the class `List`, and it returns true if a list contains the given element, false otherwise.

After defining the generic rules of a symbol, let's define specific instances for it. The symbols available in our game are Rock, Paper, Scissors, Lizard, Spock:

Listing 26.2: Creating instances for Symbol

```
// file src/main/scala/org/example/game/entities/Symbol.scala
package org.example.game.entities

sealed trait Symbol { ①
    protected def beats: List[Symbol]

    def wins(other: Symbol): Boolean =
        beats.contains(other)
}

/**
 * Rock crushes Lizard
 * Rock crushes Scissors.
 */
case object Rock extends Symbol { ②
    protected val beats = List(Lizard, Scissors)
}

/**
 * Paper covers Rock
 * Paper disproves Spock
 */
case object Paper extends Symbol { ②
    protected val beats = List(Rock, Spock)
}

/**
 * Scissors cuts Paper
 * Scissors decapitates Lizard
 */
case object Scissors extends Symbol { ②
    protected val beats = List(Paper, Lizard)
}

/**
```

```

    * Lizard poisons Spock
    * Lizard eats Paper
    */
case object Lizard extends Symbol { ②
    protected val beats = List(Spock, Paper)
}

/***
    * Spock smashes Scissors
    * Spock vaporizes Rock
    */
case object Spock extends Symbol { ②
    protected val beats = List(Scissors, Rock)
}

```

- ① **Symbol** is now a sealed trait: you want to prevent your users from defining new instances
 ② You are using case objects rather than objects so you will need to provide a string representation of each symbol that is human-readable.

Finally, let's define how to parse a string into an instance of `Symbol`. Let's create an object to define this logic:

Listing 26.3: The object Symbol

```

// file src/main/scala/org/example/game/entities/Symbol.scala
package org.example.game.entities

[...]

object Symbol {

    def fromString(text: String): Symbol =
        text.trim.toLowerCase match {
            case "rock" => Rock
            case "paper" => Paper
            case "scissors" => Scissors
            case "lizard" => Lizard
            case "spock" => Spock
            case unknown => ①
                val errorMsg = s"Unknown symbol $unknown. " +
                    "Please pick a valid symbol [Rock, Paper, Scissors, Lizard, Spock]"
                throw new IllegalArgumentException(errorMsg)
        }
}

```

- ① It binds the matched text to a value called `unknown` rather than discarding it because it displays it in the error message.

The name of the object in this snippet of code has no particular relevance: you could also rename it to `SymbolUtils` and obtain the same result. You can have an object and a trait with the same name because the Scala compiler is smart enough to understand when referring to one rather than the other.

26.1.3 Representing a Player

Another fundamental concept of your game is the player. A player has a name and a symbol to represent a valid move. You could represent this using a case class, but this would expose its name

and its symbol. Avoid exposing a player's name: you need to prevent your application from using it when selecting a winner. By doing so, you will guarantee that your program doesn't have any biases based on a player's name. You could represent a player using a class as follows:

Listing 26.4: The class Player

```
// file src/main/scala/org/example/game/entities/Player.scala
package org.example.game.entities

class Player(name: String, val symbol: Symbol) {

    override def toString: String = s"Player $name with symbol $symbol"
}
```

Because you have declared `Player` as a class, not as a case class, only its fields marked as `val` are publicly accessible. Do not forget to override the function `toString`: your program should always produce a human-readable representation when converting an instance of `Player` to text. If you fail to do so, the compiler will use `toString`'s default definition and return a string similar to "Player@12345".

You now need to define how to parse some text into a player. One possible solution is to define a companion object for the class `Player` and define an `apply` function for it:

Listing 26.4: The object Player and the apply method

```
// file src/main/scala/org/example/game/entities/Player.scala
package org.example.game.entities

object Player {

    // valid example: "Daniela: Spock"
    def apply(text: String): Player =
        text.split(":", 2) match { ①
            case Array(name, symbol) => ②
                new Player(name.trim, Symbol.fromString(symbol))
            case _ =>
                val errorMsg = s"Invalid player $text. " +
                    "Please, use the format <name>: <symbol>"
                throw new IllegalArgumentException(errorMsg)
        }
}
```

① It splits a text into up to two tokens using ":" as the delimiter and returns an `Array`.

② It matches an instance of `Array` that has exactly two elements. You are going to see more of this when you'll learn about collections.

Finally, let's also define an `unapply` method for `Player` to use pattern match on players.

Listing 26.5: The object Player and the unapply method

```
// file src/main/scala/org/example/game/entities/Player.scala
package org.example.game.entities

object Player {

    [...]
```

```
def unapply(player: Player): Option[Symbol] = Some(player.symbol) ①
}
```

- ① The expressions `Option[Symbol]` and `Option[(Symbol)]` are equivalent because you can always omit tuples that contain only one element.

26.1.4 Defining a Game

You are now ready to represent the concept of a game. A game must have two players, and it must provide an outcome to the game. You could represent a game using a case class as follows:

Listing 26.6: The Game case class

```
// file src/main/scala/org/example/game/entities/Game.scala
package org.example.game.entities

case class Game(playerA: Player, playerB: Player) { ①

    private val winner: Option[Player] =
        (playerA, playerB) match { ②
            case (pA @ Player(sA), Player(sB)) if sA.wins(sB) => Some(pA) ③
            case (Player(sA), pB @ Player(sB)) if sB.wins(sA) => Some(pB) ③
            case _ => None // it's a draw!
        }

    val result: String = winner.map(player => s"$player wins!")
                                .getOrElse("It's a draw!")
}
```

- ① Game is a case class, so all its fields are publicly accessible
 ② Using tuples to pattern match on both players
 ③ You can pattern match on Player instances because its companion object has an unapply function for it.
 ④ It provides a human-readable message to describe the game's outcome

Finally, you can now define how to parse an instance of `Game` from a string. Let's define another `apply` method in its companion object:

Listing 26.7: The Game companion object

```
// file src/main/scala/org/example/game/entities/Game.scala
package org.example.game.entities

[...]

object Game {

    // valid example: "Daniela: Spock - Martin: Paper"
    def apply(text: String): Game =
        text.split("-", 2) match {
            case Array(playerA, playerB) =>
                apply(Player(playerA), Player(playerB)) ①
            case _ =>
                val errorMsg = s"Invalid game $text. " +
                    s"Please, use the format <name>: <symbol> - <name>: <symbol>"
                throw new IllegalArgumentException(errorMsg)
}
```

```
    }
}
```

① Calling apply method that the compiler automatically adds in the companion object Game.

Note that you can have multiple definitions of the `apply` and `unapply` methods in a companion object, as long as they have different signatures. In this code snippet, `Game` is a case class, so its companion object has two `apply` methods: one takes a `String` as its parameter (i.e., the one you have implemented), the other takes two parameters of type `Player` (i.e., the one the compiler generates at compile time).

26.1.5 The API Routes

After coding your business logic, you define the API of your HTTP server. Let's create a class called `GameApi` to describe the shape of the requests it should handle. It should process any POST request to the URI `/play`, parse its body as raw text into a game and reply with code "200 - Ok" containing the game's outcome.

You are now defining your HTTP layer, so you are going to add files to the package `org.example.game` instead of `org.example.game.entities`. A possible implementation of `GameApi` is the following:

Listing 26.8: The GameApi class

```
// file src/main/scala/org/example/game/GameApi.scala
package org.example.game

import cats.effect.IO
import org.example.game.entities.Game
import org.http4s.HttpRoutes
import org.http4s.dsl.Http4sDsl

class GameApi extends Http4sDsl[IO] {

    val routes = HttpRoutes.of[IO] {
        case req @ POST -> Root / "play" => ①
            for { ②
                text <- req.as[String] ③
                response <- Ok(Game(text).result) ④
            } yield response ⑤
    }
}
```

① You need to return an expression of type `IO[Response[IO]]`

② Using for-comprehension over the type `IO`

③ Extracting the body of the request as `String`

④ Creating a successful reply containing the game's outcome

⑤ The assignment `response` has type `Response[IO]`

The type `IO` represents computations that may have side effects. Because it has an implementation for the functions `map` and `flatMap`, you can use for-comprehension to manipulate and combine values: more about this later in the book when you'll discover the type `IO`.

26.1.6 The HTTP Server

The last step of your implementation consists of associating the API routes you have defined to an HTTP server. Let's create a class `GameApp` in the `org.example.game` package as follows:

Listing 26.9: The GameApp class

```
// file src/main/scala/org/example/game/GameApp.scala
package org.example.game

import cats.effect.{ExitCode, IO, IOApp}
import org.http4s.implicits._
import org.http4s.server.Router
import org.http4s.server.blaze.BlazeServerBuilder

import scala.concurrent.ExecutionContext

object GameApp extends IOApp {

    private val httpApp = Router(
        "/" -> new GameApi().routes
    ).orNotFound

    override def run(args: List[String]): IO[ExitCode] =
        stream(args).compile.drain.as(ExitCode.Success)

    private def stream(args: List[String]) =
        BlazeServerBuilder[IO](ExecutionContext.global)
            .bindHttp(8080, "0.0.0.0") ①
            .withHttpApp(httpApp)
            .serve
}
```

① The server runs on localhost on port 8080

The implementation of your Game HTTP server is now complete and ready to run.

26.1.7 Let's try it out!

Congratulations on completing the implementation of “Rock, Paper, Scissors, Lizard, Spock!”. It's now time to try to run your server and play. Navigate to the run folder and execute the command `sbt run`: it will compile and run your server. If everything goes according to plan, you should see that your server is now running on <http://localhost:8080> after a few seconds:

```
[info] Done packaging.
[info] Running org.example.game.GameApp
08:49:33.981 INFO  - Service bound to address /0:0:0:0:0:0:0:8080
08:49:33.999 INFO  -
08:49:34.001 INFO  - [|_|_|_|_|_|_|_|_|_|_|_|_|_|
08:49:34.001 INFO  - |_|`_|_|_|_|_|_|_|_|_|_|_|_|_|_
08:49:34.001 INFO  - |_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_
08:49:34.001 INFO  - _|
08:49:34.211 INFO  - http4s v0.21.9 on blaze v0.14.14 started at http://[0:0:0:0:0:0:0:8080]/
```

Let's send a few POST requests to your server and see if it reflects the given business requirements. For example, suppose you have two players: Leonard and Sheldon. The first time

Leonard selects Rock, while Sheldon picks Spock. Sheldon should win because "Spock vaporizes Rock":

```
$ curl -i -d "Leonard: Rock - Sheldon: Spock" -X POST http://localhost:8080/play
HTTP/1.1 200 OK
Content-Type: text/plain; charset=UTF-8
Date: Mon, 04 Feb 2019 09:07:47 GMT
Content-Length: 38

Player Sheldon with symbol Spock wins!
```

If both players select Spock, the server should inform that is a tie:

```
$ curl -i -d "Leonard: Spock - Sheldon: Spock" -X POST http://localhost:8080/play
HTTP/1.1 200 OK
Content-Type: text/plain; charset=UTF-8
Date: Mon, 04 Feb 2019 09:09:06 GMT
Content-Length: 12

It's a draw!
```

Play a few more rounds of "Rock, Paper, Scissors, Lizard, Spock" and see if you can spot any bugs or possible improvements to your server: let's highlight a few of them in the next section.

26.2 The ugly bits of our solution

Congratulations on implementing your game! Your implementation respects the requirements, but it suffers from a few defects: let's see what these are and what techniques can help you overcome them.

NO ERROR HANDLING

If you try to send a POST request containing a message that the server cannot parse, the server will reply with status code "500 – Internal Server Error" because it doesn't handle exceptions correctly:

```
$ curl -i -d "Leonard - Sheldon" -X POST http://localhost:8080/play
HTTP/1.1 500 Internal Server Error
Connection: close
Date: Mon, 04 Feb 2019 18:42:06 GMT
Content-Length: 0
```

Your application should reply with a response with status code "400 – Bad Request" containing a message that hints at the reason for the failure. Exceptions are unpredictable and surprisingly simple to forget. A better approach is to rely on the compiler to stop your mistakes. Later on in the book, you'll learn about the type `Try`: it marks computations that can fail and it ensures that your program handles them correctly.

DATA FORMAT

Your application does not use a standard format to transmit information between machines. Clients that want to use your server need to learn its data format. Having a custom format also requires some heavy lifting to ensure your application can correctly parse some text: you need to trim it, convert it to all lowercase, split it into tokens. Using a standard data format, such as JSON and

XML, makes your server more straightforward to use. It also makes your code simpler to write because you can rely on external libraries to parse your data. Later on in the book, you will learn how to use `circe`, a popular library to serialize and deserialize data to and from JSON.

26.3 Summary

In this capstone, you have implemented an HTTP server to play the game “Rock, Paper, Scissors, Lizard, Spock!”, a variation of the game “Rock, Paper, Scissors”.

- You have represented the main components of the game using case classes and case objects.
- You have defined pure functions to determine its winner using the `Option` type.
- You have used tuples to group elements and defined `apply` and `unapply` methods to compose and decompose classes.
- You have also seen your code in action and discuss a few possible improvements to your implementation.

Unit 5

List

In Unit 4, you have discovered how to code data with immutable structures and handle nullable values using the `Option` type. In this unit, you'll use the functionalities of the `List` collection to query a publicly accessible dataset, called "The Movies Dataset" by Rounak Banik, which provides information on more than 45000 films. In particular, you are going to learn about the following subjects:

- Lesson 27 teaches you how to define an instance of `List` and add elements to it. You'll also see how to use pattern matching to traverse it and manipulate its items.
- Lesson 28 shows you how to transform its elements and chain multiple sequences using the `map`, `flatten`, and `flatMap` functions.
- Lesson 29 demonstrates how you can query on a list's fundamental properties, such as its size or the characteristics of its items.
- Lesson 30 teaches you different strategies to select a single element from it. You are going to discover how to pick one either by its position or by its features. You are also going to find the minimum/maximum item based on specific criteria.
- Lesson 31 shows you how to filter elements of a list either by position or by feature. You are also going to see how to remove duplicates from a sequence.
- Lesson 32 introduces you to different sorting strategies for a list and how to produce a human-readable representation for it. You are also going to learn how to group elements per feature and how to sum numerical sequences.
- Finally, you'll use all the operations you have discovered on `List` to query a movie dataset and display your results in a consistent and human-readable manner in lesson 33.

After learning about the `List` collection, you'll continue with Unit 6, in which you'll discover other useful collections in the Standard Scala Collection and different strategies of error handling.

27

List

After reading this lesson, you will be able to:

- Define an ordered sequence of items
- Adding elements to an existing list
- Traverse and transform its items using pattern matching

After discovering the `Option` type in the previous unit, you'll learn about `List` in this lesson: many of the concepts you have mastered for optional values are also applicable to lists but in a slightly different context. The type `List` allows you to represent an immutable ordered sequence of elements. This concept is not exclusive to the Scala language. For example, you use an `ArrayList` or a `LinkedList` in Java, a list in C++, a `list` literal in Python, an array in Javascript. You'll find that lists in Scala are relatively similar to many other languages, with a fundamental difference: they are immutable by default. Using mutable lists is still possible but discouraged. In this lesson, you'll learn how to create a list and add elements to it. You'll also see how to pattern match on lists. In the capstone, you'll use lists to represent the information presented in the Movies Dataset.

Consider this

Suppose you need to represent all the books you own. What data structure would you use?

27.1 Creating a List

Imagine you want to write a program to keep track of your contacts. You could present your data using a list data structure as follows:

Listing 27.1: Creating a list of contacts

```
case class Contact(name: String, surname: String, number: String) ①

val martin = Contact(name = "Martin",
                      surname = "Odersky",
                      number = "+123456789")

val daniela = Contact(name = "Daniela",
                      surname = "Sfregola",
                      number = "+987654321")

val contacts = List(martin, daniela) ②
```

① Representation of a contact

② The value contacts has type List[Contact]

A list is a data structure representing a sequence of items of the same kind in a given order. Figure 27.1 shows you its typical recursive structure in which you have the “head” as its first element, followed by all the others represented as its “tail”: you can refer to it as a “linked list”.

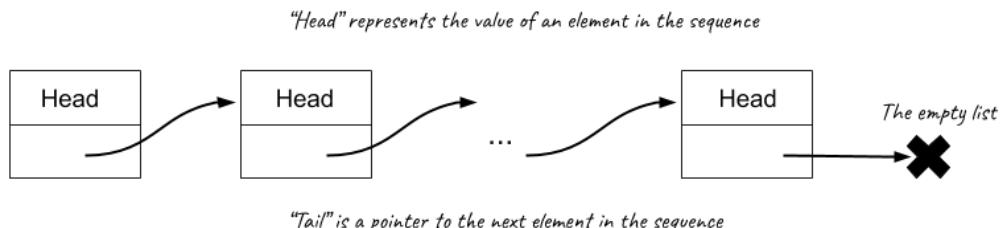


Figure 27.1: The typical structure of a sequence in a given order. Zero or more elements define a list. Each has a “head” containing a value and a “tail” pointing to the next. A symbol for the empty list identifies its end.

In Scala, `List` provides you with a collection that reflects this particular head-tail data structure. Listing 27.2 shows you the (simplified) definition for the class `List`:

Listing 27.2: The List Type

```
package scala

sealed abstract class List[A] { ①
  def head: A
  def tail: List[A]
}

case object Nil extends List[Nothing] { ②
  def head = throw new NoSuchElementException("head of empty list")
  def tail: List[Nothing] =
    throw new UnsupportedOperationException("tail of empty list")
}

case class ::[A](head: A, tail: List[A]) extends List[A] ③
```

- 1 It represents the general structure of a list
- 2 It represents an empty list
- 3 It represents a list that contains at least one element

This implementation has many commonalities with the one for `Option` you have seen in section 22.2. Let's analyze each component of the Scala's `List` implementation:

- package scala

The class `List` lives in the `scala` package, so it's ready for you to use without the need to add an import statement.

- sealed abstract class `List[A]`

`List` is a class with a well-defined set of possible implementations (i.e., `Nil` for an empty `List`, and `::` for a non-empty list). It also has one type parameter that identifies the type of instances it can contain. Note that you cannot initialize an instance of `List` directly because declared as abstract. However, you can write expressions such as "`List(1,2,3)`" thanks to an `apply` method defined into a companion object `List`:

```
object List {
    def apply[A](xs: A*): List[A] = xs.toList
}
```

The expression `A*` uses a special syntax called "varargs", a short name for "variable arguments". Scala has inherited this concept from Java: it indicates that a method can accept zero or more arguments assigned to a named value (e.g., in this specific example, the value `xs`). The class `List` has one type parameter: all the elements in the list must be of the same kind. When defining a `List`, Scala will infer its type parameter by selecting the closest type compatible with all its elements. Let's see a few examples:

```
scala> List(1, 2, 3)
res0: List[Int] = List(1, 2, 3)
// The inferred type is List[Int] as all the elements are of type Int

scala> List("Hello", "Scala", "!")
res1: List[String] = List(Hello, Scala, !)
// The inferred type is List[String]

scala> List(1, 2, 42.24)
res2: List[Double] = List(1.0, 2.0, 42.24)
// The inferred type is List[Double] because the compiler can
// automatically convert elements of type Int to Double,
// but not the other way round: Double "wins the duel".

scala> List(42, "Scala")
res3: List[Any] = List(42, Scala)
// The inferred type is List[Any] because the first common type
// between an Int and a String is Any (i.e., the root of the Scala class hierarchy)
```

- case object `Nil` extends `List[Nothing]`

`Nil` is one of the possible implementations of `List`, and it represents the concept of an empty sequence. It is a case object, which means it is a singleton with a meaningful string representation (i.e., the string “`Nil`”). The concept of an “emptiness” can be associated with any `List`, independently from the type of elements it contains. `Nil` extends `List[Nothing]`, which makes it compatible with any list thanks to `Nothing` being the subclass of any other class:

```
scala> val ints: List[Int] = Nil
ints: List[Int] = List()

scala> val strings: List[String] = Nil
strings: List[String] = List()
```

An empty list does not have any element or a tail, the reason way `Nil`’s implementations for the methods `head` and `tail` throw `NoSuchElementException` exceptions when called.

- `case class ::[A] (head: A, tail: List[A]) extends List[A]`

The class `::`, which some developers refer to as “the Cons class”, is the implementation of `List` to represent a non-empty list. It is a case class, which means the compiler will automatically generate getters for its fields `head` and `tail`. You can create a non-empty list by invoking its `apply` method as follows:

```
scala> new ::(1, ::(2, Nil))
res0: scala.collection.immutable::[Int] = List(1, 2)
```

Building a list by calling the `apply` method on the class `::` can make your code difficult to read: Scala developers rarely use it. Thanks to a cleverly named method defined in the abstract class `List`, you also alternative and more concise way to obtain the same result:

```
scala> 1 :: 2 :: Nil
res1: List[Int] = List(1, 2)
```

Figure 27.2 provides a visual representation of all the possible implementations of the type `List`.

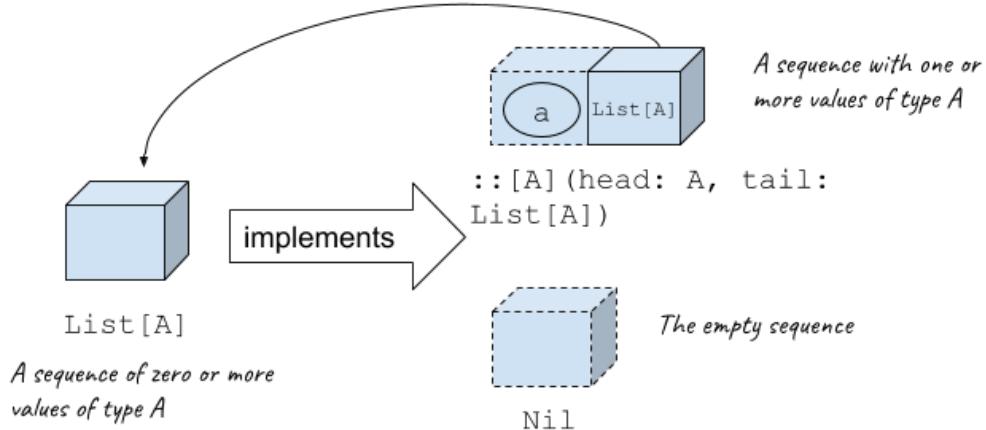


Figure 27.2: Visual summary of the structure of `List` in Scala. You have two possible implementations: `Nil` identifies the empty list, while “`:::`” represents the non-empty list with a head and a tail.

QUICK CHECK 27.1

Using the Scala REPL, create a list containing the number 42 and the nullable string “scala”. What is the type of your `List` instance? Why?

27.2 Adding elements to a List

After learning how to create a list in Scala, let’s discover how you can add elements to it.

Let’s go back to the example scenario you have seen at the beginning of section 27.1 and imagine that you have a new contact to add to your address book. Listing 27.3 shows how to achieve this:

List 27.3: Adding a new Contact

```
val jon = Contact(name = "Jon",
                  surname = "Pretty",
                  number = "+43 3544665 3434")

val moreContacts = contacts :+ jon ①
```

① The value `moreContacts` contains a new list of all the elements in `contacts` together with the `jon` contact

By design, Scala encourages the use of immutable data structures. You’ll obtain a new list rather than modify your existing instance when adding new elements because the class `List` is immutable.

PREPEND TO LIST

Prepend is the operation that adds an element to the beginning of a sequence. It is a particularly efficient operation for `List` as it happens in constant time. It is independent of its size: it requires

only to create a new instance containing the new element as head and the existing list as its tail. In Scala, you can prepend an element using the method “`+:`”:

```
scala> 42 +: List(1, 2, 3)
res0: List[Int] = List(42, 1, 2, 3)
```

APPEND TO LIST

“Append” is the process of adding an element to the end of a list. This operation is more expensive than prepending. Its complexity is proportional to its size: you need to traverse the entire sequence to find its end before adding the new element. In Scala, you can append an element to the operator “`:+:`”:

```
scala> List(1, 2, 3) :+ 42
res1: List[Int] = List(1, 2, 3, 42)
```

CONCATENATING LISTS

You can also merge two lists: you can also refer to this operation as “concatenation”. You can achieve this thanks to the method “`++:`”:

```
scala> List(1, 2, 3) ++ List(42)
res2: List[Int] = List(1, 2, 3, 42)
```

```
scala> List(1, 2, 3) ++ Nil
res3: List[Int] = List(1, 2, 3)
```

Think in Scala: “`+:`” versus “`:+:`”

The operators to prepend (`+:`) and append (`:+:`) are easy to confuse. A simple trick can help you select the correct symbol without relying on the compiler: the character “`:`” is always pointing towards its collection component. A few examples are the following:

`List(1) +: 3 // does not compile!`

`List(1) :-+ 3 // correct`

`3 :+: List(1) // does not compile!`

`3 +: List(1) // correct`

QUICK CHECK 27.2

Use the Scala REPL to prepend the number 42 to the list containing the string “`scala`”.

27.3 Pattern matching on a List

After learning the structure of `List` and its implementations, let’s discover how you can use pattern matching on it.

Imagine you need to define a function to ensure you always have at least two contacts in your address book. Two possible solutions using pattern matching are the following:

Listing 27.4: Pattern matching on List

```
case class Contact(name: String, surname: String, number: String)

def validateContacts(contacts: List[Contact]): List[Contact] =
  contacts match {
    case List() => ①
      throw new IllegalStateException(
        "Invalid empty address book! Please provide at least two contacts")
    case List(a) => ②
      throw new IllegalStateException(
        s"Only contact ${a.name} ${a.surname} found.
Please provide at least another one")
    case cs => cs ③
  }
```

- ① It matches an empty list
- ② It matches a list with exactly one element
- ③ It matches any list

Alternatively, you can also use the notation shown in listing 27.5:

Listing 27.5: Pattern Matching using the “::” operator

```
case class Contact(name: String, surname: String, number: String)

def validateContacts(contacts: List[Contact]): List[Contact] =
  contacts match {
    case Nil => ①
      throw new IllegalStateException(
        "Invalid empty address book! Please provide at least two contacts")
    case a :: Nil => ②
      throw new IllegalStateException(
        s"Only contact ${a.name} ${a.surname} found.
Please provide at least another one")
    case cs => cs ③
  }
```

- ① It matches any empty list
- ② It matches a list with exactly one element
- ③ It matches any list

When pattern matching on a list, you can use two different notations: you can either use the `unapply` method on `List` (e.g., `List()` and `List(a)`) or use the `:::` operator (e.g., `Nil` and `a :: Nil`). The two notations are generally equivalent, and you can mix them within the same pattern matching construct. The `unapply` method on `List` is usually more readable, but the `:::` operator allows you to express more complex conditions. For example, let's assume you need to match over any list that has at least three elements. You can write the following condition thanks to the `:::` operator:

```
case a :: b :: c :: tail => ???
```

When using the `unapply` method on `List`, you need to use the help of an if condition to your case clause to achieve the same result:

```
case list if list.size >= 3 => ???
```

Let's have a look at more complex examples of using pattern matching on a sequence. Another common task consists of using pattern matching to traverse all the list elements and modify them. For example, let's imagine you need to extract the surnames of all the contacts in your address book. You can achieve this using pattern matching as follows:

Listing 27.6: Traversing a list with Pattern Matching

```
case class Contact(name: String, surname: String, number: String)

def getSurnames(contacts: List[Contact]): List[String] = contacts match {
  case Nil => Nil
  case head :: tail => head.surname :: getSurnames(tail)
}
```

The function `getSurnames` returns the empty list if there are no contacts. It extracts the surname for its head element and repeats the `getSurnames` operation on its tail otherwise. This technique of calling the same function but with different parameters is called "recursion".

QUICK CHECK 27.3

Use pattern matching to define a function `sum` that takes a list of integers as its parameter, and it returns an integer representing the sum of all its elements.

27.4 Summary

In this lesson, my objective was to teach you about the basics of the collection `List`.

- You have seen that it has an immutable head-tail structure. In particular, you have discovered Scala provides an implementation for an empty list called `Nil`, and a non-empty one called `::`.
- You have learned how to add elements to a sequence and how to concatenate two lists.
- Also, you have seen the different ways you can pattern match on `List` and use it to manipulate its elements.

Let's see if you got this!

TRY THIS

Define a function to filter all the even numbers of a sequence of integers: pass the list as its parameter and use pattern matching on it.

27.5 Answers to Quick Checks

QUICK CHECK 27.1

You can create the list by typing the following instruction in the Scala REPL:

```
scala> 42 :: Some("scala") :: Nil
res0: List[Any] = List(42, Some(scala))
```

Or alternatively:

```
scala> List(42, Some("scala"))
```

```
res1: List[Any] = List(42, Some(scala))
```

The list has type `List[Any]` because `Any` is the most specific type to represent both `Int` and `Option[String]`.

QUICK CHECK 27.2

The following expressions prepends the number 42 to a list containing the word "scala":

```
scala> 42 +: List("scala")
res0: List[Any] = List(42, scala)
```

Alternatively, you can also do the following:

```
scala> List(42) ++ List("scala")
res1: List[Any] = List(42, scala)
```

QUICK CHECK 27.3

A possible implementation for the function `sum` is the following:

```
def sum(numbers: List[Int]): Int = numbers match {
  case Nil => 0
  case head :: tail => head + sum(tail)
}
```

28

Working with List: map and flatMap

After reading this lesson, you will be able to:

- Transform the elements of a sequence using the `map` function
- Simplify a nested structure using the `flatten` method
- Manipulate and combine lists using the `flatMap` operation
- Chain instances of `List` using for-comprehension

In the previous lesson, you have learned the basics of the type `List` of the Scala Standard Collection library. In this lesson, you'll learn about the basic operations you can perform on lists similar to those you have seen for the class `Option`. You will see how to use the `map` operation to apply a function to a sequence's elements, unify nested lists using `flatten`, and chain them together using `flatMap`. You'll learn how to use for-comprehension to combine and manipulate multiple lists into one. In the capstone, you will use the operations to extract information from the Movies Dataset.

Consider this

Imagine you need to traverse and apply a function to every element of a list. How would you implement it?

28.1 The map, flatten, and flatMap operations

In the previous lesson, you have represented the contacts of your address book using the following representation:

```
case class Contact(name: String, surname: String, number: String)
```

Suppose you now need to change your program to include extra information about each of your contacts. For example, you could track multiple phone numbers for the same contact and label them with a category, as well as store any email address or company name. Listing 28.1 shows you how you could modify your representation of a contact:

Listing 28.1: Representation of a contact

```
case class Contact(name: String,
                    surname: String,
                    numbers: List[ContactNumber],
                    company: Option[String],
                    email: Option[String])

sealed trait Label
case object Work extends Label
case object Home extends Label

case class ContactNumber(number: String, label: Label)
```

Let's see how you can use the `map`, `flatten`, and `flatMap` higher order functions to query your address book's contacts.

28.1.1 The map function

Suppose the first operation you need to implement requires you to extract all the surnames of your contacts. In listing 27.6, you have done this using pattern matching as follows:

```
def getSurnames(contacts: List[Contact]): List[String] = contacts match {
  case Nil => Nil
  case head :: tail => head.surname :: getSurnames(tail)
}
```

However, a more elegant way of achieving the same is using the `map` higher order function:

Listing 28.2 Getting all the contact surnames

```
def getSurnames(contacts: List[Contact]): List[String] =
  contacts.map(contact => contact.surname) ①
```

- ① For each contact in `contacts`, it extract its surname. You can omit the variable `contact` and write “`contacts.map(_surname)`”

In Scala, you use the `map` function to iterate through each element of a list and apply a given transformation. The `map` operation on `List` has a very similar signature to the one defined on `Option`.

```
def map[B](f: A => B): List[B]
```

It is a higher order function that takes a parameter `f`, and it behaves as follows:

- If the list has at least an element, it applies the function `f` to its head, and it recursively invokes the same operation to its tail.
- If empty, it returns `Nil`.

QUICK CHECK 28.1

Define a function `plus5` that adds five to each element of a given list of integers using the `map` function.

```
def plus5(ns: List[Int]): List[Int] = ???
```

28.1.2 The flatten function

Suppose you now need to extract the numbers of all the contacts. You could use the `map` operation as follows:

```
def getNumbers(contacts: List[Contact]): List[List[ContactNumber]] =  
  contacts.map(_.numbers)
```

The function `getNumbers` return type is `List[List[ContactNumber]]`. However, rather than having an unnecessary nested structure, you would like to return an instance of `List[ContactNumber]`. You can use the `flatten` function to combine two sequences into one:

Listing 28.3: Getting all the contact numbers

```
def getNumbers(contacts: List[Contact]): List[ContactNumber] =  
  contacts.map(_.numbers).flatten
```

You can invoke the function `flatten` on nested collections, such as `List[List[A]]`: it concatenates the inner sequences in order to produce a non-nested structure. A few examples are the following:

```
scala> List(List(), List(1, 2), List(3)).flatten  
res0: List[Int] = List(1, 2, 3)  
  
scala> List(List()).flatten  
res1: List[Nothing] = List()
```

QUICK CHECK 28.2

Can you apply the function `flatten` on an instance of `List[Double]`? Why?

28.1.3 The flatMap function

The `flatMap` function combines the `map` and `flatten` operations. You can re-implement your `getNumbers` method from listing 28.3 as follows:

Listing 28.4: Getting all the contact numbers with flatMap

```
def getNumbers(contacts: List[Contact]): List[ContactNumber] =  
  contacts.flatMap(_.numbers)
```

The `flatMap` function on `List[A]` is a higher order function that applies a function `f` of type `A => List[B]` to produce an instance of `List[B]`:

- If the list has at least one element, it applies the function `f` to its head, and it recursively invokes the same operation on its tail.
- If empty, it returns the object `Nil`.

Listing 28.5 shows a possible implementation for the `flatMap` function using pattern matching:

Listing 28.5: The flatMap function on List

```
def flatMap[B](f: A => List[B]): List[B] =
  this match {
    case Nil => Nil
    case head :: tail => f(head) ++ tail.flatMap(f)
  }
```

QUICK CHECK 28.3

Define a method `triple` that takes a list of integers as its parameter and returns a new list with each element from the original sequence repeated three times. For example, when invoking it with `List(1, 2, 3)` it should return `List(1, 1, 1, 2, 2, 2, 3, 3, 3)`. Use the `flatMap` function.

```
def triple(ns: List[Int]): List[Int] = ???
```

Table 28.1 provides a summary of the signature and usage of the basic operations on `List`.

Table 28.1: Summary of the three fundamental operations on List. The function `map` applies a given function to each element of the sequence, while the function `flatten` creates a list by combining two nested structures. The `flatMap` operation combines the `map` and `flatten` operations to chain sequences together.

	Acts on	Signature	Usage
<code>map</code>	<code>List[A]</code>	<code>map(f: A => B): List[B]</code>	It applies a function to each element of the list.
<code>flatten</code>	<code>List[List[A]]</code>	<code>flatten: List[A]</code>	It merges two nested lists into one.
<code>flatMap</code>	<code>List[A]</code>	<code>flatMap(f: A => List[B]): List[B]</code>	The combination of <code>map</code> followed by <code>flatten</code> . It chains sequences together.

28.2 For-comprehension

Let's go back to your address book program. Imagine you now need to select your contacts based on a given list of emails. Listing 28.6 shows a possible implementation for it:

Listing 28.6: Selecting contacts by email

```
def selectByEmails(contacts: List[Contact],
  emails: List[String]): List[Contact] =
  contacts.flatMap { contact =>
    emails.flatMap { email =>
      if (contact.email.exists(_.equalsIgnoreCase(email))) ①
        List(contact)
      else List()
    }
  }
```

```
}
```

① Emails are case-insensitive

A more elegant way of expressing the same is using for-comprehension as follows:

Listing 28.7: Selecting contacts by email using for-comprehension

```
def selectByEmails(contacts: List[Contact],
                   emails: List[String]): List[Contact] =
  for {
    contact <- contacts
    email <- emails
    if contact.email.exists(_.equalsIgnoreCase(email)) ①
  } yield contact
```

① Emails are case-insensitive

You encountered for-comprehension when learning about `Option`: you have now seen that `List` is another Scala type that supports it. In future lessons, you'll discover that are many others! You can rewrite your code to use for-comprehension for every type with a `flatMap` function.

QUICK CHECK 28.4

In the previous Quick Check, you have implemented a function, called `triple`, using `flatMap`: re-implement it using for-comprehension.

28.3 Summary

In this lesson, my objective was to teach you about the basic operations on `List`.

- You have learned that you can use the `map` function to apply some transformation to all the elements of a sequence.
- You have seen that the `flatten` method allows you to merge two nested lists into one.
- You have discovered the `flatMap` function as the combination of the `map` and `flatten` operations to chain sequences together.
- You have also mastered how to manipulate lists using for-comprehension, which is syntactic sugar for one or more nested `flatMap` and `map` operations.

Let's see if you got this!

TRY THIS

Define a function that takes a list of people and extract their names if they are 18 years old or more. Use the following case class to represent a person:

```
case class Person(name: String, age: Int)
```

28.4 Answers to Quick Checks

QUICK CHECK 28.1

A possible implementation for the function `plus5` is the following:

```
def plus5(ns: List[Int]): List[Int] = ns.map(_ + 5)
```

QUICK CHECK 28.2

You cannot apply the function `flatten` on an instance of `List[Double]` because it operates on nested structures only. When trying to do so, the compiler will complain that it cannot convert `Double` to a type that is compatible with `List` (i.e., an instance of `IterableOnce`).

```
scala> List(12.34).flatten
<console>:12: error:
|         No implicit view available from Double => IterableOnce[B]
|         where:   B is a type variable
|             .[B].
```

QUICK CHECK 28.3

An implementation for the function `triple` using `flatMap` is the following:

```
def triple(ns: List[Int]): List[Int] =
  ns.flatMap(n => List(n, n, n))
```

QUICK CHECK 28.4

You can implement the function `triple` using for-comprehension as follows:

```
def triple(ns: List[Int]): List[Int] =
  for {
    n <- ns
    i <- List(n, n, n)
  } yield i
```

29

Working with List: properties

After reading this lesson, you will be able to:

- Inquire over the size of a list
- Check if a sequence contains a specific item
- Count the number of elements that respect a given predicate

In the previous lesson, you have learned how to use the `map` and `flatMap` functions over an instance of `List`. These are two of its fundamental operations: when combined with pattern matching, they allow you to define the vast majority of operations you can perform for a sequence. However, implementing the same procedures over and over may not be performant and challenging to maintain. For these reasons, the class `List` provides you with many ready-to-use and performant functions for many of its common transformations and queries. In this lesson, you'll discover which methods you can use to analyze the properties of a sequence. You'll learn how to get the size of a list and check if it contains a given element. You'll also investigate how many of its items respect a given predicate. In the capstone, you will use these operations to query and analyze the Movies Dataset.

Consider this

Suppose you have a list representing the exam results of a class, and you need to provide statistics to describe the percentage of students that failed, those that passed, and those that excelled. How would you implement it?

29.1 Size of a List

In the previous lesson, you have represented the contacts in your address book using an instance of `List[Contact]`, where each Contact has a name, surname, a list of numbers, and potentially an email and company:

Listing 29.1: Representation of a contact

```
case class Contact(name: String,
                   surname: String,
                   numbers: List[ContactNumber],
                   company: Option[String],
                   email: Option[String])

sealed trait Label
case object Work extends Label
case object Home extends Label

case class ContactNumber(number: String, label: Label)
```

Suppose you need to implement a function to ensure that your address book respects the following library requirements:

- It cannot be empty
- It cannot contain more than one thousand contacts

Listing 29.2 shows you one possible solution for this:

Listing 29.2: Property Operations on List

```
private def reject(msg: String) = throw new IllegalStateException(msg)

private def validateNonEmpty(contacts: List[Contact]): List[Contact] =
  if (contacts.isEmpty) reject("Address book cannot be empty!") ①
  else contacts

private val maxCollSize = 1000
private def validateWithinSize(contacts: List[Contact]): List[Contact] = {
  val size = contacts.size ②
  if (size > maxCollSize)
    reject(s"Address book collection too big! Found $size contacts,
maximum allowed is $maxCollSize")
  else contacts
}

def validateAddressBook(contacts: List[Contact]): List[Contact] = {
  validateNonEmpty(contacts)
  validateWithinSize(contacts)
}
```

① The function `isEmpty` returns true if the list has size zero, false otherwise.

② The method `size` returns the number of elements in the sequence

The first set of operations you'll discover allows you to inquire about the size of a list. These are the following:

- `size` – The function `size` takes no parameters, and it returns the number of elements in the list:

```
scala> List().size
res0: Int = 0

scala> List(1,2,3).size
res1: Int = 3
```

- `isEmpty` – The method `isEmpty` requires no parameters, and it returns true if a sequence has no elements, false otherwise:

```
scala> List().isEmpty
res2: Boolean = true

scala> List(1,2,3).isEmpty
res3: Boolean = false
```

- `nonEmpty` – The function `nonEmpty` is the negation of the function `isEmpty`. It has no parameters, and it returns true for all the lists that have at least one element, false for the empty list:

```
scala> List().nonEmpty
res4: Boolean = false

scala> List(1,2,3).nonEmpty
res5: Boolean = true
```

Have a look at table 29.1 for a summary of the methods regarding the size of a collection.

Table 29.1 The methods to investigate the size of an instance of `List`.

Signature	Description	Example
<code>def size: Int</code>	The size of the sequence	<code>List(1,2).size</code> // it returns 2
<code>def isEmpty: Boolean</code>	It asserts if the list is empty.	<code>List(1,2).isEmpty</code> // it returns false
<code>def nonEmpty: Boolean</code>	It states if the list has at least one element.	<code>List(1,2).nonEmpty</code> // it returns true

QUICK CHECK 29.1

What is the value returned by the following snippet of code?

```
List("").isEmpty
```

29.2 Properties of the elements in a list

Let's imagine you now need to provide functionalities so that you can perform the following queries on your address book:

- Verify if a specific contact is present

- Check if a contact with a given name exists
- Count the number of contacts from a given company

Have a look at listing 29.3 for a possible implementation of these operations:

Listing 29.3: Inquiring of the elements of a list

```
def isPresent(addressBook: List[Contact], contact: Contact): Boolean =
  addressBook.contains(contact) ①

def isPresentByName(addressBook: List[Contact], name: String): Boolean =
  addressBook.exists(_.name == name) ②

def countByCompany(addressBook: List[Contact], company: String): Int =
  addressBook.count(_.company.contains(company)) ③
```

- ① the function `contains` returns true if a given element is in the list, false otherwise.
- ② The method `exists` returns true if at least one item in the list satisfies a given predicate. It returns false otherwise.
- ③ The method `count` counts for how many list elements respect a given predicate.

When inquiring about the elements contained in a list, you can use the following methods:

- `contains` – The method `contains` takes one element as its parameter, and it asserts if it equals to any of the items in the sequence:

```
scala> List().contains("scala")
res0: Boolean = false

scala> List(1, 2, 3).contains("scala")
res1: Boolean = false

scala> List(1, 2, 3).contains(3)
res2: Boolean = true
```

- `exists` – For an instance of `List[A]`, the function `exists` takes a function of type `A => Boolean`. It returns true if any of the elements in the list respects the given predicate:

```
scala> List(1,2,3).exists(_ > 42)
res3: Boolean = false
// Is there an element bigger than 42?

scala> List(1,2,3).exists(e => e > 2 && e < 5)
res4: Boolean = true
// Is there an element bigger than 2, and smaller than 5?
```

- `count` – For an instance of `List[A]`, the method `count` returns the number of elements that respect a given predicate of type `A => Boolean`.

```
scala> List(1,2,3).count(_ > 1)
res5: Int = 2

scala> List(1,2,3).count(_ > 3)
res6: Int = 0
```

Table 29.2 summarizes all the functions you can use to verify if or how many items in the sequence have specific characteristics.

Table 29.2 Recap of the methods you can use to investigate the properties of the elements in an instance of `List[A]`. Where necessary, the method signatures have been simplified to hide non-relevant implementation details.

Signature	Description	Example
<code>def contains(elem: A): Boolean</code>	It verifies if the sequence contains a given element.	<code>List(1,2).contains(3)</code> // it returns false
<code>def exists(p: A => Boolean): Boolean</code>	It asserts if the list contains at least one element that respects a given predicate.	<code>List(1,2).exists(_ > 1)</code> // it returns true

QUICK CHECK 29.2

What is the return value of the following snippets of code? Use the REPL to validate your hypothesis.

1. `List("Welcome", "to", "Scala").contains("scala")`
2. `List("Welcome", "to", "Scala").exists(_.endsWith("me"))`
3. `List("Welcome", "to", "Scala").count(_.contains("o"))`
4. `class A(i: Int); List(new A(1)).contains(new A(1))`
5. `case class B(i: Int); List(new B(1)).contains(new B(1))`

Create an empty list for a given type

When creating an empty list, the compiler struggles to infer its type correctly since it has no elements to help it in its guessing. Without any hint or cast on its intended type, the compiler infers the expression `List()` to be of type

```
List[Nothing]:
scala> List()
res0: List[Nothing] = List()
// the compiler has no type information
// so it infers the type List[Nothing]

scala> val myList: List[Int] = List()
myList: List[Int] = List()
// the compiler assigns it the type List[Int]

scala> List().asInstanceOf[List[Int]]
res1: List[Int] = List()
// Explicit casting from List[Nothing] to List[Int]

For a type A, you can use the method List.empty[A] to create an empty list of type List[A] in a more compact and elegant way:
scala> List.empty
res25: List[Nothing] = List()
// the compiler has no type information
```

```
// so it creates an empty list of type List[Nothing]

scala> List.empty[Int]
res26: List[Int] = List()
// the compiler creates a list of type List[Int]
```

29.3 Summary

In this lesson, my objective was to teach you some of the common operations on a list.

- You have learned how to inquire about the size of a sequence.
- You have seen how to check if it contains a given element or an element with determined characteristics.
- You have also discovered how to count how many list items have certain features.

Let's see if you got this!

TRY THIS

Given a list of people, write a function to ensure an individual with a given name is in it. You should use the following case class to represent a person:

```
case class Person(name: String, age: Int)
```

29.4 Answers to Quick Checks

QUICK CHECK 29.1

The expression returns `false` because the list contains one element: the empty string.

```
scala> List("").isEmpty
res0: Boolean = false
```

QUICK CHECK 29.2

The answers are the following:

1. The compiler evaluates the expression to `false` because the sequence does not contain the word "scala". String equality is case sensitive, so "scala" and "Scala" are considered different.
2. The snippet of code returns `true` because the word "Welcome" ends with "me".
3. The expression returns the integer 2: the words "Welcome" and "to" are the ones containing the string "o".
4. It returns `false` because class equality requires two classes to be the same, if and only if their memory allocation address is the same. For this reason, the following expression returns `false`:

```
scala> new A(1).equals(new A(1))
res0: Boolean = false
```

5. It returns `true` because case class equality requires two classes to be the same if they have the same structure. The following expression returns `true` because `B` is a case class:

```
scala> new B(1).equals(new B(1))
res1: Boolean = true
```

30

Working with List: element selection

After reading this lesson, you will be able to:

- Select the first element of a sequence.
- Pick the nth item of a list.
- Find the first element of a sequence that respects a given predicate.
- Find the minimum/maximum item in a list according to some criteria.

In the previous lesson, you have learned how to analyze the properties of a sequence. In this lesson, you'll discover how to select one item in a list by its position. You'll also learn how to find an item that has specific characteristics. Finally, you'll see how to select the minimum or maximum element in a list based on natural ordering or custom ordering criteria. In the capstone, you'll analyze the Movies Dataset to determine which movie has the highest profit.

Consider this

Consider you have a list of exam scores. How would you find the highest and the lowest ones?

30.1 Selecting an element by its position

Let's continue to expand the functionalities of your address book program, in which each contact is represented as the following:

Listing 30.1: Representation of a contact

```
case class Contact(name: String,
                   surname: String,
                   numbers: List[ContactNumber],
                   company: Option[String],
                   email: Option[String])

sealed trait Label
case object Work extends Label
case object Home extends Label

case class ContactNumber(number: String, label: Label)
```

Imagine you now need to select a contact based on its position in your address book. Listing 30.2 shows you how you can achieve this:

Listing 30.2: Get contact by position

```
def getByPosition(addressBook: List[Contact], n: Int): Contact =
  addressBook.apply(n) ①
```

- ① Alternatively, you can also use the equivalent expression `addressBook(n)`. If the index you provide is invalid, it throws an `IndexOutOfBoundsException`.

When selecting an element in a sequence by its position, you can use the following functions on an instance of the class `List`:

- `apply` – The method `apply` takes the index of the element in a sequence to return. In Scala, indexes always start from zero: the first element will have index 0, the second one index 1, etc. It is an impure function because it throws `IndexOutOfBoundsException` if no item is available for the given index.

```
scala> List(1,2,3).apply(0)
res12: Int = 1
// expression equivalent to List(1,2,3)(0)

scala> List(1,2,3).apply(3)
java.lang.IndexOutOfBoundsException: 3
  at scala.collection.LinearSeqOptimized.apply(LinearSeqOptimized.scala:63)
  ... 28 elided
// expression equivalent to List(1,2,3)(3)
```

- `headOption` – The function `head` and its equivalent `apply(0)` are impure functions as they throw exceptions for empty sequences: the method `headOption` is their pure alternative. When selecting the first element in a list, it returns its first element wrapped in a `Some` if present, otherwise `None` if missing.

```
scala> List().headOption
res10: Option[Nothing] = None

scala> List(1, 2, 3).headOption
res11: Option[Int] = Some(1)
```

Table 30.1 provides a summary of the methods `apply` and `headOption` for the class `List`.

Table 30.1- Summary of the functions for an instance of `List[A]` to select an element by its position.

Signature	Description	Example
<code>def apply(n: Int): A</code>	It returns the element of the sequence at position n. It throws an exception if the index is invalid.	<code>List(0,1,2).apply(3)</code> // it throws an <code>IndexOutOfBoundsException</code>
<code>def headOption: Option[A]</code>	It returns a nullable value containing the first element of the sequence, if any.	<code>List(0,1,2).headOption</code> // it returns <code>Some(0)</code>

QUICK CHECK 30.1

The method `apply` is impure because it throws an exception when the given index does not conform to the sequence's length. Implement a method called `safeApply` as its pure equivalent. Your function should return an optional value other than throwing an exception:

```
def safeApply[A](list: List[A], n: Int): Option[A]
```

30.2 Finding an element with given features

Imagine you'd like to find the first contact with a name starting with a specific text. You can achieve this thanks to the function `find`:

Listing 30.3 Finding Contact by Name

```
def findByName(addressBook: List[Contact], name: String): Option[Contact] =
  addressBook.find(contact => contact.name.startsWith(name)) ①
```

① Assuming here that name is case sensitive

When you need to find an element by its feature rather than its position, you can use the `find` method. For an instance of `List[A]`, the function `find` optionally returns the first element in a sequence that respects a given predicate with shape `A => Boolean`. It returns an item as soon as it finds one, so it can avoid having your program traversing the entire list.

```
scala> List(1,2,3).find(_ > 1)
res11: Option[Int] = Some(2)

scala> List(1,2,3).find(_ > 3)
res12: Option[Int] = None
```

Table 30.2 summarizes when and how to use the function `find` on an instance of `List`.

Table 30.2 When working with a sequence and needing to find an element with specific properties, you should use the function `find`.

Signature	Description	Example
<code>def find(p: A => Boolean): Option[A]</code>	It returns the first element in the collection for which the predicate <code>p</code> is true, if any.	<code>List(0,1,2) .find(_ <= 1) // it returns Some(0)</code>

QUICK CHECK 30.2

Let's consider your addressBook program again. Implement a function `findByCompany` to find one contact from a given company:

```
def findByCompany(addressBook: List[Contact],  
                 company: String): Option[Contact]
```

Ensure to consider the company name case insensitive.

30.3 Picking the minimum or maximum item

Let's consider your address book program again. Suppose you need to find the contact with the shortest full name (i.e., surname and name) and the last one in alphabetical order. Listing 30.4 shows you how to achieve this:

Listing 30.4: Selecting minimum and maximum elements

```
def shortestFullName(addressBook: List[Contact]): Contact =  
  addressBook.minBy { contact => ①  
    contact.surname.length + contact.name.length ③  
  }  
  
def lastContactByFullName(addressBook: List[Contact]): Contact =  
  addressBook.maxBy { contact => ②  
    s"${contact.surname} ${contact.name}" ③  
  }
```

① Ordering per length of surname and name

② Ordering per alphabetical order of surname and name

③ If `addressBook` is empty, it throws an `UnsupportedOperationException`

You can use the below function to select the minimum or maximum element in a list:

- `max` – For an instance of `List[A]` where `A` is a type with a given ordering, the method `max` returns the maximum element in the sequence. In Scala, you define an ordering for a type `A` by providing an implementation for `Ordering[A]` – you'll learn how to specify a custom order for any given type when discussing the implicit language feature. When the list is empty, it throws an `UnsupportedOperationException`. In Scala, many types have a pre-defined order: `String`, `Int`, `BigDecimal`, `Char`, `Byte`, and `Boolean` are some of them.

```
scala> List(1, 2, 3).max
```

```

res25: Int = 3

scala> List(1.4, 2.5, 3.6).max
res26: Double = 3.6

scala> List.empty[Float].max
java.lang.UnsupportedOperationException: empty.max
... 28 elided

scala> List("scala", "hello").max
res27: String = scala
// By default, Scala compares strings in alphabetical order

```

- `min` – The function `min` is complementary to the method `max`. For an instance of `List[A]` where `A` is a type with a given ordering, the method `min` returns the minimum element in the sequence. If the list is empty, it throws an `UnsupportedOperationException`.

```

scala> List(1, 2, 3).min
res28: Int = 1

scala> List(1.4, 2.5, 3.6).min
res29: Double = 1.4

scala> List.empty[Float].min
java.lang.UnsupportedOperationException: empty.min
... 28 elided

scala> List("hello", "scala").min
res30: String = hello
// By default, Scala sorts strings in alphabetical order

```

- `maxBy` – The method `maxBy` returns the maximum element according to a given parameter. For an instance of `List[A]`, it takes a function `f` with type `A => B`. The type `B` must have a given ordering: in other words, it needs to have an implementation for `Ordering[B]`. The function `maxBy` will use the function `f: A => B` to select the maximum element in the sequence. It throws an `UnsupportedOperationException` exception when the list is empty.

```

scala> case class Foo(n: Int, text: String)
defined case class Foo

scala> List(Foo(1, "z"), Foo(9, "a")).maxBy(_.n)
res0: Foo = Foo(9,a)

scala> List(Foo(1, "z"), Foo(9, "a")).maxBy(_.text)
res1: Foo = Foo(1,z)

scala> List.empty[Foo].maxBy(_.n)
java.lang.UnsupportedOperationException: empty.maxBy
... 28 elided

```

When defining a custom ordering rule, you can also use tuples to provide multiple criteria with different priorities. For example, you could pick the maximum element for a sequence of `Foo` instances by looking at their `text` field's length first and then at their alphabetical order:

```
scala> List(Foo(1, "zz"), Foo(9, "a"), Foo(8, "aa")).maxBy { foo =>
  // ordering by the size of text
  // and then by its alphabetical order
  (foo.text.size, foo.text)
}
res2: Foo = Foo(1, "zz")
```

- `minBy` – The function `minBy` is complementary to `maxBy`. For an instance of `List[A]`, it takes a function `f` with type `A => B`, where `B` has a given ordering. The function `minBy` will apply the function `f: A => B` to determine the minimum element in the sequence. When there are no elements, it throws an instance of `UnsupportedOperationException`.

```
scala> case class A(n: Int, text: String)
defined case class A

scala> List(A(1, "z"), A(9, "a")).minBy(_.n)
res33: A = A(1,z)

scala> List(A(1, "z"), A(9, "a")).minBy(_.text)
res34: A = A(9,a)

scala> List.empty[A].minBy(_.n)
java.lang.UnsupportedOperationException: empty.minBy
... 28 elided
```

Table 30.3 provides a summary of the functions to pick the minimum and maximum item in a sequence.

Table 30.3 Summary of the methods to select the minimum or maximum element in an instance for `List[A]` by natural or given ordering criteria. A valid custom ordering predicate is a function that transforms an item of type `A` to a type `B`, where `B` has a known natural ordering. When necessary, I have simplified the signature to hide unnecessary implementation details.

Signature	Description	Example
<code>def max(implicit ord: Ordering[A]): A</code>	It returns the maximum element in a sequence based the element's type natural ordering	<code>List(0,1,2).max</code> // it returns 2
<code>def min(implicit ord: Ordering[A]): A</code>	It picks the minimum item in a list according to the element's type natural ordering	<code>List(0,1,2).min</code> // it returns 0
<code>def maxBy[B](f: A => B)(implicit ord: Ordering[B]): A</code>	It selects the maximum item in a sequence given custom order criteria.	<code>List(0,1,2).maxBy(_ * -12.34)</code> // it returns 0
<code>def minBy[B](f: A => B)(implicit ord: Ordering[B]): A</code>	It returns the minimum element in a list according to custom ordering criteria.	<code>List(0,1,2).minBy(_ * -12.34)</code> // it returns 2

QUICK CHECK 30.3

Consider your address book program and the following snippet of code:

```
def topContact(addressBook: List[Contact]): Contact =
    addressBook.max
```

Does it compile? What does it return? Why? Use the REPL to validate your hypothesis.

30.4 Summary

In this lesson, my objective was to teach you about different approaches to select an item in a sequence.

- You have seen how to pick an element based on its position
- You have discovered how to find an item with specific properties
- Also, you have learned how to select the minimum or maximum element based on different ordering criteria.

Let's see if you got this!

TRY THIS

Imagine you are developing a rating system program for movies. Each movie has a title, a director, its publication year, a short description, and a list of awards it received. Write a function to find the most recent production with the most awards.

30.5 Answers to Quick Checks

QUICK CHECK 30.1

A possible implementation for the function `safeApply` is the following:

```
def safeApply[A](list: List[A], n: Int): Option[A] =
    if (0 < n && n < list.size) Some(list.apply(n))
    else None
```

QUICK CHECK 30.2

You can implement the method `findByCompany` as below:

```
def findByCompany(addressBook: List[Contact],
                  company: String): Option[Contact] =
    addressBook.find { contact =>
        contact.company.exists(_.equalsIgnoreCase(company))
    }
```

QUICK CHECK 30.3

The code does not compile because the compiler does not know it should order a sequence of contacts. It errors with the following message: "No implicit Ordering defined for Contact". You can solve this by using the `maxBy` function and providing an explicit ordering rule. You could also define a natural ordering for contacts: you'll see how to do this in unit 7, in which you'll discover the feature implicits.

31

Working with List: filtering

After reading this lesson, you will be able to:

- Remove or select items from a sequence based on their position
- Filter elements of a list that respect a given predicate
- Remove duplicated items in a sequence

In the previous lesson, you have learned how to select a single element in a sequence based on its position or features. In this lesson, you'll continue to discover other operations you can perform on an instance of `List`. You'll see how to create a new list from an existing one by selecting one of its subsections. You'll discover how to pick elements that have specific characteristics. Finally, you'll learn how to create a new sequence that contains no duplicated items. In the capstone, you'll use these operations to create a subset of movies that have specific characters.

Consider this

Suppose you need to define a function to select all the even numbers in a given sequence of integers. How would you implement it?

31.1 Dropping and Taking elements

In the previous lessons, you have been developing a program to store your contacts. Let's recall that your program represents a contact record as following:

Listing 31.1: Representation of a Contact

```
case class Contact(name: String,
                   surname: String,
                   numbers: List[ContactNumber],
                   company: Option[String],
```

```

    email: Option[String])

sealed trait Label
case object Work extends Label
case object Home extends Label

case class ContactNumber(number: String, label: Label)

```

Let's imagine you need have to select the first n contacts in your address book. Listing 31.2 shows you a possible way of implementing it:

Listing 31.2: First n contacts

```
def firstN(addressBook: List[Contact], n: Int): List[Contact] =
  addressBook.take(n) ①
```

① if you have less than n contacts, it returns the address book unchanged.

When working with lists, you can drop and take elements based on different criteria:

- `drop` – The function `drop` takes an integer n as its parameter, and it creates a new list without its first n items. If the sequence has less than n elements, it returns the empty list.

```

scala> List(1,2,3).drop(1)
res0: List[Int] = List(2, 3)

scala> List(1,2,3).drop(0)
res1: List[Int] = List(1, 2, 3)

scala> List(1,2,3).drop(4)
res2: List[Int] = List()

```

- `take` – The method `take` is complementary to `drop`: it takes an integer n as its parameter, and it creates a new list containing its first n items. If the sequence has less than n elements, it returns the given parameter.

```

scala> List(1,2,3).take(1)
res3: List[Int] = List(1)

scala> List(1,2,3).take(0)
res4: List[Int] = List()

scala> List(1,2,3).take(4)
res5: List[Int] = List(1, 2, 3)

```

- `dropWhile` – For an instance of `List[A]`, the method `dropWhile` creates a new list by removing elements starting from its head until a given predicate `A => Boolean` is respected.

```

scala> List(1,2,3,-1,-2,-3).dropWhile(_ < 2)
res6: List[Int] = List(2, 3)

scala> List(1,2,3,-1,-2,-3).dropWhile(_ < 0)
res7: List[Int] = List(1, 2, 3)

```

- `takeWhile` – The method `takeWhile` is complementary to `dropWhile`. For an instance of

`List[A]`, the function `takeWhile` creates a new sequence by selecting elements from its head until a given predicate `A => Boolean` is verified.

```
scala> List(1,2,3,-1,-2,-3).takeWhile(_ < 2)
res8: List[Int] = List(1)

scala> List(1,2,3,-1,-2,-3).takeWhile(_ < 0)
res9: List[Int] = List()
```

Have a look at table 31.1 for a recap of the drop and take operations on a sequence.

Table 31.1 The functions to take and drop elements either by position or feature for the class `List[A]`. Where necessary, the method signature has been simplified to hide non-relevant implementation details.

Signature	Description	Example
<code>def drop(n: Int): List[A]</code>	It creates a new list by selecting all elements but the first n.	<code>List(0,1,2).drop(1)</code> // it returns <code>List(1,2)</code>
<code>def take(n: Int): List[A]</code>	It creates a new list by picking the first n elements.	<code>List(0,1,2).take(1)</code> // it returns <code>List(0)</code>
<code>def dropWhile(p: A => Boolean): List[A]</code>	It creates a new list by selecting elements until the predicate p is true	<code>List(0,1,2) .dropWhile(_ < 2)</code> // it returns <code>List(2)</code>
<code>def takeWhile(p: A => Boolean): List[A]</code>	It creates a new list by picking elements until the predicate p is true	<code>List(0,1,2) .takeWhile(_ < 2)</code> // it returns <code>List(0,1)</code>

QUICK CHECK 31.1

Using the functions `drop` and `take`, implement the following method to paginate a sequence of strings:

```
def paginate(data: List[String], pageN: Int, pageSize: Int): List[String]
```

To simplify, you can assume that the parameters `pageN` and `pageSize` are positive numbers. If the sequence is not big enough for the requested pagination, return an empty list. For example, given a page size of ten, page 1 should return the first ten elements, while page 2 should return from the 11th element to the 20th included.

31.2 Filtering Items of a List

Let's consider your book address program again, and let's imagine you now need to identify all the contacts that belong to a given company. Listing 31.3 shows you how you could achieve this:

Listing 31.3: All contacts from a given company

```
def fromCompany(addressBook: List[Contact], corp: String): List[Contact] =
  addressBook.filter(contact =>
    contact.company.exists(_.equalsIgnoreCase(corp)) ①
  )
```

① Assuming here that the company name is case insensitive

The class `List` has two functions that allow you to filter elements based on a given criterion:

- `filter` – For an instance of `List[A]`, the method `filter` takes a function of type `A => Boolean` as its parameter, and it returns a new list contains all the elements that respect the predicate.

```
scala> List(1,2,3).filter(_ > 0)
res13: List[Int] = List(1, 2, 3)
```

```
scala> List(1,2,3).filter(_ > 2)
res14: List[Int] = List(3)
```

- `filterNot` – it the complementary of the `filter` method. For an instance of `List[A]`, the function `filterNot` returns a new list containing all the elements that do not respect a given predicate of type `A => Boolean`.

```
scala> List(1,2,3).filterNot(_ > 0)
res15: List[Int] = List()
```

```
scala> List(1,2,3).filterNot(_ > 2)
res16: List[Int] = List(1, 2)
```

Table 31.2 summarizes the methods on a sequence to filter elements based on their features.

Table 31.2 – The functions `filter` and `filterNot` allow you to filter items of an instance of class `List[A]` based on given criteria. Some method signatures are simplified to hide non-relevant implementation details.

Signature	Description	Example
<code>def filter(p: A => Boolean): List[A]</code>	<code>It returns a new sequence containing all the elements for which the predicate p is true</code>	<code>List(0,1,2) .filter(_ <= 1) // it returns List(0,1)</code>
<code>def filterNot(p: A => Boolean): List[A]</code>	<code>It returns a new sequence containing all the elements for which the predicate p is false</code>	<code>List(0,1,2) .filterNot(_ <= 1) // it returns List(2)</code>

QUICK CHECK 31.2

Implement a function that given a list of double, returns a new sequence containing only its non-negative numbers:

```
def filterNonNegative(numbers: List[Double]): List[Double]
```

31.3 Removing duplicates

Imagine that you would like to remove any duplicated contact from your book address. Let's assume that two contacts are the same if and only if they contain the same information. Listing 31.4 shows you an elegant way of doing this:

Listing 31.4: All contacts with no duplicates

```
def removeDuplicates(addressBook: List[Contact]): List[Contact] =
  addressBook.distinct
```

When in need of removing duplicated elements in a list, you can use the method `distinct`: it operates on a sequence by creating a new one containing the same items but without duplicates.

```
scala> List(1,2,3,3,3).distinct
res21: List[Int] = List(1, 2, 3)

scala> List(1,2,3).distinct
res22: List[Int] = List(1, 2, 3)
```

Have a look at table 31.3 for a description of the `distinct` function for a `List`.

Table 31.3 - The method `distinct` removes duplicates from an instance of class `List[A]`.

Signature	Description	Example
<code>def distinct: List[A]</code>	It returns a new collection without duplicate elements	<code>List(1,0,0,1).distinct // it returns List(1,0)</code>

QUICK CHECK 31.3

Consider the following snippet of code:

```
class A(i: Int)
val myList = List(new A(1), new A(2), new A(1))
```

What is the value returned by the expression `myList.distinct`? Why? Use the REPL to validate your hypothesis.

31.4 Summary

In this lesson, my objective was to teach you about creating a sequence by filtering elements of an existing one.

- You have learned how to take or drop elements based on their position or until your program

meets one with a particular feature.

- You have discovered how to select items based on a given predicate.
- Also, you have seen how to remove duplicates from a list.

Let's see if you got this!

TRY THIS

Represent a collection of books, in which each book has a title, a list of authors, and a genre. Possible genres are action, comic, and drama. Implement a function to return all its drama authors: ensure there are no duplicates in the sequence.

31.5 Answers to Quick Checks

QUICK CHECK 31.1

A possible implementation for the function `paginate` is the following:

```
def paginate(data: List[String],
            pageN: Int,
            pageSize: Int): List[String] = {
    val toSkip = (pageN - 1) * pageSize
    data.drop(toSkip).take(pageSize)
}
```

QUICK CHECK 31.2

You can implement the function `filterNonNegative` suing the `filter` function:

```
def filterNonNegative(numbers: List[Double]): List[Double] =
    numbers.filter(_ >= 0)
```

Alternatively, you can also use the `filterNot` method:

```
def filterNonNegative(numbers: List[Double]): List[Double] =
    numbers.filterNot(_ < 0)
```

QUICK CHECK 31.3

The expression `myList.distinct` returns the sequence `myList`. You have declared `A` as a regular class rather than a case class: class equality requires two classes to be the same if an only if their memory allocation address is the same. Notice that the following expression returns `false`:

```
scala> (new A(1)).equals(new A(1))
res0: Boolean = false
```

On the other hand, case class equality only requires two classes to be the same if they have the same structure. If you change the code to declare `A` as a case class, you will receive a sequence containing only two elements when invoking the `distinct` function.

32

Working with List: sorting and other operations

After reading this lesson, you will be able to:

- Sort a list
- Produce a string representation of a sequence
- Sum all the numerical elements of a list
- Group items according to given criteria

In the previous lesson, you have mastered how to filter elements of a sequence. In this lesson, you'll learn about a variety of operations you can perform on lists. You'll discover different approaches to sorting the items of a sequence. Also, you'll learn how to produce a text representation for it. Finally, you'll see how to restructure your list into a dictionary-like structure that groups elements with common features. In the capstone, you'll use these operations to rank the films in the Movies Dataset, and display your data analysis in a human-readable form.

Consider this

Imagine you have a numerical list, and you'd like to sort its elements from its largest number to its smallest one. How would you implement it?

32.1 Sorting elements

In this unit, you have developed a software program to manage an address book in which you have represented each contact as the following:

Listing 32.1: Representing a Contact

```
case class Contact(name: String,
                   surname: String,
                   numbers: List[ContactNumber],
                   company: Option[String],
                   email: Option[String])

sealed trait Label
case object Work extends Label
case object Home extends Label

case class ContactNumber(number: String, label: Label)
```

So far, you have been storing your contacts in no particular order. However, you may want to sort them so that people can easily consult them. Imagine you'd like to sort them alphabetically by surname and name. Have a look at listing 32.2 to discover how to do this:

Listing 32.2: Sorting the contacts alphabetically

```
def sort(addressBook: List[Contact]): List[Contact] =
  addressBook.sortBy { contact =>
    (contact.surname, contact.name) ①
  }
```

① Sorting in alphabetical order by surname first, then by name.

You have several different options when sorting a sequence: the most popular functions to do so are the following.

- `sorted` – For an instance of `List[A]` where `A` is a type with a given order, the function returns a new sequence with its elements ordered accordingly. Let's recall that type `A` has a given order if it has an implementation for `Ordering[A]`.

```
scala> List(0.4, -2, 3).sorted
res39: List[Double] = List(-2.0, 0.4, 3.0)

scala> List("my", "example").sorted
res40: List[String] = List(example, my)

scala> List.empty[Double].sorted
res41: List[Double] = List()
```

- `sortBy` – The method `sortBy` sorts the elements of a sequence according to a given criterion. For an instance of `List[A]`, the function `sortBy` takes a function `f` with type `A => B` as its parameter, where the type `B` must have a defined ordering. You can combine multiple criteria by using a tuple: the first criteria in the tuple will have priority over the second one, and so on.

```
scala> List(0.4, -2, 3).sortBy(i => -i)
res42: List[Double] = List(3.0, 0.4, -2.0)
// Ordering doubles in descending order

scala> case class A(n: Int, text: String)
defined case class A
```

```
scala> List(A(1, "z"), A(9, "a"), A(1, "a")).sortBy(e => (e.text, e.n))
res43: List[A] = List(A(1,a), A(9,a), A(1,z))
// Ordering by the field text first, then by n
```

- `reverse` – The method `reverse` returns a new list containing elements in reverse order.

```
scala> List(1, 3, 2).reverse
res44: List[Int] = List(2, 3, 1)

scala> List.empty[String].reverse
res45: List[String] = List()
```

How to shuffle items in a list

Imagine you have a sequence, and you'd like to change the order of its elements randomly. You can achieve this thanks to the `scala.util.Random` class as follows:

```
scala> import scala.util.Random
import scala.util.Random

scala> Random.shuffle(List(1, 2, 3))
res0: List[Int] = List(2, 1, 3)

scala> Random.shuffle(List(1, 2, 3))
res1: List[Int] = List(3, 2, 1)
// calling Random.shuffle multiple times returns a different result!
```

Table 32.1 provides a summary of the methods you can use to sort the elements of a list.

Table 32.1: The different functions you can use to sort an instance of the class List[A].

Signature	Description	Example
<code>def sorted(implicit ord: Ordering[A]): List[A]</code>	It sorts a list according to its elements' type natural ordering. Type A must have a natural ordering, that is an implementation of Ordering[A]	<code>List(1,0,2).sorted // it returns // List(0,1,2)</code>
<code>def sortBy[B](f: A => B)(implicit ord: Ordering[B]): List[A]</code>	It sorts the sequence according to a given custom ordering criteria.	<code>List(1,0,2) .sortBy(_ * -10) // it returns // List(2,1,0)</code>
<code>def reverse: List[A]</code>	It returns a new list in which its elements are in reverse order.	<code>List(1,0,2) .reverse // it returns // List(2,0,1)</code>

QUICK CHECK 32.1

Consider the following snippet of code:

```
List().sorted
```

Does it compile? If so, what does it return? Why? Use the REPL to validate your hypothesis.

32.2 Converting a List to a String

Now that you have seen how to sort your contacts, you may want to display a summary representation for them. For example, you may want to produce a text to list the first n contacts' surname and name. Listing 32.3 shows you a possible way of doing so:

Listing 32.3: Pretty representation of Contact

```
case class Contact(name: String,  
                   surname: String,  
                   numbers: List[ContactNumber],  
                   company: Option[String],  
                   email: Option[String]) {  
  
  def toPrettyString: String = s"$surname $name" ①  
}  
  
def describeFirstN(n: Int, addressBook: List[Contact]): String =  
  addressBook.take(n).map(_.toPrettyString).mkString("\n") ②
```

① Rather than using the default implementation of `toString` for the case class `Contact`, you define an alternative string conversion

② Invoking `toPrettyString` for each contact and concatenating the results with `\n`.

When building a string that represents a sequence and its items, you can use the function `mkString`. It returns a string representing the list and its elements: it converts each element into text by invoking the `toString` method, which the compiler ensures it exists for every instance, and it concatenates them using a separator, set by default to the empty string `"`. You can also provide strings to use as prefix and suffix of the produced string:

```
scala> List("Hello", "Scala").mkString
res17: String = HelloScala
// Using the default separator ""

scala> List("Hello", "Scala").mkString(", ")
res18: String = Hello, Scala
// Using the separator ", "

scala> List("Hello", "Scala").mkString("[", "-", "]")
res19: String = [Hello-Scala]
// Using the separator "-", "[" as prefix, and "]" as suffix

scala> List().mkString("[", "-", "]")
res20: String = []
// Using the separator "-", "[" as prefix, and "]" as suffix
```

Have a look at table 32.2 for a technical summary for the function `mkString` on an instance of `List[A]`.

Table 32.2: The function `mkString` allows you to produce a string representation of a sequence.

Signature	Description	Example
<code>def mkString(start: String, sep: String, end: String): String</code>	<p>It produces a string representing the list starting with the string <code>start</code> and ending with the string <code>end</code>. It converts its elements by invoking the method <code>toString</code> on each of them and concatenating the results using the string <code>sep</code>.</p>	<code>List(0,1,2) .mkString("{", ", ", "}") // it returns // "{0,1,2}"</code>
<code>def mkString(sep: String): String</code>	<p>The equivalent of invoking the function <code>mkString(start, sep, end)</code> as <code>mkString("", sep, "")</code>.</p>	<code>List(0,1,2) .mkString(",") // it returns // "0,1,2"</code>
<code>def mkString: String</code>	<p>It produces the same effect of invoking the function <code>mkString(start, sep, end)</code> as <code>mkString("", "", "")</code>.</p>	<code>List(0,1,2).mkString // it returns "012"</code>

QUICK CHECK 32.2

Consider the following two snippets of code: what value does each of them produce? Use the REPL to confirm your hypotheses.

```
1. class A(i: Int)
   List(new A(0), new A(1), new A(2)).mkString(",")
2. case class B(i: Int)
   List(new B(0), new B(1), new B(2)).mkString(",")
```

32.3 Sum elements of numerical sequences

Let's consider your address book program again. Suppose you need to compute many numbers are stored in your device, keeping in mind that each contact may have zero or more phone numbers. Have a look at listing 32.4 for a possible solution on how to achieve this:

Listing 32.4: Total phone numbers stored

```
def totalNumbers(addressBook: List[Contact]): Int =
  addressBook.map(_.numbers.size).sum ①
```

① Counting how many numbers each contact has and summing all of them up

When working with numerical sequences, you can sum its elements using the function `sum`. For an instance of `List[A]` where `A` is a numeric type, the method `sum` returns a value of type `A` representing the sum of its numbers. In Scala, a type `A` is numeric if it has an implementation for `Numeric[A]`. You'll revisit this concept in unit 7, in which you'll learn about the keyword `implicit`, and you'll show you how to create your custom numeric type. The types `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, and `BigInt` are examples of numeric types in Scala.

```
scala> List(1, 2, 3).sum
res22: Int = 6

scala> List(1.4, 2.5, 3.6).sum
res23: Double = 7.5

scala> List.empty[Float].sum
res24: Float = 0.0

scala> List("hello", "scala").sum
<console>:13: error: could not find implicit value for parameter num: Numeric[String]
// The compiler could not find an implementation for Numeric[String]
// since String is not a numeric type
```

Table 32.3 provides a summary of the method `sum` for an instance of `List[A]`.

Table 32.3: You can use the method `sum` to sum the elements of a numerical sequence.

Signature	Description	Example
<code>def sum(implicit num: Numeric[A]): List[A]</code>	It sums the elements of a numerical list. A type A is numerical if it has an implementation for <code>Numeric[A]</code> .	<code>List(0,1,2).sum</code> // it returns 3

QUICK CHECK 32.3

Write a function `sumOfFirstN` to sum all numbers for 0 to n inclusive:

```
def sumOfFirstN(n: Int): Int
```

For example, `sumOfFirstN(10)` should return 55, and `sumOfFirstN(-10)` should return 0.

HINT: You can generate a sequential structure containing all numbers from 0 to n inclusive using the operator `to`. For example, the expression `0 to 3` returns a range containing the numbers 0, 1, 2, and 3.

32.4 Grouping elements by feature

Imagine you now need to display contacts per company. Listing 32.5 shows you how to do this:

Listing 32.5: Contacts per company

```
def perCompany(
  addressBook: List[Contact]): Map[Option[String], List[Contact]] = ①
  addressBook.groupBy(_.company)
```

- ① The function returns a key-value structure called “map” where the key is the nullable company name (i.e., “`Option[String]`”), and the value is the list of contacts with that company value (i.e., “`List[Contact]`”).

You can use the method `groupBy` on a sequence to group elements based on their characteristics by producing a key-value structure called “map”. For an instance of `List[A]`, the function `groupBy` takes a parameter `f` with type `A => B`, and it returns a value of type `Map[B, List[A]]` – you’ll learn about the type `Map` in the next unit. The function `groupBy` uses the function `f` to determine each key of the dictionary and its corresponding values.

```
scala> case class A(n: Int, text: String)
defined case class A

scala> List(A(1, "z"), A(9, "a")).groupBy(_.text)
res0: scala.collection.immutable.Map[String,List[A]] =
  Map(z -> List(A(1,z)), a -> List(A(9,a)))
// A dictionary with two keys: "z" and "a"

scala> List("hello", "world", "scala").groupBy(_.length)
res1: scala.collection.immutable.Map[Int,List[String]] =
  Map(5 -> List(hello, world, scala))
```

```
// A dictionary containing the key 5

scala> List("hello", "world", "scala").groupBy(_.contains('a'))
res2: scala.collection.immutable.Map[Boolean,List[String]] =
  Map(false -> List(hello, world), true -> List(scala))
// A dictionary with two keys: false and true

scala> List.empty[String].groupBy(_.contains('a'))
res3: scala.collection.immutable.Map[Boolean,List[String]] =
  HashMap()
// The empty dictionary
```

Option and String as special implementations of List

The Scala `List` collection is full of useful functions that make it versatile to use. Many of the methods you have seen are defined for several other types, not just `List`! For example, you will see that both `Set` and `Map` have a filter operation.

The compiler makes these functionalities available for all the types that can automatically convert to a sequence-like structure. You can consider the type `String` as a sequence of characters and perform the following operations:

```
scala> "scala".max
res0: Char = s
// 's' is the char with the highest ASCII code
```

```
scala> "scala".min
res1: Char = a
// 'a' is its char with the lowest ASCII code
```

You can also see an `Option` type as a special case of `List` that has either zero or one element:

```
scala> Some(5).filter(_ < 3)
res2: Option[Int] = None
```

```
scala> Some(5).size
res3: Int = 1
```

Table 32.4 provides a summary of the `groupBy` function for an instance of `List[A]`.

Table 32.3: The method `groupBy` allows you to group elements of a list based on their features.

Signature	Description	Example
<code>def groupBy[K](f: A => K): Map[K, List[A]]</code>	It groups the items of a sequence according to the computation of a value <code>K</code> .	<code>List(0,1,2) .groupBy(_ % 2) // it returns // Map(// 0 -> List(0,2), // 1 -> List(1))</code>

QUICK CHECK 32.4

Implement a function `perLetter` to group your address book contacts according to the first letter of their surname:

```
def perLetter(addressBook: List[Contact]): Map[Char, List[Contact]]
```

You should add those contacts with an empty surname to a category identified by the space char ''.

32.5 Summary

In this lesson, my objective was to teach you about operations you can perform on a list.

- You have discovered different strategies to sort its elements.
- You have mastered how to use the `mkString` function to produce a more expressive string representation for your sequence.
- You have learned how to sum numerical lists
- Also, you have seen how to group elements with features in common thanks to the `groupBy` function.

Let's see if you got this!

TRY THIS

Imagine you are building a program to mark exams. Assume that a mark has an exam name, a score, and a student id. Write a function that takes a sequence of marks and prints a human-readable message to the console containing the top five scores' student id.

32.6 Answers to Quick Checks

QUICK CHECK 32.1

The snippet of code `List().sorted` does not compile:

```
scala> List().sorted
<console>:13: error: diverging implicit expansion for type scala.math.Ordering[B]
starting with method Tuple9 in object Ordering
      List().sorted
           ^
```

The snippet `List()` has type `List[Nothing]`. When invoking the function `sorted` on it, the compiler looks for an instance of `Ordering[Nothing]` to use in the ordering: the compiler cannot find exactly one, so it rejects the expression as valid.

QUICK CHECK 32.2

The first snippet produces a string value similar to the following:

```
res0: String = A@ed2f2f6,A@7c281eb8,A@65f40689
```

The second one produces a more readable text:

```
res1: String = B(0),B(1),B(2)
```

The function `mkString` invokes the function `toString` for each element in the list. Class `B` is a case class: the compiler changes its `toString` implementation to describe its structural composition. On the other hand, `A` is a regular class: its `toString` method refers to its default implementation in `java.lang.Object` which returns a text containing the class name and the memory address of the instance.

QUICK CHECK 32.3

A possible implementation for the function `sumOfFirstN` is the following:

```
def sumOfFirstN(n: Int): Int = (0 to n).sum
```

The expression `0 to n` produces a sequence like structure called “inclusive range” containing all the numbers from `0` to `n` inclusive. You can invoke the `toList` function to convert a range to a list:

```
scala> val range = 0 to 10
range: scala.collection.immutable.Range.Inclusive = Range 0 to 10
scala> range.toList
res0: List[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

QUICK CHECK 32.4

You can implement the function `perLetter` as the following:

```
def perLetter(addressBook: List[Contact]): Map[Char, List[Contact]] =
  addressBook.groupBy(_.surname.headOption.getorElse(' '))
```

33

The Movies Dataset

In this capstone, you will:

- Define ordered sequences of elements.
- Transform and count the items of a list.
- Find the minimum and maximum elements according to specific features.
- Filter items based on their characteristics and selecting them based on their position.
- Sort lists and produce string representation for them.

In this capstone, you'll analyze data for more than 45000 movies. The information is a subset of a popular and publicly accessible dataset called "The movies Dataset" by Rounak Banik. On its website, you can find its latest version as well as an extensive description of its content:

These files contain metadata for all 45,000 movies listed in the Full MovieLens Dataset. The dataset consists of movies released on or before July 2017. Data points include cast, crew, plot keywords, budget, revenue, posters, release dates, languages, production companies, countries, TMDB vote counts and vote averages.

This dataset also has files containing 26 million ratings from 270,000 users for all 45,000 movies. Ratings are on a scale of 1-5 and have been obtained from the official GroupLens website.

From <https://www.kaggle.com/rounakbanik/the-movies-dataset>

For this capstone, you'll focus on a subset of its data contained in a file called movies_metadata.csv. Its rows provide information on movies, such as their title, language, release date, vote average, and popularity: have a look at table 33.1 for a list of their properties you'll consider for this capstone.

Table 33.1 Summary of the features for a movie from The Movie Dataset” by Rounak Banik that you’ll consider for this capstone. For a full list of the analyzed characteristics, have a look at <https://www.kaggle.com/rounakbanik/the-movies-dataset> or the header of movies_metadata.csv file.

Feature	Description	Format	Nullable
genres	The list of genres the movies belongs to	JSON	NO
id	Its unique identifier	Int	NO
imdb_id	External reference for the IMDB dataset	String	NO
original_language	The original language of the movie	String	NO
original_title	Its original title	String	NO
tittle	Its English title	String	YES
overview	A short description of the plot	String	NO
popularity	Popularity score for the movie	Float	YES
release_date	Its release date	Local Date	YES
revenue	Revenue in USD for the movie	Int	YES
budget	Budget in USD for the movie	Int	NO
runtime	Its duration in minutes	Double	YES
vote_average	Average vote from reviewers	Float	YES
vote_count	Number of reviewers that rated the film	Float	YES

In this capstone, you will interrogate the dataset to discover information about these movies. In particular, you'll find answers to the following ten questions:

1. How many movies are there in the dataset?
2. How many of them were released in 1987?
3. Find the top five productions per vote average and count with at least 50 votes
4. Find the top five ones per popularity
5. Select five non-English films
6. Which movie made the most profit?

33.1 Download the base project

Rather than creating an empty sbt project from scratch, let's use `git` to checkout code that you'll use as the starting point for your capstone. You can ensure that `git` is available on your machine by executing the following command in your terminal:

```
$ git --version
git version 2.21.1 (Apple Git-122.3)
```

If needed, look at section 2.4.1 of lesson 2 for instructions on installing `git` on your machine. You can now navigate to an empty folder of your choosing and checkout the remote branch containing the code:

```
$ git init
$ git remote add daniela https://github.com/DanielaSfregola/get-programming-with-scala.git
$ git fetch daniela
$ git checkout -b my_lesson33 daniela/baseline_unit5_lesson33
```

The code you've downloaded is an sbt project with a few files ready to use. The files `project/build.properties` and `build.sbt` provide information on the sbt and Scala versions to use and which external dependencies to use. The `src/main/resources/movies_metadata.csv` is the resource containing the movie data that you'll analyze. It provides many fields for each movie, but you'll focus only on some of them. Finally, the `src/scala` folder contains some useful implementations:

- `org.example.movies.MoviesDataset` uses an external dependency, called `scala-csv`, to read the file and parse each line into a `Movie` instance by invoking the `org.example.movies.entities.Movie.parse` function.
- `org.example.movies.entities.Parsers` contains several functions to parse string values. Each parse function tries to convert some text into a specific type value. It returns `None` in case of failure, and it wraps the parsed value into a `Some` otherwise. These parser functions take advantage of the `Map` and `Try` types and the `circe` library to parse JSON objects, which are topics that you'll learn about in the next units of the book.
- `org.example.movies.entities.Movie` provides the structure of a movie and its fields you should consider in your analysis. You will provide an implementation for its `parse` function by combining the provided base parse functions.

Let's compile your project by executing the command `sbt compile` to download all its external dependencies and ensure the correctness of the code you have so far. You are now ready to start coding!

33.2 Parsing a Movie entity

Let's begin by completing the implementation for the `Movie` entity. Listing 33.1 shows you the initial content of the `Movie.scala` file:

Listing 33.1: The initial content of Movie.scala

```

package org.example.movies.entities

import java.time.LocalDate
import org.slf4j.LoggerFactory

case class Genre(id: Int, name: String) ①

case class Movie(genres: List[Genre], ②
                 id: Int,
                 imdbId: String,
                 originalLanguage: String,
                 originalTitle: String,
                 title: String,
                 overview: String,
                 popularity: Option[Float],
                 releaseDate: Option[LocalDate],
                 revenue: Int,
                 budget: Int,
                 duration: Option[Double],
                 voteAverage: Float,
                 voteCount: Float)

object Movie {
  import Parsers._

  private val logger = LoggerFactory.getLogger(this.getClass)

  def parse(row: Map[String, String]): Option[Movie] = ??? ③
}

```

① A genre has an id and a name

② Fields of a movie you'll consider

③ The parse function to implement

The `scala-csv` library will convert each line of the `movies_metadata.csv` file into a dictionary-like structure called `Map`, in which it associates each movie feature with its corresponding string value. You'll now implement the function `Movie.parse`: it converts the `row` parameter into a `Movie` instance. It will rely on the `parse` functions already implemented in `Parsers`. Their signatures below:

```

def parseInt(row: Map[String, String], key: String): Option[Int]
def parseDouble(row: Map[String, String], key: String): Option[Double]
def parseString(row: Map[String, String], key: String): Option[String]
def parseFloat(row: Map[String, String], key: String): Option[Float]
def parseLocalDate(row: Map[String, String], key: String): Option[LocalDate]
def parseGenres(row: Map[String, String], key: String): Option[List[Genre]]

```

These functions will try to find the value associated with a given key/feature and create an instance for their specific type. If a value exists and is compatible with the expected type, it will produce an instance of that type wrapped into a `Some`. For all the other cases, it will return `None`. A few example usages are the following:

```

parseInt(Map("id" -> "1"), "id")
// returns Some(1)

parseInt(Map("id" -> "1"), "myId")
// returns None because a value for key myId does not exist

parseInt(Map("id" -> "test"), "id")
// returns None because the string "test" cannot be converted to an Int

```

Listing 33.2 shows you how to combine the existing parse function to parse a Movie instance:

Listing 33.2: The Movie.parse function

```

def parse(row: Map[String, String]): Option[Movie] = {

  val movie = for {
    /* MANDATORY FIELDS */ ①
    genres <- parseGenres(row, "genres")
    id <- parseInt(row, "id")
    imdbId <- parseString(row, "imdb_id")
    originalLanguage <- parseString(row, "original_language")
    originalTitle <- parseString(row, "original_title")
    overview <- parseString(row, "overview")
    budget <- parseInt(row, "budget")
  } yield {

    /* NULLABLE FIELDS */ ②
    val popularity = parseFloat(row, "popularity")
    val releaseDate = parseLocalDate(row, "release_date")
    val runtimeInMinutes = parseDouble(row, "runtime")

    /* NULLABLE FIELDS WITH DEFAULTS */ ③
    val revenue = parseInt(row, "revenue").getOrElse[Int](0)
    val title = parseString(row, "title").getOrElse(originalTitle)
    val voteAverage = parseFloat(row, "vote_average").getOrElse[Float](0)
    val voteCount = parseFloat(row, "vote_count").getOrElse[Float](0)

    Movie(genres,
          id,
          imdbId,
          originalLanguage,
          originalTitle,
          title,
          overview,
          popularity,
          releaseDate,
          revenue,
          budget,
          runtimeInMinutes,
          voteAverage,
          voteCount)
  }

  if (movie.isEmpty) logger.warn(s"Skipping malformed movie row") ④
  movie
}

```

① Mandatory fields requested to create an instance of Movie

- ② Features of a movie that may be missing
- ③ Optional properties of a film that have a reasonable default value.
- ④ Warning to making visible when an entire row cannot be parsed

Although the dataset documents that every movie has a unique identifier under the label id, this is missing for three films in the CSV file. Rather than failing at runtime, your program gracefully handles it by logging a warning and skipping the row.

33.3 Printing Query Results

When querying the dataset, you'll print human-readable messages containing the question you asked and its answer. Let's ensure that this is done consistently in your program by creating a few helper functions. Add the following file to the `org.example.movie` package:

Listing 33.3: The print helper functions

```
// file src/main/scala/org/example/movies/PrintResultHelpers.scala
package org.example.movies

object PrintResultHelpers {

    def printResult(question: String, answer: String): Unit =
        printResult(question, answers = List(answer))

    def printResult(question: String, answer: Option[String]): Unit =
        printResult(question: String, answers = answer.toList)

    def printResult(question: String, answers: List[String]): Unit = {
        println()
        println("====")
        println(s"$question")

        if (answers.isEmpty) println("NOT FOUND") ①
        else println(answers.map(a => s"- $a").mkString("\n")) ②
    }
}
```

① It prints a message when no answers are available.

② It prints one or more answers.

After implementing these helper functions, you can now define the main class to query the dataset.

33.4 Querying the Movie Dataset

The components of your program are now ready for you to analyze the dataset. Let's define the main application that loads the dataset and parse it to produce an instance of `List[Movie]`:

Listing 33.5: The Movie Application

```
// file src/main/scala/org/example/movies/MovieApp.scala
package org.example.movies

import PrintResultHelpers._ ①

object MovieApp extends App {
```

```

val dataset = new MoviesDataset("movies_metadata.csv") ②
val movies = dataset.movies

private val unknown = "--" ③

// add your queries here!

}

```

- ① This is needed to print questions and answers consistently
- ② It loads and parses the CSV file to produce an instance of List[Movie]
- ③ You'll use this for empty nullable values, rather than displaying the text "None".

You can now execute the command `sbt run` to run your main application to load and parse the data. In the terminal, you should see an output similar to the following:

```

$ sbt run
[info] running org.example.movies.MovieApp
21:11:10.645 [run-main-0] INFO org.example.movies.MoviesDataset - Processing file
  movies_metadata.csv...
21:11:14.120 [run-main-0] INFO org.example.movies.MoviesDataset - Completed processing of file
  movies_metadata.csv! 45466 records loaded
21:11:15.550 [run-main-0] WARN org.example.movies.entities.Movie$ - Skipping malformed movie row
21:11:15.764 [run-main-0] WARN org.example.movies.entities.Movie$ - Skipping malformed movie row
21:11:15.905 [run-main-0] WARN org.example.movies.entities.Movie$ - Skipping malformed movie row
[success] Total time: 21 s, completed 20-Jan-2020 21:11:16

```

Notice three expected warnings in the logs: the three movies without a value for id are causing them. Your program can load and parse the movie information from the dataset, and it is ready for you to query.

HOW MANY MOVIES ARE THERE IN THE DATASET?

You can use the function `size` to count the movies in the dataset:

Listing 33.6: How many movies are there in the dataset?

```

printResult(
  question = "How many movies are there in the dataset?",
  answer = {
    val totCount = movies.size
    s"$totCount movies"
  }
)

```

You'll see that there are 45463 films in the dataset when executing the `sbt run` command:

```

=====
How many movies are there in the dataset?
- 45463 movies

```

This result is consistent with the 45466 records in the CSV file and the three skipped movie rows.

HOW MANY OF THEM WERE RELEASED IN 1987?

Thanks to the count method, you can compute the numbers of movies in the dataset that have release date with the year 1987:

Listing 33.7: How many movies were released in 1987?

```
printResult(
    question = "How many movies were released in 1987?",
    answer = {
      val countFrom1987 = movies.count(
        _.releaseDate.exists(_.getYear == 1987)) ①
      s"$countFrom1987 movies"
    }
)
```

① The field releaseDate is optional

Executing your main class will produce the following output:

```
=====
How many movies were released in 1987?
- 462 movies
```

The message shows that 462 movies were released in 1987. How would you count how many movies with no release date?

TOP FIVE MOVIES PER VOTE AVERAGE AND COUNT

Let's find the five films with the highest vote average and count. However, you want to penalize those with a low vote count: you should discard all movies with less than 50 votes.

Listing 33.8: Top 5 movies per vote average and count

```
printResult(
    question = "TOP 5 movies per vote average and count",
    answers = {
      val topPerVote = movies.filter(_.voteCount >= 50).sortBy(movie =>
        (- movie.voteAverage, - movie.voteCount))
      .take(5)
      topPerVote.map { movie =>
        s"[AVG: ${movie.voteAverage}, COUNT: ${movie.voteCount}]
          ${movie.title}" }
    }
)
```

Executing the command `sbt run`, will produce the following message:

```
=====
TOP 5 movies per vote average and count
- [AVG: 9.5, COUNT: 50.0] Planet Earth II
- [AVG: 9.1, COUNT: 661.0] Dilwale Dulhania Le Jayenge
- [AVG: 8.8, COUNT: 176.0] Planet Earth
- [AVG: 8.7, COUNT: 68.0] Sansho the Bailiff
- [AVG: 8.6, COUNT: 98.0] Human
```

Notice how the movie with the highest score, titled Planet Earth II, barely survived the vote count selection: it has 50. Simply discarding movies with an arbitrary low count may not be good enough: can you think of a better strategy?

FIND THE TOP FIVE MOVIES PER POPULARITY

Let's now compute a movie ranking per popularity and select the top five:

Listing 33.9: TOP 5 movies per popularity

```
printResult(
    question = "TOP 5 movies per popularity",
    answers = {
        val topPerPopularity = movies.sortBy(movie =>
            - movie.popularity.getOrElse(0f) ①
        ).take(5)
        topPerPopularity.map { movie =>
            s"[POPULARITY: ${movie.popularity.getOrElse(unknown)}] ②
                ${movie.title}"
        }
    }
)
```

① popularity is an optional field, so you should provide a reasonable default

You can produce the following by running your executable object:

```
=====
TOP 5 movies per popularity
- [POPULARITY: 547.4883] Minions
- [POPULARITY: 294.33704] Wonder Woman
- [POPULARITY: 287.25366] Beauty and the Beast
- [POPULARITY: 228.03275] Baby Driver
- [POPULARITY: 213.84991] Big Hero 6
```

The result shows a clear winner: the film Minions has a much higher popularity score than any other movie in the dataset.

SELECT FIVE NON-ENGLISH FILMS

So far, all the movies that you are selected are movies in English. Are there any movies that are not in English?

Listing 33.10: Select five non-English films

```
printResult(
    question = "5 non-english movies",
    answers = {
        val topNonEnglishPerPopularity = movies.filterNot(
            _.originalLanguage == "en"
        ).take(5)
        topNonEnglishPerPopularity.map { movie =>
            s"[LANG: ${movie.originalLanguage},
                RELEASE DATE: ${movie.releaseDate.getOrElse(unknown)}]
                    ③
                    ${movie.title} (${movie.originalTitle})"
        }
    }
)
```

)

① The field releaseDate is optional

Executing the command `sbt run` reveals the following output:

```
=====
5 non-english movies
- [LANG: fr, RELEASE DATE: 1995-05-16] The City of Lost Children (La Cité des Enfants Perdus)
- [LANG: zh, RELEASE DATE: 1995-04-30] Shanghai Triad (摇啊摇, 摆到外婆桥)
- [LANG: fr, RELEASE DATE: 1996-09-18] Wings of Courage (Guillaumet, les ailes du courage)
- [LANG: it, RELEASE DATE: 1994-01-01] Lamerica (Lamerica)
- [LANG: it, RELEASE DATE: 1994-09-22] The Postman (Il postino)
```

The list shows movies from French, Italian, and Chinese productions. Could you count how many Italian movies it contains? What about German films?

WHICH FILM MADE THE MOST PROFIT?

Let's now find which movie the most profit. Although profit is not a property in our movie dataset, you can derive it from its revenue and budget.

Listing 33.11: Which film made the most profit?

```
printResult(
    question = "Which movie made the most profit?",
    answer = {
        val mostProfit = movies.maxBy(movie => movie.revenue - movie.budget)
        val formattedProfit = {
            val formatter = java.text.NumberFormat.getInstance() ①
            formatter.format(mostProfit.revenue - mostProfit.budget)
        }
        s"[PROFIT: USD $formattedProfit] ${mostProfit.title}"
    }
)
```

① Using a formatter to make monetary amounts easier to read

When running your executable object `MovieApp`, you will see the following result:

```
=====
Which movie made the most profit?
- [PROFIT: USD 1,823,223,624] Star Wars: The Force Awakens
```

The Disney production *The Force Awakens* is the most profitable movie with a return of 1.8 billion USD.

Try a few more queries and see if you can find any unexpected answers. For example, can you list all the genres available in the dataset? Which movie is the most recent one? What is the duration of the longest film? Which of them made the smallest profit?

33.5 Summary

In this capstone, you have queried a subset of the movie dataset.

- You have created a parser for the movie entity by chaining optional values.
- You have created helper functions to consistently display your query results by creating lists

and producing a custom string representation for them.

- You have selected elements based on their features and positions.
- You have filtered and sorted items in a sequence according to different criteria.
- You have also determined minimum and maximum elements based on given features.

Unit 6

Other Collections and Error Handling

In Unit 5, you have learned about the structure `List` and its most common operations. In this unit, you'll discover other collections that Scala has to offer, and you'll learn how to handle errors in a more functional style. In the capstone, you'll parse data from the "goodbooks-10k" dataset to define a book collection for your library and manage book loans. In particular, we will discuss the following subjects:

- Lesson 34 introduces you to the class `Set` to represent an unordered group of distinct values. You'll also learn how to traverse its elements and manipulate them using its `map`, `flatten`, and `flatMap` operations.
- Lesson 35 shows you how to perform the union, difference, intersection operations on two sets. You'll see how inquiry about the property of an instance of `Set`, select one or more of its elements according to given criteria.
- Lesson 36 teaches you about a key-value structure called `Map`. You'll add and remove entries from it, as well as manipulate its content using its `map`, `flatten`, and `flatMap` functions.
- Lesson 37 gives you an overview of the operations that you can perform on `Map`. You'll inquire about its size and the properties of the entries it contains, as well as filter them based on their features.
- Lesson 38 introduces you to the class `Either` to represent a value that can have one of two possible types, an approach that is particularly useful when performing validation tasks. You'll also learn how to manipulate its content using its `map`, `flatten`, and `flatMap` operations.
- Lesson 39 teaches you how to handle `Either` instances by checking their kind and extracting their value.
- Lesson 40 shows you how to use the class `Try` to handle errors without throwing exceptions. You'll use pattern matching on it, as well as retrieve its value.

- Finally, you'll parse book information to define a book collection and use it to represent a library in which users can search, reserve, and return books in lesson 41.

After learning more about collections and handling errors without relying on exceptions, you'll continue with Unit 7, in which you'll master how to handle asynchronous computations.

34

Set

After reading this lesson, you will be able to:

- Define an unordered group of distinct values, called `Set`.
- Add and remove items from it.
- Manipulate its elements using the `map`, `flatten`, and `flatMap` methods.
- Chain multiple set instances using for-comprehension.

In the previous unit, you have learned about `List` and the operations you can perform on it. In this lesson, you'll discover the collection `Set` as an immutable representation of a group of elements. Sets and lists have many features in common and similar syntax, with a fundamental difference: the items of a set are unique and have no order. You'll see how to create it, add and remove elements from it. You'll discover how to manipulate its items using the `map`, `flatten`, and `flatMap` operations. Finally, you'll chain multiple instances of `Set` using for-comprehension. In the capstone, you'll use sets to store the book loans of a library.

Consider this

Suppose the list of your favorite movies, and you'd like to convert it to the list of your favorite genres. You should consider each genre once: how would you ensure that there are no duplicates?

34.1 Creating a Set

Suppose you are writing a program to track which topics a student has selected: each of them must be unique. Listing 34.1 shows you how to code this using `Set`:

Listing 34.1: The Student and Exam classes

```
case class Student(id: Int, name: String, topics: Set[String]) ①

val jon = Student(
  id = 1,
  name = "Jon Snow",
  topics = Set("History", "Math") ②
)
```

① The field `topics` has type `Set[String]`

② Initializing a set containing two elements

In Scala, you should use the collection `Set` to represent a group of unique elements that have no specific order. A few examples of how to create a set in Scala are the following:

```
scala> Set(1, 2, 3)
res0: scala.collection.immutable.Set[Int] = Set(1, 2, 3)
// a set containing the numbers 1, 2, and 3

scala> Set("hello", "hi", "hello")
res1: scala.collection.immutable.Set[String] = Set(hello, hi)
// A set contains no duplicates, so it contains the word "hello" only once

scala> Set(1, "scala")
res2: scala.collection.immutable.Set[Any] = Set(1, scala)

scala> Set()
res3: scala.collection.immutable.Set[Nothing] = Set()
// An empty set for instances of Nothing

scala> Set.empty[Double]
res4: scala.collection.immutable.Set[Double] = Set()
// The empty function allows you to create an empty Set for a given type
```

The elements of a `Set` are unordered. You can iterate through a set, but the order of its items is not guaranteed. When typing the expression `Set(1, 2)`, the REPL produces a message showing its elements in the same order 1,2: this is a coincidence that is dependent on the underlying implementation, and it can change depending on the Scala version you are using.

QUICK CHECK 34.1

Define a set containing the numbers 3 and 12.34 using the Scala REPL. What is the type of your `Set` instance? Why?

34.2 Adding and removing elements

Imagine that you'd like to update the topics a student has selected by removing and adding elements to the set. You can achieve this as follows:

Listing 34.2: Adding and removing exam records

```
val updatedTopics = jon.topics + "Chemistry" - "Math" ①
jon.copy(topics = updatedTopics)
```

① Adding "Chemistry" and removing "Math".

In Scala, `Set` is an immutable collection: your program will create a new set every time you add or remove an element from it. You can add an element to a set using the `+` operator:

```
scala> Set() + 1
res0: scala.collection.immutable.Set[Int] = Set(1)
// adding 1 to the empty set

scala> Set(1) + 2
res1: scala.collection.immutable.Set[Int] = Set(1, 2)
// adding 2 to a set containing one element

scala> Set(1) + 1
res2: scala.collection.immutable.Set[Int] = Set(1)
// a set contains no duplicates
```

When removing an element from a set, use the `-` operator as the following:

```
scala> Set(1, 2) - 2
res3: scala.collection.immutable.Set[Int] = Set(1)
// remove the element 2 from the set

scala> Set(1, 2) - 3
res4: scala.collection.immutable.Set[Int] = Set(1, 2)
// removing an element not in the set returns the same set

scala> Set() - 2
res5: scala.collection.immutable.Set[Int] = Set()
// removing an element from the empty set, returns the empty set
```

Adding and removing elements from a `Set` are efficient operations that happen in constant time. This is not the case for `List`: prepending an element is performant, but appending is not because it requires traversing the entire sequence.

QUICK CHECK 34.2

Consider the following two expressions. What value do they return? Are they equivalent?

1. `Set(2) + 1`
2. `1 + Set(2)`

Use the Scala REPL to validate your hypothesis.

34.3 The map, flatten, and flatMap operations

When working with `Option` and `List`, you have learned that you can use the methods `map`, `flatten`, and `flatMap` to manipulate their elements. You can also use these operations with `Set`: you'll see how in the following sections.

34.3.1 The map function

Suppose that your program needs to extract the names of a group of unordered students. Listing 34.3 shows you a possible way of achieving this using the function `map`:

Listing 34.3: Extracting the names of a set of students

```
case class Student(id: Int, name: String, topics: Set[String]) ①
def getIds(students: Set[Student]): Set[String] =
  students.map(_.name) ②
```

① For each student, extract the field name

The `map` function for a `Set` behaves similarly to the one for `List` and `Option`. It allows you to iterate over each element of a set and apply a transformation to it. In particular, the `map` operation on `Set[A]` is a higher order function that takes a parameter `f` of type `A => B` to produce a value of type `Set[B]`. It behaves as follows:

- It returns the empty set if it contains no elements.
- If not empty, it traverses each value, and it applies the function `f` to each of them.

A few examples for the function `map` on an instance of `Set` are the following:

```
scala> Set(0, 2, 4).map(_ * 3)
res0: scala.collection.immutable.Set[Int] = Set(0, 6, 12)
// multiplying each element by three

scala> Set.empty[Int].map(_ * 3)
res1: scala.collection.immutable.Set[Int] = Set()
// The set is empty, no elements to multiply by three!
```

QUICK CHECK 34.3

Define a function called `allUpper` that takes a set of words, and it returns a new one in which each element is now uppercase.

```
def allUpper(words: Set[String]): Set[String] = ???
```

34.3.2 The flatten function

Imagine you want to extract all the topics that a group of students is following. You could use the `map` operation, but you would obtain a return type of `Set[Set[String]]`:

```
def getTopics(students: Set[Student]): Set[Set[String]] =
  students.map(_.topics)
```

This solution is not what you'd like, as this will contain a set for each student, rather than the group of all topics. You can simplify a nested structure thanks to the `flatten` function:

Listing 34.4: Extracting topics of a group of students using flatten

```
def getTopics(students: Set[Student]): Set[String] =
  students.map(_.topics).flatten
```

When you have a nested set structure, you can simplify it using the `flatten` method. Its behavior and usage match the one for `List` and `Option`. A few examples are the following:

```
scala> Set(Set(1), Set(2)).flatten
res0: scala.collection.immutable.Set[Int] = Set(1, 2)

scala> Set(Set(1), Set(1)).flatten
res1: scala.collection.immutable.Set[Int] = Set(1)
// duplicated element 1 is shown once

scala> Set(Set(), Set()).flatten
res2: scala.collection.immutable.Set[Nothing] = Set()
```

QUICK CHECK 34.4

Consider the snippet of code `Set(3).flatten`. What does it return? Why? Use the Scala REPL to validate your hypothesis.

34.3.3 The flatMap function

Let's consider the function you have implemented in Listing 34.4 to extract the topics that a group of students has selected. Listing 34.5 shows you how you can rewrite it use the `flatMap` function:

Listing 34.5.: Extracting the topics of a group of students using flatMap

```
def getTopics(students: Set[Student]): Set[String] =
  students.flatMap(_.topics)
```

The method `flatMap` combines the behavior of the `map` and `flatten` operations. The `flatMap` method on `Set[A]` is a higher order function that takes a parameter `f` of type `A => Set[B]` to produce an instance of `Set[B]`. It behaves as follows:

- It returns the empty set if it contains no elements.
- If not empty, it extracts each value, and it applies the parameter `f` to it. Then, it combines all the results in one `Set`.

A few examples of what you can achieve thanks to the `flatMap` function are the following:

```
scala> Set(1, 2, 3).flatMap(n => Set("a", "b").map(_ * n))
res0: scala.collection.immutable.Set[String] = Set(a, b, bbb, aa, bb, aaa)
// repeating the strings "a" and "b" a number of times
// equal to the corresponding set element.

scala> Set.empty[Int].flatMap(n => Set("a", "b").map(_ * n))
res1: scala.collection.immutable.Set[String] = Set()
// The set is empty, so it returns a group with no strings.
```

QUICK CHECK 34.5

Using the `flatMap` operation, define a function called `crossMultiplier` that takes two sets of integers as its parameters, and returns a new one containing all the numbers produced by multiplying each element of the first one for the second.

```
def crossMultiplier(groupA: Set[Int], groupB: Set[Int]): Set[Int] = ???
```

For example, when applying the `crossMultiplier` function to two sets containing the numbers 1,3 and 2,4,6 respectively, it should return a new one containing the numbers 2, 4, 6, 12, 18.

Table 34.1 provides a summary of the `map`, `flatten`, `flatMap` functions acting on a `Set`.

Table 34.1: Technical recap of the three fundamental operations on Set. The function map applies a given function to each element in the group, while the function flatten creates a set by unifying two nested structures. The flatMap operation combines the map and flatten operations to chain values together.

	Acts on	Signature	Usage
map	<code>Set[A]</code>	<code>map(f: A => B): Set[B]</code>	It applies a function to each value in the set.
flatten	<code>Set[Set[A]]</code>	<code>flatten: Set[A]</code>	It merges two nested sets into one.
flatMap	<code>Set[A]</code>	<code>flatMap(f: A => Set[B]): Set[B]</code>	The combination of map followed by flatten. It chains sets together.

34.4 For-comprehension

Let's consider your program to track students and their selected topics, and imagine that you now need to return the group of topics that a selected group of students are following. You could achieve this using the `flatMap` operation as follows:

Listing 34.6: Retrieving topics for student ids using flatMap

```
def getTopicsForStudentIds(students: Set[Student],
                           ids: Set[Int]): Set[String] =
  students.flatMap { student => ①
    ids.flatMap { id => ②
      if (student.id == id) student.topics ③
      else Set.empty ④
    }
  }
```

- ① Iterating through each student
- ② Looping through each id
- ③ If the student id match, return the student's topics
- ④ If the student id doesn't match, return no topics

A more elegant way of rewriting the same is by using for-comprehension:

Listing 34.7: Retrieving topics for student ids using for-comprehension

```
def getTopicsForStudentIds(students: Set[Student],
                           ids: Set[Int]): Set[String] =
  for {
    student <- students ①
    id <- ids ②
    if student.id == id ③
    topic <- student.topics ④
  } yield topic ⑤
```

- ① Iterating through each student
- ② Looping through each id
- ③ Filtering only students with a matching id
- ④ Extracting each topic belonging to the student
- ⑤ Inserting each topic into the set

The function `getTopicsForStudentIds` has `Set[String]` as its return type: you need to iterate through each topic a student has. Suppose you were to `yield` the value `student.topics` rather than a single topic: the for-comprehension construct would return an instance of `Set[Set[String]]` rather than the desired `Set[String]`.

Every time your code has nested calls to `flatMap` and `map` functions, you should consider rewriting it using for-comprehension to improve its readability.

For example, consider the following snippet of code you have seen in the previous subsection on `flatMap`:

```
scala> Set(1, 2, 3).flatMap(n => Set("a", "b").map(_ * n))
res0: scala.collection.immutable.Set[String] = Set(a, b, bbb, aa, bb, aaa)
```

You can refactor it using for-comprehension as follows:

```
scala> for {
  |   n <- Set(1, 2, 3)
  |   s <- Set("a", "b")
  | } yield s * n
res1: scala.collection.immutable.Set[String] = Set(a, b, bbb, aa, bb, aaa)
```

QUICK CHECK 34.6

Rewrite the function `crossMultiplier` you have implemented in quick check 34.5 using for-comprehension.

34.5 Summary

In this lesson, my objective was to teach you about the `Set` collection.

- You have learned how to create a set.
- You have seen how to add and remove elements from it using the `+` and `-` operators.
- You have discovered how to manipulate its values using the `map`, `flatten`, `flatMap` functions.
- Also, you have mastered how to chain multiple set instances using for-comprehension.

Let's see if you got this!

TRY THIS

Define a function that takes a set of books to return the set of genres that a given author has written. Use the following case class to represent a book:

```
case class Book(title: String, authors: List[String], genres: Set[String])
```

34.6 Answers to Quick Checks

QUICK CHECK 34.1

You can define a set as follows:

```
scala> Set(3, 12.4)
res0: scala.collection.immutable.Set[Double] = Set(3.0, 12.4)
```

Your instance has type `Set[Double]` because the compiler can unify both integers and doubles under the type `Double`.

QUICK CHECK 34.2

The two expressions are not equivalent. The first snippet of code creates a set containing the numbers 1 and 2. You can rewrite the expression `Set(2) + 1` as `Set(2).+(1)`: you are calling the method called `+` defined in the class `Set`. The second expression does not compile. The expression `1 + Set(2)` corresponds to `1.+(Set(2))`: you are calling the method `+` in the class `Int`. For this reason, the compiler thinks you are trying to add the set to the number one, which is an illegal operation.

QUICK CHECK 34.3

A possible implementation for the function `allUpper` is the following:

```
def allUpper(words: Set[String]): Set[String] =
  words.map(_.toUpperCase)
```

QUICK CHECK 34.4

The expression `Set(3).flatten` does not compile. You can use the `flatten` function only on nested structures: `Set(3)` is not.

```
scala> Set(3).flatten
<console>:12: error: No implicit view available from Int => IterableOnce[B].
          Set(3).flatten
                      ^
```

QUICK CHECK 34.5

You can implement the `crossMultiplier` function as follows:

```
def crossMultiplier(groupA: Set[Int], groupB: Set[Int]): Set[Int] =
  groupA.flatMap { a =>
    groupB.map(b => a * b)
  }
```

QUICK CHECK 34.6

You can rewrite the `crossMultiplier` function using for-comprehension as the following:

```
def crossMultiplier(groupA: Set[Int], groupB: Set[Int]): Set[Int] =  
  for {  
    a <- groupA  
    b <- groupB  
  } yield a * b
```

35

Working with Set

After reading this lesson, you will be able to:

- Perform union, intersection, and difference operations on sets.
- Inquire about the properties of a set
- Select one of its elements based on its features
- Filter its items according to their characteristics

In the previous lesson, you have learned about the structure of a set and how to transform its elements. In this lesson, you'll discover other methods that the class `Set` has to offer. You'll apply the operations of union, intersection, and difference on two sets. Then, I'll introduce you to more complex transformations you can do on them: you'll notice a lot of overlap with the concepts you have discovered for the `List` collection in unit 5. You'll inquire about the properties of a set, such as its size or the presence of an item. You'll select one of its elements based on its characteristics and determine its minimum and maximum. You'll filter its values based on their features. In the capstone, you'll use these operations to analyze the book loans of your library.

Consider this

Imagine you have a group of books, and you'd like to check if a given title is present. Your application performs this operation often, and it needs to be as efficient as possible. How would you implement it?

35.1 The Union, Intersection, Difference Operations

Let's consider the program to track the students and their selected topics you have seen in the previous lesson. Suppose you want to identify students based on their topic selection. Listing 35.1 shows you how to implement this:

Listing 35.1: Students selection based on their topics

```
case class Student(id: Int, name: String, topics: Set[String])

def eitherTopics(topicA: Set[Student],
                topicB: Set[Student]): Set[Student] =
  topicA.union(topicB) ①

def bothTopics(topicA: Set[Student],
              topicB: Set[Student]): Set[Student] =
  topicA.intersect(topicB) ②

def topicAноTopicB(topicA: Set[Student],
                   topicB: Set[Student]): Set[Student] =
  topicA.diff(topicB) ③

def topicBnoTopicA(topicA: Set[Student],
                   topicB: Set[Student]): Set[Student] =
  topicB.diff(topicA) ④
```

- ① The students taking topicA or topicB
- ② Those who have selected topicA and topicB
- ③ Those following topicA but not topicB
- ④ Students who have picked topicB but not topicA

In Scala, the class `Set` mimics the concept of a mathematical set. In this section, you'll learn how to apply the operations of union, intersection, and difference: have a look at figure 35.1 for the Venn diagram of their mathematical meaning.

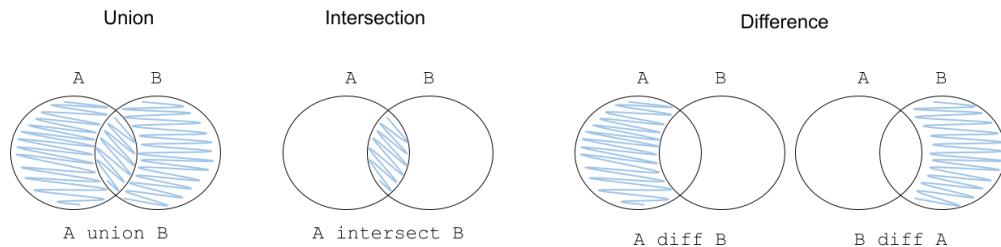


Figure 35.1: The fundamental operations on sets. The operation `union` merges them, while the `intersection` one selects only the elements they have in common. Finally, the `difference` includes the values in one but not the other.

When working with two or more sets, you can perform the following operations to combine their elements and analyze their commonalities and differences:

- `union` – For an instance of `Set[A]`, the method `union` takes another `Set[A]` as its parameter. It returns a new set that contains all the values from any of them. You can also use the operator `++` as an alias for the method `union`.

```
scala> Set(1, 2, 3).union(Set(1, 4))
res0: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)
// The following two expressions are also equivalent:
```

```
// Set(1, 2, 3) union Set(1, 4)
// Set(1, 2, 3) ++ Set(1, 4)

scala> Set(1, 2, 3).union(Set())
res1: scala.collection.immutable.Set[Int] = Set(1, 2, 3)
// Alternatively, you can also write ...
// Set(1, 2, 3) union Set()
// Set(1, 2, 3) ++ Set()
```

- **intersect** – For an instance of `Set[A]`, the method `intersect` takes another `Set[A]` as its parameter. It returns a new set with only the values that they both contain. It also has an alias: the method `&`.

```
scala> Set(1, 2, 3).intersect(Set(1, 4))
res0: scala.collection.immutable.Set[Int] = Set(1)
// You can also write:
// Set(1, 2, 3) intersect Set(1, 4)
// Set(1, 2, 3) & Set(1,4)

scala> Set(1, 2, 3).intersect(Set())
res1: scala.collection.immutable.Set[Int] = Set()
// Or alternatively:
// Set(1, 2, 3) intersect Set()
// Set(1, 2, 3) & Set()
```

- **diff** – For an instance of `Set[A]`, the function `diff` takes `Set[A]` as its parameter. It returns a new set containing all the elements that are in the original set but not in the given one. You can also use the operator `--` as its alias.

```
scala> Set(1, 2, 3).diff(Set(1, 4))
res0: scala.collection.immutable.Set[Int] = Set(2, 3)
// The following two expressions are also equivalent:
// Set(1, 2, 3) diff Set(1, 4)
// Set(1, 2, 3) -- Set(1, 4)

scala> Set(1, 4).diff(Set(1, 2, 3))
res1: scala.collection.immutable.Set[Int] = Set(4)
// Or you can write:
// Set(1, 4) diff Set(1, 2, 3)
// Set(1, 4) -- Set(1, 2, 3)
```

Have a look at table 35.1 for a summary of the union, intersect, and difference operations for two sets.

Table 35.1: Summary of the functions to perform basic operations on two instances of Set[A]. Where necessary, I have simplified their signatures to hide non-relevant implementation details.

Signature	Description	Example
<code>def union(other: Set[A]): Set[A]</code>	It unifies the elements of the two sets.	<code>Set(1,0,2) .union(Set(1, 3)) // it returns // Set(0,1,2,3)</code>
<code>def ++(other: Set[A]): Set[A]</code>	It behaves the same as union.	<code>Set(1,0,2)++ Set(1, 3) // it returns // Set(0,1,2,3)</code>
<code>def intersect(other: Set[A]): Set[A]</code>	It selects the values present in both groups.	<code>Set(1,0,2) .intersect(Set(1, 3)) // it returns // Set(1)</code>
<code>def &(other: Set[A]): Set[A]</code>	It aliases intersect.	<code>Set(1,0,2) & Set(1, 3) // it returns // Set(1)</code>
<code>def diff(other: Set[A]): Set[A]</code>	It returns a new set containing the items present in this set, but not in other.	<code>Set(1,0,2) .diff(Set(1, 3)) // it returns // Set(0,2)</code>
<code>def --(other: Set[A]): Set[A]</code>	It behaves the same as diff.	<code>Set(1,0,2) -- Set(1, 3) // it returns // Set(0,2)</code>

QUICK CHECK 35.1

Consider these two snippets of code: are they equivalent? Use the Scala REPL to validate your hypotheses.

1. `Set(1, 2, 3).diff(Set(3,4))`
2. `Set(3,4).diff(Set(1,2,3))`

35.2 Other Operations on Set

Suppose you need to analyze certain features of the group of students and their topics. In particular, you'd like to:

- Check if a student with a specific id is in the group
- Filter those that have selected a given class
- Find the one that is following the most topics

Listing 35.2 shows how you could implement them:

Listing 35.2: Analyzing a group of students

```
case class Student(id: Int, name: String, topics: Set[String])

def existsById(students: Set[Student], id: Int): Boolean =
  students.exists(_.id == id) ①

def filterByTopic(students: Set[Student], topic: String): Set[Student] =
  students.filter { student => ②
    student.topics.contains(topic) ③
  }

def maxByTopics(students: Set[Student]): Student =
  students.maxBy { student => ④
    student.topics.size ⑤
  }
```

- ① It checks if a value with the given characteristic exists
- ② It filters elements based on a specific feature
- ③ It checks if the set contains a value
- ④ It returns the maximum item according to a given criterion
- ⑤ It returns the size of a set

In Scala, both `List` and `Set` implement a common interface, called `Iterable`. Thanks to this design choice, they share a consistent group of methods for you to use. You can apply the operations that you have discovered so far for `List` on an instance of `Set`, with the following exceptions:

- You cannot sort a set because their elements are unordered by design: you cannot invoke the methods `sorted` or `sortedBy` on it.
- You cannot remove duplicates from it, as its items are already unique: it doesn't have a `distinct` function.

`List` and `Set` also differ in performance when checking if an element exists. The element lookup time in a list is proportional to its size, while it is constant for a set.

QUICK CHECK 35.2

Define a function called `sumInRange` that takes a set of doubles, and it returns the sum of all its values between 0 and 100 excluded. For example, given the numbers 0.5, -1, 0 50.5, 99, and 100, it should return the double 150.00.

```
def sumInRange(numbers: Set[Double]): Double
```

35.3 Summary

In this lesson, my objective was to teach you about the operations to perform on a set.

- You have seen how to merge, intersect and subtract two sets
- You have learned how to check the size of a set and the properties of its elements
- You have mastered how to select one of its elements based on a given criterion.
- Also, you have seen how to filter its values according to their features.

Let's see if you got this!

TRY THIS

Implement a function that takes a group of students and a set of topics as its parameters. It returns a new set containing the students that are taking any of the given topics. Use the student representation you have used in this lesson:

```
case class Student(id: Int, name: String, topics: Set[String])
```

35.4 Answers to Quick Checks

QUICK CHECK 35.1

The two expressions are not equivalent. The difference between sets is not commutative, which means that its parameters' order does change the result. While the first expression returns the instance `Set(1, 2)`, the other evaluates to `Set(4)`.

QUICK CHECK 35.2

A possible implementation for the function `sumInRange` is the following:

```
def sumInRange(numbers: Set[Double]): Double =  
  numbers.filter(d => d > 0 && d < 100).sum
```

36

Map

After reading this lesson, you will be able to:

- Define a key-value structure, called `Map`.
- Add and remove entries to it
- Compute the union and difference of two `Maps`
- Manipulate the elements of a key-value structure using the `map` and `flatMap` functions
- Chain multiple instances using for-comprehension

In the previous lesson, you have mastered the operations that you can perform on a `Set`. In this lesson, you'll discover a new data structure called `Map`. In Scala, a `Map` is an immutable data structure to store a set of keys, each of them associated with a value. The concept of mapping keys to values is not unique to the Scala language: some languages, such as Java, refer to it using the term *hashmap*; others, such as Python, call it *dictionary*. You'll create an instance of `Map`, add and remove elements to it. You'll merge and subtract the keys of two maps to create a new one. You'll manipulate and transform its entries using the `map` and `flatMap` functions. You'll also combine multiple values using for-comprehension. In the capstone, you'll use `Map` to read the data from a CSV file.

Consider this

Consider you have a list of books, and you'd like to group them per genre. Which data structure would you use to represent it?

36.1 Creating a Map

Consider the program to track the students and their selected topics you have developed in the previous lessons. Imagine you want to modify it so that you can record the students registered for an exam session. Listing 36.1 shows you how to represent this using a `Map`:

Listing 36.1: Tracking students registered to each exam session

```
import java.time.LocalDate

// Representing our data...
case class Student(id: Int, name: String)
case class ExamSession(title: String, localDate: LocalDate)

// Instances of ExamSession
val historySession = ExamSession(
  "History", localDate = LocalDate.now.plusDays(30))
val chemistrySession = ExamSession(
  "Chemistry", localDate = LocalDate.now.plusDays(45))

// Instances of Student
val jon = Student(id = 1, name = "Jon Snow")
val daenerys = Student(id = 2, name = "Daenerys Targaryen")
val arya = Student(id = 3, name = "Arya Stark")

val registrations: Map[ExamSession, List[Student]] = ①
  Map(
    historySession -> List(jon, daenerys), ②
    chemistrySession -> List(jon, arya)
  )
```

① It creates a map with key `ExamSession` and `List[Student]` as its value

② It creates a tuple of type `(ExamSession, List[Student])`

A `Map` is a key-value data structure. It has a set of keys that are unordered, unique, and have values linked to each of them. Scala represents each key-value association, called entry, with a tuple. Its keys and values have specific types: each instance of a `Map` has type `Map[K, V]`, in which `K` is the type of its keys, and `V` the type of their values. Have a look at figure 36.1 for a visual summary of the structure of a `Map`.

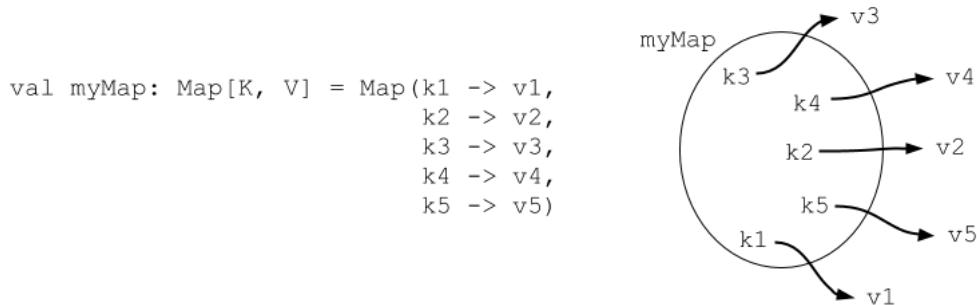


Figure 36.1: Visual representation of the key-value structure of `Map`. Its keys are unordered, unique, and of a given type `K`. Each of them has an associated value of a given type `V`. You can create a key-value structure `Map[K, V]` by invoking its constructor with zero or more tuples of the type `(K, V)`.

When creating a `Map` of type `Map[K, V]`, you represent each of its entries using tuples of type `(K, V)` and pass them to its constructor:

```

scala> Map((1,"hi"), (2,"scala"))
res0: scala.collection.immutable.Map[Int,String] = Map(1 -> hi, 2 -> scala)
// A Map containing two entries

scala> Map((42.31, 42))
res1: scala.collection.immutable.Map[Double,Int] = Map(42.31 -> 42)
// A Map containing one entry of type (Double, Int)

scala> Map()
res2: scala.collection.immutable.Map[Nothing,Nothing] = Map()
// The empty Maap

scala> Map.empty[String, Double]
res3: scala.collection.immutable.Map[String,Double] = Map()
// The empty Map with key of type String, and values of type Double

```

Scala offers an alternative syntax to create tuples. When creating a tuple, you can also use the `->` operator. The following expressions are equivalent:

```

scala> ("hello", "world")
res0: (String, String) = (hello,world)
// creating a tuple

scala> "hello" -> "world"
res1: (String, String) = (hello,world)
// Equivalent creation of tuple using the -> operator

```

This can improve your code's readability, especially in certain contexts such as the creation of instances of `Map`:

```

scala> Map(1 -> "hi", 2 -> "scala")
res0: scala.collection.immutable.Map[Int,String] = Map(1 -> hi, 2 -> scala)
// Equivalent to Map((1,"hi"), (2,"scala"))

```

```
scala> Map(42.31 -> 42)
res1: scala.collection.immutable.Map[Double,Int] = Map(42.31 -> 42)
// An alternative to Map((42.31, 42))
```

QUICK CHECK 36.1

Create a key-value structure of type `Map[Int, String]` to associate a number to its corresponding day of the week. For example, you should associate the number 1 to Monday, while 2 to Tuesday.

36.2 Adding and Removing elements

Imagine you need to add a new exam session for Math and remove the one for Chemistry in your exam tracking program. Listing 36.2 show you how to achieve this:

Listing 36.2: Adding and Removing Exam Sessions

```
val mathSession = ExamSession("Math", localDate = LocalDate.now)
registrations + (mathSession -> List(daenerys)) ①
- chemistrySession ②
```

① Adding the entry for Math and the student Daenerys

② Removing the entry associated with Chemistry

A `Map` is an immutable structure: adding or removing elements to it creates a new instance rather than modifying the existing one. To create a new instance with an added entry, you can use the invoke the method `+` with the entry to add. For example, you can do the following:

```
scala> Map(1 -> "hello") + (2 -> "scala")
res0: scala.collection.immutable.Map[Int,String] = Map(1 -> hello, 2 -> scala)

scala> Map() + (1 -> "scala")
res1: scala.collection.immutable.Map[Int,String] = Map(1 -> scala)
```

A `Map` has unique keys by keys. For this reason, if you add an entry for an existing key, its corresponding value will also be replaced:

```
scala> Map(1 -> "hello") + (1 -> "scala")
res2: scala.collection.immutable.Map[Int,String] = Map(1 -> scala)
// The value "hello" is no longer in the map.
```

When removing the entry, you can use the method `-` with the key of the entry to remove:

```
scala> Map(1 -> "hello", 2 -> "scala") - 1
res0: scala.collection.immutable.Map[Int,String] = Map(2 -> scala)
// Removing entry with key 1

scala> Map(1 -> "hello", 2 -> "scala") - 3
res1: scala.collection.immutable.Map[Int,String] =
Map(1 -> hello, 2 -> scala)
// Removing a non-existing key returns the original Map.
```

The operation of adding and removing elements are fast because they happen in constant time independently from the size of your dictionary.

QUICK CHECK 36.2

Consider the following snippet of code. What does it return? Why? Use the Scala REPL to validate your hypothesis.

```
Map(42 -> "hi") + 3
```

36.3 Merge and remove multiple entries

Let's consider your program to track the exam sessions and the students registered to them. Imagine you have two separate key-value structures to represent two different exam registrations in the year, and you'd like to merge them. Listing 36.3 show you how:

Listing 36.3: Merging two exams registrations

```
def merge(regA: Map[ExamSession, List[Student]],
          regB: Map[ExamSession, List[Student]]): Map[ExamSession, List[Student]] =
  regA ++ regB ①
```

① Merging the two registrations together

In Scala, you can consider a Map as a set of keys, each linked to a value: you can perform operations on them similar to the ones you have learned for `Set`. You can merge the entries of two maps using the operator `++`. For the keys in common, the entries of the second map override those of the first one. A few examples are the following:

```
scala> Map(1 -> "hello") ++ Map(2 -> "scala")
res0: scala.collection.immutable.Map[Int,String] =
Map(1 -> hello, 2 -> scala)
// Merging two key-value structures

scala> Map(1 -> "hello") ++ Map()
res1: scala.collection.immutable.Map[Int,String] = Map(1 -> hello)
// Merging a Map with the empty one

scala> Map(1 -> "hello") ++ Map(1 -> "scala")
res2: scala.collection.immutable.Map[Int,String] = Map(1 -> scala)
// The entry (1, "scala") overrides (1, "hello")
```

You can also use the method `--` to remove multiple entries by providing their keys. Its parameter has type `Iterable`: it indicates you can use any list-like structure, such as `Set` or `List`.

```
scala> Map("Rome" -> "Italy", "London" -> "UK") -- Set("Rome", "Paris")
res01: scala.collection.immutable.Map[String,String] = Map(London -> UK)
// Removing the keys "Rome" and "Paris" using Set as Iterable

scala> Map("Rome" -> "Italy", "London" -> "UK") -- List("Berlin")
res15: scala.collection.immutable.Map[String,String] =
Map(Rome -> Italy, London -> UK)
// Removing a non-existing key, returns the original data structure
// using List as Iterable.
```

QUICK CHECK 36.3

In Quick Check 36.1, you have defined a key-value data structure of type `Map[Int, String]` to represent the numbers and their corresponding days of the week. Use the operator `--` to create a new map containing only weekdays (i.e., all but Saturday and Sunday).

36.4 The map and flatMap operations

A `Map` is a key-value data structure that you can see as an iterable of tuples. As you have discovered previously, `Iterable` is a shared interface between `Set` and `List`. For this reason, the `map` and `flatMap` operations on `Map` behave similarly to the iterable collections you have seen so far. Let's see them in action in the following subsections.

36.4.1 The map function

Let's consider your program to track the exam sessions and the students registered to it and imagine you'd like to create a key-value data structure that links each exam session to the number of students enrolled for it. Listing 36.4 shows you how you could implement this:

Listing 36.4: Number of registrations for each exam session

```
def registrationsPerSession(
    registrations: Map[ExamSession, List[Student]]): Map[ExamSession, Int] =
  registrations.map { ❶
    case (examSession, students) => examSession -> students.size ❷
  }
```

❶ Iterating through each entry in the `Map`.

❷ Using a partial function to decompose the elements of each tuple

When applying the function `map` over an instance of `Map`, you are iterating through each tuple. For example, given a capital to country mapping, you can swap it so that it becomes a country to capital one:

```
scala> Map("Rome" -> "Italy", "London" -> "UK").map {
    |   tuple => tuple._2 -> tuple._1
    | }
res0: scala.collection.immutable.Map[String,String] =
Map(Italy -> Rome, UK -> London)
```

Although the methods `._1` and `._2` on a tuple work as expected, you often want to improve your code's readability by providing a more descriptive name to each element of the tuple. You can achieve this by using a partial function on each tuple as follows:

```
scala> Map("Rome" -> "Italy", "London" -> "UK").map {
    |   case (capital, country) => country -> capital
    | }
res1: scala.collection.immutable.Map[String,String] =
Map(Italy -> Rome, UK -> London)
```

Scala 3 compiler is more capable than the Scala 2 one when inferring types. You do not need a partial function to decompose each entry tuple in Scala 3 because the compiler can do this for you automatically. You can omit the `case` keyword by writing the following if you are using Scala 3:

```
scala> Map("Rome" -> "Italy", "London" -> "UK").map {
|   (capital, country) => country -> capital
| }
val res2: Map[String, String] = Map(Italy -> Rome, UK -> London)
```

You can also transform each entry into a type rather than a tuple: in such case, the compiler will not build a key-value structure and return an iterable of that type:

```
scala> Map("Rome" -> "Italy", "London" -> "UK").map {
|   case (capital, country) => s"$capital is the capital of $country"
| }
res3: scala.collection.immutable.Iterable[String] =
List(Rome is the capital of Italy, London is the capital of UK)
```

The Map collection and the flatten function

You cannot flatten a key-value data structure. In Scala, the compiler can consider an instance of type `Map[K, V]` equivalent to `Iterable[(K, V)]`: this structure is not nested because a tuple is not an iterable. If you try to invoke the method `flatten` on `Map`, the compiler will complain that it doesn't know how to transform your tuple into an `Iterable`:

```
scala> Map("hello" -> "world").flatten
^
error: No implicit view available from (String, String) => scala.collection.IterableOnce[B].
```

QUICK CHECK 36.4

Consider the following snippet of code. What does it return? Why? Use the Scala REPL to validate your hypothesis.

```
Map("hello" -> 1, "scala" -> 1).map { case (w, n) => n -> w }
```

36.4.2 The flatMap function

Imagine that in your program to track exam registrations, you'd like to return data specific to a given group of student ids. You can implement this using the `flatMap` function as follows:

Listing 36.5: Filtering exam registrations by student ids using flatMap

```
def filterByStudentId(registrations: Map[ExamSession, List[Student]],
  ids: List[Int]): Map[ExamSession, List[Student]] =
  registrations.flatMap { case (examSession, students) => ①
    val matches = students.filter(student => ids.contains(student.id))
    if (matches.nonEmpty) List(examSession -> matches)
    else List.empty
  } ②
```

① You can omit the keyword `case` in Scala 3

② `Map` and `List` are both implementations of `Iterable`, so you can mix them when using `flatMap`

When using a `flatMap` on a `Map`, the compiler transforms it into an iterable of tuples. It will return a value of type `Map` if the structure of the returned value allows it to do so. For example, you can filter the entries containing positive numbers:

```
scala> Map(1 -> 2, 0 -> 2, 2 -> 0).flatMap { case (a, b) =>
|   if (a > 0 && b > 0) Some(a -> b) else None
| }
res0: scala.collection.immutable.Map[Int,Int] = Map(1 -> 2)
// Option is automatically converted to List,
// which is an implementation of Iterable
```

Also, you can multiply the two numbers in the tuple and return an instance of type `Iterable[Int]`:

```
scala> Map(1 -> 2, 0 -> 2, 2 -> 0).flatMap { case (a, b) =>
|   if (a > 0 && b > 0) Some(a * b) else None
| }
res1: scala.collection.immutable.Iterable[Int] = List(2)
```

QUICK CHECK 36.5

Consider the following snippet of code. What does it return? Why? Use the Scala REPL to validate your hypothesis.

```
Map("hello" -> 1, "scala" -> 10).flatMap { case (w, n) =>
if (w.length > n) Some(w -> n) else None
}
```

36.5 For-comprehension

Let's consider your implementation to filter your exam registrations based on given student ids in listing 36.6. You can refactor it using for-comprehension as follows:

Listing 36.6: Filter exam registration by student ids using for-comprehension

```
def filterByStudentId(registrations: Map[ExamSession, List[Student]],
ids: List[Int]): Map[ExamSession, List[Student]] =
  for {
    (examSession, students) <- registrations
    matches = students.filter(student => ids.contains(student.id)) ①
    if matches.nonEmpty
  } yield examSession -> matches
```

① You can omit val here

You can refactor any expressions using `flatMap` on `Map` with a for-comprehension construct. For example, you can rewrite the examples you have seen for it as the following:

```
scala> for {
|   (a, b) <- Map(1 -> 2, 0 -> 2, 2 -> 0)
|   if a > 0 && b > 0
| } yield a -> b
res0: scala.collection.immutable.Map[Int,Int] = Map(1 -> 2)

scala> for {
|   (a, b) <- Map(1 -> 2, 0 -> 2, 2 -> 0)
|   if a > 0 && b > 0
| }
```

```
| } yield a * b
res1: scala.collection.immutable.Iterable[Int] = List(2)
```

QUICK CHECK 36.6

Re-implement the following snippet of code using for-comprehension:

```
Map("hello" -> 1, "scala" -> 10).flatMap { case (w, n) =>
if (w.length > n) Some(w -> n) else None
}
```

36.6 Summary

In this lesson, my objective was to teach you about the `Map` collection.

- You have mastered how to create a `Map` using both the tuple constructor and its `->` operator.
- You have seen how to add and remove single and multiple entries from a key-value structure.
- You have learned how to transform its entries and combine its elements using the `map` and `flatMap` operations.

Let's see if you got this!

TRY THIS

Suppose you have two key-value structures representing a capital to its country, and a country to its continent. Combine the two instances of `Map` to link each capital to its continent.

36.7 Answers to Quick Checks

QUICK CHECK 36.1

You can define your instance of `Map` as the following:

```
Map(
  1 -> "Monday",
  2 -> "Tuesday",
  3 -> "Wednesday",
  4 -> "Thursday",
  5 -> "Friday",
  6 -> "Saturday",
  7 -> "Sunday"
)
```

Alternatively, you can also use the standard constructor for tuples:

```
Map(
  (1, "Monday"),
  (2, "Tuesday"),
  (3, "Wednesday"),
  (4, "Thursday"),
  (5, "Friday"),
  (6, "Saturday"),
  (7, "Sunday")
)
```

QUICK CHECK 36.2

The snippet of code doesn't compile because an integer is not a valid representation for the entry of a dictionary:

```
scala> Map(42 -> "hi") + 3
scala> Map(42 -> "hi") + 3
1 |Map(42 -> "hi") + 3
   |
   |          ^
   |          Found:    (3 : Int)
   |          Required: (Any, Any)
```

QUICK CHECK 36.3

You can create a new map representing the weekdays as the following:

```
val days = Map(
  1 -> "Monday",
  2 -> "Tuesday",
  3 -> "Wednesday",
  4 -> "Thursday",
  5 -> "Friday",
  6 -> "Saturday",
  7 -> "Sunday"
)

val weekdays = days -- Set(6,7)
```

Alternatively, you can use `List` rather than `Set` to indicate the group of keys to remove:

```
val weekdays = days -- List(6,7)
```

QUICK CHECK 36.4

The expression returns a key-value data structure containing one entry rather than two. Your output may change slightly depending on the Scala version you are using:

```
scala> Map("hello" -> 1, "scala" -> 1).map { case (w, n) => n -> w }
res0: scala.collection.immutable.Map[Int, String] = Map(1 -> scala)
```

The snippet of code creates a new `Map` by swapping keys with values. Because its keys must be unique, the compiler overrides new entries with the same key without warnings. You should be careful when transforming a `Map`'s data, or you may discard it without realizing it!

QUICK CHECK 36.5

The snippet of code evaluates to a key-value data structure containing the entry ("hello", 1):

```
scala> Map("hello" -> 1, "scala" -> 10).flatMap { case (w, n) =>
  |   if (w.length > n) Some(w -> n) else None
  | }
res0: scala.collection.immutable.Map[String, Int] = Map(hello -> 1)
```

The compiler can return its resulting value of type `Iterable[(String, Int)]` as an instance of `Map[String, Int]`.

QUICK CHECK 36.6

You can refactor the snippet of code using for-comprehension as follows:

```
for {
  (w, n) <- Map("hello" -> 1, "scala" -> 10)
  if w.length > n
} yield w -> n
```

37

Working with Map

After reading this lesson, you will be able to:

- Retrieve the value associated with a key
- Get all keys and all values stored in a Map
- Check its size and the feature of its entries
- Filter its elements based on given criteria

In the previous lesson, you have learned about the basics of Map. In this lesson, you'll master the most common operations you can apply to it. You'll use different strategies to retrieve a value linked to a given key. You'll get the keys and the values in an instance of a Map. Finally, you will get its size, inquire about its entries' properties, and filter its elements according to their characteristics by reusing the same methods you have seen for List and Set. In the capstone, you will apply these operations to extract the book information from a CSV file and create your library's book collection.

Consider this

Imagine that you have a key-value structure representing a sequence of books per genre, and you'd like to retrieve books with genre drama. How would you implement it?

37.1 Retrieving a value for a given key

Consider the program to track the exam registration that you are implementing. Imagine you now need to retrieve the students that have enrolled for a given exam session. Listing 37.1 shows you how to do that:

Listing 37.1: Getting students for an exam session

```
import java.time.LocalDate

case class Student(id: Int, name: String)
case class ExamSession(title: String, localDate: LocalDate)

def getStudents(registrations: Map[ExamSession, List[Student]],
               session: ExamSession): List[Student] =
  registrations.getOrElse(session, List.empty) ①
```

- ① It returns the students for the session if present, an empty list otherwise.

Retrieve the value associated with a key in a dictionary is a fast operation that is constant in time. Scala offers several strategies to do so. They are the following:

- `get` – For an instance of `Map[K, V]`, the method `get` takes one parameter of type `K`, and it returns an `Option[V]`: its value wrapped in a `Some` if the given key exists, `None` if missing.

```
scala> Map(1 -> "a", 2 -> "b").get(2)
res0: Option[String] = Some(b)

scala> Map(1 -> "a", 2 -> "b").get(3)
res1: Option[String] = None
```

- `getOrElse` – For an instance of `Map[K, V]`, the function `getOrElse` takes two parameters: an element of type `K`, and an expression to produce a value of type `V`. If a key equal to the given `K` parameter exists, it returns its associated value. If missing, it will evaluate the provided expression to produce a value of type `V`: your program will do this only if the key is missing, so it can contain side effects.

```
scala> def defaultValue: String = { println("Missing Key!!!"); "N/A" }
defaultValue: String

scala> Map(1 -> "a", 2 -> "b").getOrElse(2, defaultValue)
res2: String = b

scala> Map(1 -> "a", 2 -> "b").getOrElse(3, defaultValue)
Missing key!!!
res3: String = N/A
```

- `apply` – For an instance of `Map[K, V]`, the method `apply` takes one parameter of type `K`, and it returns a value of type `V`: its value if a key equal to the given one exists, it throws a `NoSuchElementException` exception otherwise.

```
scala> Map(1 -> "a", 2 -> "b").apply(2)
res4: String = b

scala> Map(1 -> "a", 2 -> "b").apply(3)
java.util.NoSuchElementException: key not found: 3
  at scala.collection.immutable.Map$Map2.apply(Map.scala:135)
  ... 28 elided
```

This method is unsafe because it throws an exception if the key is missing. You either must be confident that the given key exists or add an ad-hoc error handling strategy to it. Try to use the safe functions `get` and `getOrElse` instead.

Have a look at table 37.1 for a summary of the different methods you can use to retrieve a value associated with a given key.

Table 37.1: Recap of the functions to retrieve a value for a given key for an instance of `Map[K, V]`. The notation `=> V` indicates that your program will evaluate the expression only when needed, which you can also refer to as *lazy evaluation*.

Signature	Description	Example
<code>def get(key: K): Option[V]</code>	It returns the value associated with the given key if the key exists. Otherwise, it returns <code>None</code> .	<code>Map("hi" -> 5).get("hi")</code> // it returns <code>// Some(5)</code>
<code>def getOrElse(key: K, default: => V): V</code>	It returns the value associated with the given, if the key exists. Otherwise, it evaluates the default value.	<code>Map("hi" -> 5) .getOrElse("hello", -1)</code> // it returns <code>// -1</code>
<code>def apply(K): V</code>	It returns the value associated with the given key if present. It throws a <code>NoSuchElementException</code> exception otherwise.	<code>Map("hi" -> 5) .apply("hello")</code> // it throws a <code>// NoSuchElementException</code>

QUICK CHECK 37.1

Implement a function called `getCountry` that takes two parameters: a key-value data structure representing a set of capitals matched to their countries, and a capital. It returns either the capital's country or the text "Unknown".

```
def getCountry(capitalToCountry: Map[String, String],  
capital: String): String = ???
```

37.2 Getting all keys and values

Let's consider your program to track exam registrations and imagine that you need to get its exam sessions and registered students. Have a look at listing 37.2 for a possible implementation:

Listing 37.2: Getting the exam sessions

```
def getExamSessions(registrations: Map[ExamSession, List[Student]]):
    Iterable[ExamSession] =
        registrations.keys ①

def getStudents(registrations: Map[ExamSession, List[Student]]):
    Iterable[Student] =
        registrations.values ②
    .flatten
```

- ① It returns the keys in a Map
 ② It returns the values in a Map

When working with a `Map`, you can retrieve all its keys and values using the following methods:

- `keys` – you can apply the function `keys` on an instance of `Map[K, V]`. It returns a value of type `Iterable[K]` containing the keys in your key-value structure.

```
scala> Map(1 -> "a", 2 -> "b").keys
res0: Iterable[Int] = Set(1, 2)
```

```
scala> Map.empty[String, Int].keys
res1: Iterable[String] = Set()
```

- `values` – For an instance of `Map[K, V]`, the method `values` takes no parameters, and it returns an `Iterable[V]` containing all its values.

```
scala> Map(1 -> "a", 2 -> "b").values
res3: Iterable[String] = MapLike.DefaultValuesIterable(a, b)
```

```
scala> Map.empty[String, Int].values
res4: Iterable[Int] = MapLike.DefaultValuesIterable()
```

Table 37.2 provides a technical summary of the methods you can apply on a key-value structure to retrieve its keys and its values.

Table 37.2: Summary of the functions to retrieve the keys and the values of an instance of `Map[K, V]`.

Signature	Description	Example
<code>def keys: Iterable[K]</code>	It returns the keys of a key-value structure.	<code>Map("hi" -> 5).keys</code> <code>// it returns</code> <code>// Set("hi")</code>
<code>def values: Iterable[V]</code>	It returns the values of a Map.	<code>Map("hi" -> 5).values</code> <code>// it returns</code> <code>//MapLike.DefaultValuesIterable(5)</code>

QUICK CHECK 37.2

Implement a function called `getCapitals` that returns a list of the capitals in a key-value data structure representing a group of capitals and their countries.

```
def getCapitals(capitalToCountry: Map[String, String]): List[String] = ???
```

37.3 Other Operations on Map

Imagine that you'd like to analyze your exam registrations. In particular, you'd like to:

- Check the number of scheduled exam sessions
- Filter the exam sessions on a given date
- Find the exam sessions with the most registrations

Have a look at listing 37.3 for a possible implementation for them:

Listing 37.3: Analyzing exam registrations

```
def totExamSessions(registrations: Map[ExamSession, List[Student]]): Int =
  registrations.size ①

def getExamSessions(registrations: Map[ExamSession, List[Student]],
                    date: LocalDate): Map[ExamSession, List[Student]] =
  registrations.filter { case (session, _) => ②
    session.localDate == date
  }

def getStudents(registrations: Map[ExamSession, List[Student]]):
(ExamSession, List[Student]) =
  registrations.maxBy{ case (_, students) => students.size } ③
```

- ① It returns the number of keys in the key-value structure
- ② It filters the entries based on some of their characteristics
- ③ It selects a maximum element according to given criteria

The `Map` collection implements the `Iterable` interface, so it shares many functionalities with `List`. You can reuse the methods you have discovered in unit 5. You can inquire about its size and the properties of its elements. You can find one of its entries with specific characteristics and pick the minimum/maximum one according to the given criteria. You can filter its keys and values. However, it has the following differences with `List`:

- You cannot sort a `Map`: its keys are unordered by design. You cannot apply the `sorted` and `sortBy` methods on it.
- A key-value structure doesn't have a `distinct` function because its elements are already unique.
- It has a `contains` function, but with a slightly different signature: it takes a key rather than a tuple to represent an entry. For an instance of type `Map[K, V]`, the method `contains` takes one parameter of type `K`. It returns true if a key equal to the given one exists, false otherwise.

```
scala> Map(1 -> "a", 2 -> "b").contains(2)
res0: Boolean = true

scala> Map(1 -> "a", 2 -> "b").contains(3)
res1: Boolean = false
```

Checking if a given key exists in a dictionary is an efficient operation, which completes in a constant time that is independent from its size.

QUICK CHECK 37.3

Implement a function called `getLongestCapitalName` that takes a capital-to-country key-value structure, and it returns the capital with the longest name.

```
def getLongestCapitalName(capitalToCountry: Map[String, String]): String = ???
```

37.4 Summary

In this lesson, my objective was to teach you about the most common operations you can perform on a `Map`.

- You have mastered how to retrieve a value associated with one of its keys.
- You have discovered how to get all its keys and all its values.
- You have learned how to inquire about its size and the feature of its entries.
- You have seen how to select one entry based on its characteristics.
- You have also discovered how to filter its elements according to the given criteria.

Let's see if you got this!

TRY THIS

Suppose you have two key-value structures representing a capital and its country, and a country to its continent. Combine the two instances of `Map` to link each capital to its continent using the retrieve value strategies you have seen in this lesson.

37.5 Answers to Quick Checks

QUICK CHECK 37.1

A possible implementation of the function `getCountry` is the following:

```
def getCountry(capitalToCountry: Map[String, String],
              capital: String): String =
  capitalToCountry.getOrElse(capital, "Unknown")
```

QUICK CHECK 37.2

You can implement the function `getCapitals` as the following:

```
def getCapitals(capitalToCountry: Map[String, String]): List[String] =
  capitalToCountry.keys.toList
```

The return type of your function is `List[String]`, rather than `Iterable[String]`: you need to use the methods `toList` to convert the generic iterable value into a list.

QUICK CHECK 37.3

A possible implementation for the function `getLongestCapitalName` is the following:

```
def getLongestCapitalName(  
    capitalToCountry: Map[String, String]): String = {  
    val (capital, _) = capitalToCountry.maxBy { case (c, _) => c.length }  
    capital  
}
```

38

Either

After reading this lesson, you will be able to:

- Define a value that can have one of two possible types, called `Either`
- Decompose it using pattern matching
- Transform its content using the `map`, and `flatMap` functions
- Chain multiple instances of `Either` using for-comprehension

In the previous lesson, you have mastered the operations you can perform on a `Map`. In this lesson, you'll discover a new Scala type called `Either`. You can use it to represent a value with one of two possible types. You'll learn about its structure and how to define an instance for it. You'll use pattern matching to handle all its possible implementations. You'll transform its values using the `map` and `flatMap` function, and you'll chain multiple values using for-comprehension. You'll use the class `Either` to validate if your library can accept a given book request in the capstone.

Consider this

Imagine you have a list of values representing an online form. You should process it if its items are valid or reject it by explaining why this happened. What type would you use to implement this?

38.1 Why `Either`?

Imagine you want to write a function to validate a phone number. You do not want to throw exceptions because they are difficult to control and too risky, so you decide to use the type `Option`: you return the phone number wrapped in a `Some` if valid, `None` otherwise. It looks similar to the following:

```

private def containsOnlyDigits(phoneNumber: String): Boolean = ???  

private def hasExpectedSize(phoneNumber: String): Boolean = ???  
  

def validatePhoneNumber(phoneNumber: String): Option[String] =  

  if (!containsOnlyDigits(phoneNumber)) None  

  else if (!hasExpectedSize(phoneNumber)) None  

  else Some(phoneNumber)

```

Alternatively, you could also achieve similar results by returning a `Boolean` value. Although these approaches work, they don't provide information on *why* a phone number may be considered invalid. Is it because of its size? Is it because it contains characters that shouldn't be there? Is it because it cannot contain spaces? If you want to find more information, you must read its full implementation and all its checks to find the one that applies to your input. `Boolean` and `Option` are often not expressive enough for validation purposes.

The type `Either` has many uses, but its most popular one is validation. It allows you to return a value that can have one of two possible types. The word "right" is a synonym of "correct" in the English language, so the right side is often mapped to the happy case, while the left one to the unhappy one. An instance of `Either[A, B]` can represent either a failure outcome by returning an instance of type `A` or the successful one by returning one of type `B`. You are going to learn more about this later in this lesson. You can re-implement your `validatePhoneNumber` function using `Either` as the following:

```

private def containsOnlyDigits(phoneNumber: String): Boolean = ???  

private def hasExpectedSize(phoneNumber: String): Boolean = ???  
  

def validatePhoneNumber(phoneNumber: String): Either[String, String] =  

  if (!containsOnlyDigits(phoneNumber))  

    Left("A phone number should only have digits")  

  else if (!hasExpectedSize(phoneNumber))  

    Left("Unexpected number of digits! A number has 10 digits")  

  else Right(phoneNumber)

```

Your function will now provide information on *why* a given phone number isn't valid. For example, when invoking `validatePhoneNumber("hello")`, it returns the `Either` instance `Left("A phone number should only have digits")`.

38.2 Creating an Either

Let's consider your program to track the exam registrations, and imagine you now want to register the outcome of an exam. A mark is either a fail or a pass for ratings of between 60 and 100. Your program should not reveal the score of a failed exam but provide a message for the student. Listing 38.1 shows you how to implement this marking logic using `Either`:

Listing 38.1: Marking Exams using Either

```

case class Pass(score: Int) {  

  require(score >= 60 && score <= 100,  

         "Invalid pass: score must be between 60 and 100") ①  

}  
  

def mark(score: Int, msg: Option[String] = None): Either[String, Pass] =

```

```
if (score >= 60) Right(Pass(score)) ②
else Left(msg.getOrElse("Score below 60")) ③
```

- ① If score is out of range, the instance will throw an exception
- ② Creating the right instance if it is a pass
- ③ Creating the left instance if it is a fail

In Scala, `Either` is an immutable structure to indicate a value with one of two possible values. Listing 38.2 shows its (simplified) definition in the compiler:

Listing 38.2: The structure of Either

```
sealed abstract class Either[A, B]

case class Left[A, B](value: A) extends Either[A, B]

case class Right[A, B](value: B) extends Either[A, B]
```

You cannot initialize the class `Either` directly, as it is abstract. Instead, you need to use one of its two implementations: `Left` and `Right`. For an instance of `Either[A, B]`, provide a value of type `A` to create a `Left[A, B]`, and a value of type `B` for `Right[A, B]`. A few examples of how to create an instance of `Either` are the following:

```
scala> val a: Either[String, Int] = Left("scala")
e: Either[String,Int] = Left(scala)

scala> val b1: Either[String, Int] = Left(42)
<console>:11: error: type mismatch;
 found   : Int(42)
 required: String
      val b1: Either[String, Int] = Left(42)
// Left must contain a value of type String!

scala> val b2: Either[String, Int] = Right(42)
b2: Either[String,Int] = Right(42)

scala> val c: Either[String, Int] = Right("scala")
<console>:11: error: type mismatch;
 found   : String("scala")
 required: Int
      val c: Either[String, Int] = Right("scala")
// Right must contain a value of type Int!
```

Have a look at figure 38.1 provides a visual summary of the structure of an `Either`.

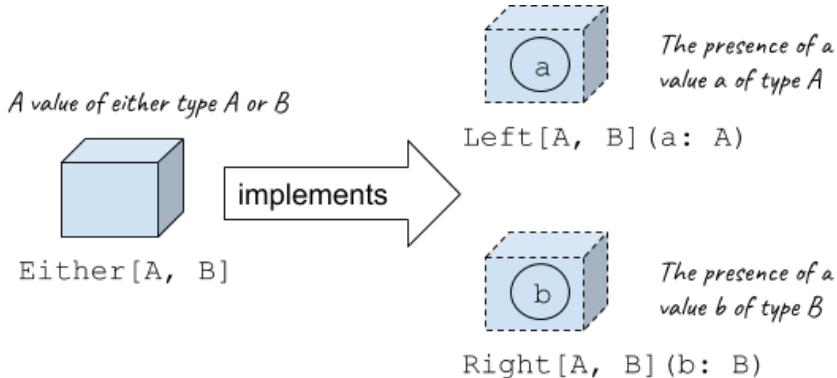


Figure 38.1: The class `Either[A, B]` represents an instance with one of two types. It has two possible implementations: `Left[A, B]` containing a value of type `A`, and `Right[A, B]` representing one of type `B`.

QUICK CHECK 38.1

Implement a function called `sqrt` that takes an integer. It returns the square root of the given number if non-negative, a message otherwise explaining that this operation for negative numbers is not supported.

```
def sqrt(n: Int): Either[String, Double] = ???
```

38.3 Pattern Matching on Either

Let's consider your marking program and imagine you now want to produce a message to the display according to their exam outcome. Have a look at listing 38.3 on how to achieve this using pattern matching:

Listing 38.3: Transforming an exam outcome to a message

```
def toMessage(outcome: Either[String, Pass]): String =
  outcome match {
    case Left(msg) => s"Fail: $msg" ①
    case Right(pass) => s"Pass with score ${pass.score}" ②
  }
```

- ① It matches an instance of `Left`
- ② Evaluated for an instance of `Right`

You need to provide case clauses for both the implementations `Left` and `Right` because the class `Either` is marked as sealed. If you forget to provide any of its implementations, the compiler shows you a warning at compile time that suggests which of its matches is not exhaustive:

```
scala> def toMessage(outcome: Either[String, Pass]): String =
|   outcome match {
|     case Right(pass) => s"Pass with score ${pass.score}"
|   }
<console>:14: warning: match may not be exhaustive.
```

```
It would fail on the following input: Left(_)
// Left(_) indicates a Left instance containing any value
```

QUICK CHECK 38.2

Implement a method called `sqrtOrZero` that takes an integer. It uses the function `sqrt` that you have implemented in the previous quick check to compute the square root of the number. Use pattern matching to return zero when the operation is not supported.

```
def sqrtOrZero(n: Int): Double = ???
```

38.4 The map and flatMap operations

The class `Either` has an implementation for the `map` and `flatMap` functions: you'll learn how to use them in the following subsections. Starting from Scala 2.13, it also has a `flatten` method, but we'll not discuss it because it's rarely used.

38.4.1 The map function

Let's consider your program to mark exams, and imagine that you'd like to provide an alternative score visualization. Rather than displaying the score itself, you'd like to convert it to a percentage. For example, rather than having a score of 60 with no indication about the maximum possible achievable mark, you may want to represent as 0.60 to indicate that 60% of it was correct. Instead of using a pattern matching expression, you can use the `map` function in the following way:

Listing 38.4: Transforming a pass into a percentage

```
case class Pass(score: Int) {
    require(score >= 60 && score <= 100,
           "Invalid pass: score must be between 60 and 100")

    def toPercentage: Double = score / 100.0
}

def toPercentage(outcome: Either[String, Pass]): Either[String, Double] =
    outcome.right.map(_.toPercentage) ①
```

① It accesses the value wrapped into the Right instance, if any. You can omit the function `right` if using Scala 2.12 or higher

You can use the `map` function to transform the value wrapped into a `Left` or `Right` instance of `Either`. The compiler needs to know if you'd like to manipulate the left or the right side of `Either`: the methods `left` and `right` allow you to specify which side to consider. The function `map` is a higher order function on `Either[A, B]` that takes a parameter `f`. When applied on its left projection (i.e., when you want to manipulate its left side), it takes a parameter `f` of type `A => C` to produce a value of type `Either[C, B]`. When working on its right projection (i.e., when you'd like to transform its right side), it takes a parameter `f` of type `B => C` to produce a value of type `Either[A, C]`. It behaves as follows:

- if your instance matches the selected side, it applies the function `f` to its value.
- If not, it returns the instance without performing any transformation.

A few examples on `map` on an instance of `Either` are following:

```
scala> val e: Either[String, Int] = Right(42)
e: Either[String,Int] = Right(42)

scala> e.left.map(_.size)
res0: scala.util.Either[Int,Int] = Right(42)
// Its left side has now type Int rather than String
// but your instance has not been modified

scala> e.right.map(_ * 2.0)
res1: scala.util.Either[String,Double] = Right(84.0)
// Your instance is a right, so it transforms it into a Double
// by multiplying it by 2.0
```

When developers use the type `Either` for validation purposes, they usually assume that its right side represents the happy case, while the left one for the rejection case. After all, the term “right” is a synonym of “correct” in the English language. Starting from Scala 2.12, the class `Either` is *right-biased*: when making no selection, the compiler will assume you’d like to manipulate its right side. You can omit the invocation to the `right` method for Scala 2.12+:

```
scala> e.map(_ * 2.0)
res2: scala.util.Either[String,Double] = Right(84.0)
// No need to invoke right as Either is right-biased
```

QUICK CHECK 38.3

Implement a function called `truncate` that takes a parameter of type `Either[Double, String]`, and it truncates its text to its first 24 characters, if any. Use the `map` method on `Either`.

```
def truncate(e: Either[Double, String]): Either[Double, String] = ???
```

38.4.2 The flatMap function

Imagine that a student has received two exam marks, and you’d like to combine them: you should return their average score if they both pass, a fail otherwise. Listing 38.5 shows you how to implement this:

Listing 38.5: Combining exam outcomes

```
def combine(outcomeA: Either[String, Pass],
           outcomeB: Either[String, Pass]): Either[String, Pass] = {
  outcomeA.flatMap { passA => ①
    outcomeB.map { passB =>
      val averageScore = (passA.score + passB.score) / 2
      Pass(averageScore)
    }
  }
}
```

① You can avoid calling the `right` method if you are using Scala 2.12+

When working with an instance of `Either[A, B]`, the `flatMap` function a parameter `f` that will have a different signature depending on which side of either you select to transform using the methods `left` and `right`. When applied on its left side, it takes a parameter `f` of type `A => Either[C, D]`, and returns a value of type `Either[C, BD]`: the compiler infers `BD` as the type of the common

superclass between `B` and `D`. When transforming its right projection, it takes a parameter `f` of type `B => Either[C, D]` and it produces a value of type `Either[AC, D]`: the compiler infers `AC` as the type of the common superclass between `A` and `C`. It behaves as follows:

- If your instance matches the selected side, it applies the function `f` to its content to produce a new `Either` value.
- If not, no transformation happens.

A few examples on how to use `flatMap` on an instance of `Either` are the following:

```
scala> val e: Either[String, Int] = Right(25)
e: Either[String,Int] = Right(25)

scala> e.flatMap { n =>
    |   if (n < 0) Left(s"Found Negative number $n")
    |   else Right(Math.sqrt(n))
    |
}
res0: scala.util.Either[String,Double] = Right(5.0)
// The compiler infers the type String for its left side

scala> e.flatMap { n =>
    |   if(n < 0) Left(-1)
    |   else Right(Math.sqrt(n))
    |
}
res1: scala.util.Either[Any,Double] = Right(5.0)
// The left side has type Any because it is
// the common superclass between String and Int

scala> e.left.flatMap { text =>
    |   if (text.isEmpty) Right(42.0)
    |   else Left(text)
    |
}
res1: scala.util.Either[String,Double] = Right(42.0)
// The right side has type Double because it is
// the common superclass between Int and Double
```

QUICK CHECK 38.4

Consider the following snippet of code:

```
def validation(a: Either[String, Int],
              b: Either[String, Int]): Either[String, Int] =
  a.flatMap { aa =>
    b.map(bb => aa + bb)
  }
```

What value does the following function call return? Why? Use the REPL to validate your hypothesis.

```
validation(Left("first failure"), Left("second failure"))
```

Have a look at table 38.1 for a summary of the functions `map` and `flatMap` on an instance of `Either`.

Table 38.1: Recap of the map and flatMap operations on an instance of Either [A, B]. Where necessary, the method signature has been simplified to hide non-relevant implementation details.

	Acts on	Signature	Usage
map	Either[A, B] .left	map(f: A => C): Either[C, B]	It applies a transformation f to the value of its left projection.
	Either[A, B] .right (default from Scala 2.12+)	map(f: B => C): Either[A, C]	It applies a transformation f to the value of its right project.
flatMap	Either[A, B] .left	flatMap(f: A => Either[C, D]): Either[C, BD]	It produces a new Either instance by applying the function f to the value of its left projection. BD is the type of the superclass of both B and D.
	Either[A, B] .right (default from Scala 2.12+)	flatMap(f: B => Either[C, D]): Either[AC, D]	It produces a new Either instance by applying the parameter f to the value of its right projection. AC is the type of the superclass of both A and C.

38.5 For-comprehension

Let's consider your implementation in listing 38.5 to combine the outcome of two exams. You could refactor it using for-comprehension as follows:

Listing 38.6: Combining exam outcomes using for-comprehension

```
def combine(outcomeA: Either[String, Pass],  
           outcomeB: Either[String, Pass]): Either[String, Pass] =  
  for {  
    passA <- outcomeA  
    passB <- outcomeB  
  } yield {  
  val averageScore = (passA.score + passB.score) / 2  
  Pass(averageScore)  
}
```

In Scala, you can use for-comprehension instead of using the `map` and `flatMap` functions on an instance of `Either`. For example, these expressions are equivalent:

```
scala> val value: Either[Int, String] = Left(1234)  
value: Either[Int, String] = Left(1234)  
  
scala> value.left.map(v => v + 1)  
res0: scala.util.Either[Int, String] = Left(1235)
```

```
scala> for {
|   v <- value.left
| } yield v + 1
res1: scala.util.Either[Int, String] = Left(1235)
```

QUICK CHECK 38.5

Re-implement the following snippet of code using for-comprehension:

```
def validation(a: Either[String, Int], b: Either[String, Int]): Either[String, Int] =
  a.flatMap { aa =>
    b.map(bb => aa + bb)
  }
```

38.6 Summary

In this lesson, my objective was to teach you about the class `Either`.

- You have learned that it has two possible implementations, called `Left` and `Right`, and how to instance them.
- You have seen how to use pattern matching on it.
- You have mastered how to manipulate its left and right projections using the `map` and `flatMap` functions.
- You have also discovered how to use for-comprehension to combine `Either` values.

Let's see if you got this!

TRY THIS

In Quick Check 38.4, you have seen that the given implementation of the function `validation` doesn't behave as expected when you need to accumulate errors. Change its return type to `Either[List[String], Int]`, and its implementation to address this issue.

HINT: Use pattern matching.

```
def validation(a: Either[String, Int],
  b: Either[String, Int]): Either[List[String], Int] = ???
```

38.7 Answers to Quick Checks

QUICK CHECK 38.1

A possible implementation for the function `sqrt` is the following:

```
def sqrt(n: Int): Either[String, Double] =
  if (n < 0) Left("Operation not supported for negative numbers")
  else Right(Math.sqrt(n))
```

QUICK CHECK 38.2

You can implement the function `sqrtOrZero` using pattern matching as follows:

```
def sqrtOrZero(n: Int): Double =
  sqrt(n) match {
    case Left(_) => 0
    case Right(d) => d
  }
```

QUICK CHECK 38.3

If your version of Scala is 2.12+, you can implement the function `truncate` as follows:

```
def truncate(e: Either[Double, String]): Either[Double, String] =
  e.map(_.take(24))
```

If you are using an older version, you need to explicitly indicate that you'd like to transform its right side:

```
def truncate(e: Either[Double, String]): Either[Double, String] =
  e.right.map(_.take(24))
```

QUICK CHECK 38.4

The function `validation` returns `Left("first failure")` because the `flatMap` operation doesn't evaluate its parameter if finds an instance of `Left`:

```
scala> validation(Left("first failure"), Left("second failure"))
res11: Either[String,Int] = Left(first failure)
```

The type `Either` does not work well when error accumulation is needed. Consider using `cats.data.Validated` from the typelevel `cats` project instead. Although this is not part of the standard library, it is relatively popular because it introduces more advanced functional programming techniques, such as applicatives, a topic which we'll unfortunately not cover in this book.

QUICK CHECK 38.5

You can re-implement the snippet of code using for-comprehension as follows:

```
def validation(a: Either[String, Int], b: Either[String, Int]): Either[String, Int] =
  for {
    aa <- a
    bb <- b
  } yield aa + bb
```

39

Working with Either

After reading this lesson, you will be able to:

- Retrieve a value wrapped in the left or right side for `Either`.
- Check if an instance is its left or right projection.
- Inquire if its value respects a given predicate.

In the previous lesson, you have learned the structure of `Either` and its basic operations. In this lesson, you'll learn about other useful methods the class `Either` has to offer. They are very similar to the one you have seen for `Option`. They do not share an interface, but this is a happy consequence of the consistent design and style of the Scala collections. You'll discover how to retrieve a value defined for its left or right side. You'll also inquire about its properties to check if a given instance is of type `Left` or `Right` and if it contains a value with specific features. You will use these operations to determine if your library has accepted a book loaning request in the capstone.

Consider this

Imagine you need to validate a text representing an email address. If invalid, you should replace it with the string "Unknown". How would you implement it using `Either`?

39.1 Retrieving an Either value

In the previous lesson, you have implemented a program to mark exams that produces a value of type `Either[String, Pass]` in which its left side represents a failure together with a message for the student, and its right side a pass with the obtained score:

Listing 39.1: Marking exams strategy

```
case class Pass(score: Int) {
  require(score >= 60 && score <= 100,
         "Invalid pass: score must be between 60 and 100")
}

def mark(score: Int, msg: Option[String] = None): Either[String, Pass] =
  if (score >= 60) Right(Pass(score))
  else Left(msg.getOrElse("Score below 60"))
```

Imagine you'd like to produce a message to anticipate the overall outcome of an exam. Rather than using pattern matching, you can use the `getOrElse` method on `Either`. Listing 39.2 shows you how to achieve this:

Listing 39.2: Generating a preview message

```
def getPreviewMessage(outcome: Either[String, Pass]): String =
  outcome.left.getOrElse("You passed the exam, well done!") ①
```

① It retrieves the value if `outcome` is left; it returns the given default value otherwise.

When retrieving a value of an `Either`, you need to indicate which side you'd like to consider: you can use the `left` and `right` functions to do so. Then, you can invoke the `getOrElse` method to retrieve its value or generate an alternative one. Your program evaluates the default only if needed so that it can contain side effects.

```
scala> Right(42).getOrElse(0)
res0: Int = 42
// You can omit the call to the right function because
// Either is right from Scala 2.12+

scala> Left("hello").left.getOrElse("scala")
res4: String = hello

scala> Right(42).left.getOrElse("scala")
res5: String = scala

scala> Right(42).getOrElse { println("generating default..."); 0 }
res1: Int = 42
// It doesn't evaluate the default expression

scala> Left("hello").getOrElse { println("generating default..."); 0 }
generating default...
res2: Int = 0
// It executes the default expression
```

Table 39.1 provides a recap on the `getOrElse` function to retrieve a left or right value of an `Either` instance.

Table 39.1: Recap of the map and flatMap operations on an instance of Either[A, B]. The expressions => A and => B indicate that your program will evaluate their values only when needed, which you can also refer to as lazy evaluation.

	Acts on	Signature	Usage
getOrElse	Either[A, B] .left	getOrElse(default: => A): A	It returns the value of its left projection if present. It evaluates default to produce a value A otherwise.
	Either[A, B] .right (default from Scala 2.12+)	getOrElse(default: => B): B	It retrieves the value of its right projection. If missing, it executes default to create a value of type B.

QUICK CHECK 39.1

Implement a function called `getOrZero` that takes a value of type `Either[String, Double]` as its parameter. It returns the value wrapped in its right projection or zero.

```
def getOrZero(value: Either[String, Double]): Double = ???
```

39.2 Properties of an Either value

Imagine that you need to add extra functionalities to your program to mark exams. In particular, you need to:

- Determine if a given outcome is a pass.
- Check if a mark is a distinction, which is a pass with a score of 80 or higher.

Have a look at listing 39.3 for a possible implementation for it:

Listing 39.3: Checks on the outcome of an exam

```
def isPass(outcome: Either[String, Pass]): Boolean =
  outcome.isRight ①

def isDistinction(outcome: Either[String, Pass]): Boolean =
  outcome.exists(pass => pass.score >= 80) ②
```

① It returns true if the instance is a right projection, false otherwise

② It returns true if it is a right projection containing a value that respects the given predicate.

When inquiring on the properties on `Either`, you can use the following methods:

- `isLeft` – the function `isLeft` returns true if the instance is of type `Left`, false otherwise.

```
scala> Right(42).isLeft
res0: Boolean = false

scala> Left("hello").isLeft
res1: Boolean = true
```

- `isRight` – the method `isRight` is the complementary of `isLeft`. It returns `true` for an instance that is a right projection, `false` otherwise.

```
scala> Right(42).isRight
res0: Boolean = true

scala> Left("hello").isRight
res1: Boolean = false
```

- `exists` – For an instance of `Either[A, B]`, the method `exists` takes one parameter. It takes a predicate function of type `A => Boolean` for its left projection, or one of type `B => Boolean` for its right one. It returns `true` if the instance matches the selected projection, and its value respects the given predicate, `false` otherwise.

```
scala> val e: Either[String, Int] = Left("hello")
e: Either[String,Int] = Left(hello)

scala> e.exists(_ > 0)
res0: Boolean = false
// Omitting the function call to right because Either is right-biased

scala> e.left.exists(_.startsWith("scala"))
res1: Boolean = false

scala> e.left.exists(_.size > 3)
res2: Boolean = true
```

Table 39.2 summarizes the methods to analyze the properties of an `Either`.

Table 39.2: Summary of the function to inquiry about the properties of an instance of Either [A, B].

	Acts on	Signature	Usage
isLeft	Either[A, B]	isLeft: Boolean	It returns true if the instance is a left projection, false otherwise.
isRight	Either[A, B]	isRight: Boolean	It returns true if the instance is right, false otherwise.
exists	Either[A, B].left	exists(p: A => Boolean): Boolean	It returns true if the instance is a left projection and its value respects the given predicate p, false otherwise.
	Either[A, B].right (default from Scala 2.12+)	exists(p: B => Boolean): Boolean	It returns true if the instance is right, and if it contains an element that asserts the predicate p, false otherwise.

QUICK CHECK 39.2

Implement a function called `isPositive` that takes a value of type `Either[String, Double]`, and it returns `true` if it contains a double bigger than zero, `false` otherwise.

```
def isPositive(value: Either[String, Double]): Boolean = ???
```

39.3 Summary

In this lesson, my objective was to teach you about the operations you can perform on an instance of `Either`.

- You have seen how to retrieve the value in one of its projections with a default for it.
- You have learned how to discover if an instance is a left or right side for an `Either`.
- Also, you have how to enquire about the properties of the value wrapped in a projection.

Let's see if you got this!

TRY THIS

Implement a function that takes either a string or an integer, and it returns `true` if the given text is "Scala", `false` otherwise.

39.4 Answers to Quick Checks

QUICK CHECK 39.1

You could implement the function `getOrZero` as follows:

```
def getOrZero(value: Either[String, Double]): Double =  
  value.getOrElse(0)
```

QUICK CHECK 39.2

A possible implementation for the function `isPositive` is the following:

```
def isPositive(value: Either[String, Double]): Boolean =  
  value.exists(_ > 0)
```

If your Scala version is older than 2.12, you need to explicitly indicate to the compiler that you'd like to consider its right projection:

```
def isPositive(value: Either[String, Double]): Boolean =  
  value.right.exists(_ > 0)
```

40

Error Handling with Try

After reading this lesson, you will be able to:

- Represent a computation that can fail using `Try`
- Decompose it using pattern matching
- Manipulate its value using the `map`, `flatten`, and `flatMap` operations
- Chain several `Try` instances using for-comprehension
- Check if a computation has failed and retrieve its value.

In the previous lessons, you have mastered the type `Either`. In this lesson, you'll learn about the class `Try` to represent a computation that can fail. Throwing exceptions is a risky and unpredictable practice. Knowing which exceptions a method may throw is challenging, if not impossible, as they are often not annotated, and they are highly dependent on the specifics of their implementation. The class `Try` allows you to control a computation that may throw exceptions, and it forces you to provide instruction for both the success and failure cases at compile time. Whenever possible, you should use `Try` instead of a try-catch expression. You'll find several similarities with the type `Either`: you can see `Try[T]` as an alternative implementation for `Either[Throwable, T]`. You'll learn about the purpose and structure of the type `Try`, and you'll use pattern matching to decompose its possible implementations. You'll transform and manipulate its value using its `map`, `flatten`, and `flatMap` methods. You'll combine multiple operations that can fail into one. You'll also inquire about the success of some computation and retrieve its value. In the capstone, you will use the type `Try` to handle possible failures when parsing data from a CSV to create a book collection for your library.

Consider this

Imagine you are developing software to check if a given postcode is correct. It relies on a third-party that it invokes via a public HTTP API. What happens if your program can no longer connect to it? How would you ensure that it can survive this unforeseen outage?

40.1 Creating a Try

Imagine you are developing software to register students for exam sessions. You need to ensure that students register only for exam sessions of topics they are selected. Listing 40.1 shows you to represent that the registration may fail using `Try`:

Listing 40.1: Registering a student for an exam session

```
import java.time.LocalDate
import scala.util.{Failure, Success, Try} ①

case class Student(id: Int, name: String, topics: Set[String])
case class ExamSession(title: String, localDate: LocalDate, topic: String)

case class Registration(studentId: Int,
                       examSession: ExamSession,
                       localDate: LocalDate = LocalDate.now()) ②

def register(student: Student,
             examSession: ExamSession): Try[Registration] = {
  if (student.topics.contains(examSession.topic))
    Success(Registration(student.id, examSession)) ③
  else
    Failure(new IllegalStateException(④
      s"Student ${student.id} is missing topic ${examSession.topic}"))
}
```

- ① Importing `Try` and its implementations into the current scope
- ② `LocalDate.now()` returns today's date
- ③ Creating the success instance if the condition is true
- ④ Creating the failure instance if the condition is false

In Scala, you can use the class `Try` to represent a computation that can fail. It lives in the package `scala.util`, and it has two possible implementations to define the success and failure cases. Listing 40.2 shows you its (simplified) structure:

Listing 40.2: The structure of Try

```
package scala.util

sealed abstract class Try[T]

case class Success[T](t: T) extends Try[T]

case class Failure[T](throwable: Throwable) extends Try[T]
```

The type `Try` is not available in your scope by default, but you need to add the instruction `import scala.util.Try` before you can use it. It is abstract, so you cannot initialize it directly. You should use one of its two implementations: `scala.util.Success` and `scala.util.Failure`. When creating an instance of `Failure`, you need to provide a throwable. The class `java.lang.Throwable` is a Java type representing everything that your system can throw: `Exception` is a subclass of `Throwable`. A few examples of how to create instances of `Try` are the following:

```
scala> import scala.util._  
import scala.util._  
// Adding Try, Success, Failure to your scope  
  
scala> val a: Try[Int] = Success(1)  
a: scala.util.Try[Int] = Success(1)  
  
scala> val b: Try[Int] = Success("scala")  
<console>:14: error: type mismatch;  
  found   : String("scala")  
  required: Int  
        val b: Try[Int] = Success("scala")  
// Success must contain an integer!  
  
scala> val b: Try[Int] = Failure(new Exception("error!"))  
b: scala.util.Try[Int] = Failure(java.lang.Exception: error!)  
  
scala> val b: Try[Int] = Failure("error!")  
<console>:14: error: type mismatch;  
  found   : String("error!")  
  required: Throwable  
        val b: Try[Int] = Failure("error!")  
// String does not implement Throwable!
```

Have a look at figure 40.1 for a visual summary of the structure of `Try`.

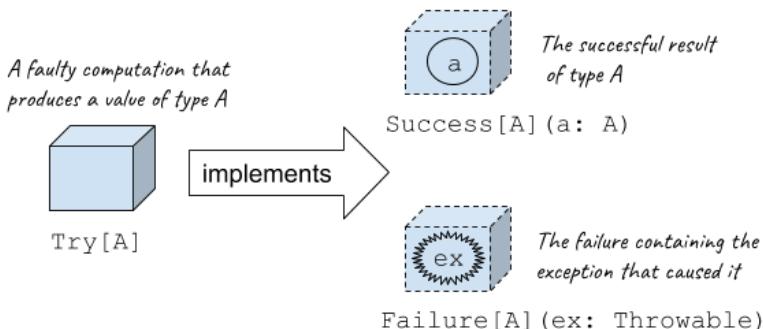


Figure 40.1: Use the class `Try[A]` to represent a computation that can fail in producing a value of type `A`. It has only two possible implementations: `Success` and `Failure`. `Success[A]` represents a completed operation containing the created instance of type `A`, while `Failure[A]` represents a faulty execution containing the exception that caused it.

When initializing an instance of `Try`, you can also use its `apply` method

Listing 40.3: The method Try.apply

```
package scala.util

import scala.util.control.NonFatal

object Try {

    def apply[T](r: => T): Try[T] = ①
        try Success(r) catch {
            case NonFatal(e) => Failure(e) ②
        }
}
```

① The parameter `r` is not evaluated until needed

② `NonFatal` matches throwable instances that are recoverable

Its `apply` method takes one parameter returning a value of type `T`. The symbol `=>` `T` indicates to delay its evaluation until its invocation. It executes the value `r` inside a try-catch expression to handle recoverable exceptions. `NonFatal` matches only `Throwable` instances that Scala considers recoverable. `OutOfMemoryError`, `ThreadDeath`, and `InterruptedException` are some of the ones it marks as fatal. You can rewrite the function in listing 40.1 using the function `Try.apply` as the following:

Listing 40.4: Registering a student for an exam session using Try.apply

```
def register(student: Student,
            examSession: ExamSession): Try[Registration] =
    Try { ①
        if (student.topics.contains(examSession.topic))
            Registration(student.id, examSession)
        else
            throw new IllegalStateException(
                s"Student ${student.id} is missing topic ${examSession.topic}")
    }
```

① Expression equivalent to `Try.apply`

Creating a `Try` instance by initializing its implementations or using the `Try.apply` method is equivalent and up to your preference. A few more examples of how to use the `Try.apply` function are the following:

```
scala> import scala.util.Try
import scala.util.Try
// Adding the type Try into your scope

scala> val b: Try[Int] = Try(10/2)
b: scala.util.Try[Int] = Success(5)
// equivalent to Try.apply(10/2)

scala> val b: Try[Int] = Try(10/0)
b: scala.util.Try[Int] = Failure(java.lang.ArithmaticException: / by zero)
// The expression 10/0 throws an ArithmaticException instance
```

QUICK CHECK 40.1

Implement a function called `toTry` that converts an instance of `Either[Throwable, T]` into one of type `Try[T]`.

```
def toTry[T](either: Either[Throwable, T]): Try[T] = ???
```

40.2 Pattern Matching on Try

Let's consider your program to register a student for an exam session. After you try to produce a registration, you now need to analyze its result to produce a message to display to the user. Listing 40.5 shows how to do this using pattern matching:

Listing 40.5: Producing a message for a possibly faulty registration

```
def toPrettyMsg(registration: Try[Registration]): String =
  registration match {
    case Success(reg) => ①
      s"Student registered for exam session ${reg.examSession.title}"
    case Failure(ex) => ②
      s"Registration failed: ${ex.getMessage}"
  }
```

- ① It matches a successful registration
- ② It matches a failed computation

The implementation of `Try` uses the keyword `sealed`: you need to provide cases for both `Success` and `Failure` when defining pattern matching to prevent the compiler from producing a warning.

```
scala> import scala.util._
import scala.util._

scala> Try(1) match {
|   case Success(n) => n
| }
<console>:16: warning: match may not be exhaustive.
It would fail on the following input: Failure(_)
          Try(1) match {
            ^
res1: Int = 1
```

QUICK CHECK 40.2

Implement a function called `toEither` that converts an instance of type `Try[T]` into one of type `Either[Throwable, T]`.

```
def toEither[T](tryT: Try[T]): Either[Throwable, T] = ???
```

40.3 The map, flatten, and flatMap operations

The class `Try` has an implementation for the methods `map`, `flatten`, and `flatMap`. They behave similarly to those for `Option`. Let's see them in action in the following subsections.

40.3.1 The map function

Suppose you now need to extract the date of an exam registration. You can achieve this using the function `map`:

Listing 40.6: Getting the date of an exam registration

```
def getRegistrationDate(registration: Try[Registration]): Try[LocalDate] =  
  registration.map(_.localDate)
```

You can use the function `map` to transform the value that is the result of a successful computation. If your instance is of type `Failure`, nothing happens:

```
scala> import scala.util._  
import scala.util._  
  
scala> Try(5).map(_ + 1.23)  
res0: scala.util.Try[Double] = Success(6.23)  
  
scala> Try(5/0).map(_ + 1.23)  
res1: scala.util.Try[Double] = Failure(java.lang.ArithmeticException: / by zero)
```

QUICK CHECK 40.3

Implement a function to transform an instance of `Try[Double]` into one of `Try[Float]` using the `map` method.

```
def toFloat(d: Try[Double]): Try[Float] = ???
```

40.3.2 The flatten function

Suppose that your program should allow students to register for the next exam session available for a given topic. Imagine that your code has a function called `getNextExamSession` that tries to find the next exam session: it will return a failure if it can't select one.

```
def getNextExamSession(topic: String): Try[ExamSession] = ???
```

You could compose the `getNextExamSession` with the function `register` you have implemented in listing 40.1 by chaining the two values using the `map` method:

```
def registerForNextExamSession(student: Student,  
                               topic: String): Try[Try[Registration]] =  
  getNextExamSession(topic).map { examSession =>  
    register(student, examSession)  
  }
```

This function produces a value of type `Try[Try[Registration]]`: a failure in the outer `Try` indicates an issue in the exam session selection, while one in the inner instance suggests a problem with the registration process. This distinction may not be informative to the user, and you may like to unify all the failures independently by what caused them. Listing 40.7 shows you how to do this using the `flatten` function:

Listing 40.7: Registering a student for the next exam session

```
import scala.util.Try

def registerForNextExamSession(student: Student,
                               topic: String): Try[Registration] =
  getNextExamSession(topic).map { examSession =>
    register(student, examSession)
  }.flatten
```

You can use the method `flatten` to merge two nested `Try` instances into one: it acts on an instance of `Try[Try[T]]` to produce a value of type `Try[T]`:

```
scala> import scala.util._
import scala.util._

scala> Try(Try(1)).flatten
res0: scala.util.Try[Int] = Success(1)

scala> Try(Try(1/0)).flatten
res1: scala.util.Try[Int] = Failure(java.lang.ArithmeticsException: / by zero)

scala> Try(1).flatten
<console>:15: error: Cannot prove that Int <:< scala.util.Try[U].
          Try(1).flatten
// You can invoke the function flatten only on nested structures
```

QUICK CHECK 40.4

Implement a function called `superFlatten` that takes an instance of `Try[Try[T]]`, and it returns one of type `Try[T]`.

```
def superFlatten[T](tryT: Try[Try[T]]): Try[T] = ???
```

40.3.3 The flatMap function

Let's consider the function `registerForNextExamSession` you have implemented in listing 40.7. You can also rewrite it using the `flatMap` function:

Listing 40.8: Registering for next exam session using flatMap

```
import scala.util.Try

def registerForNextExamSession(student: Student,
                               topic: String): Try[Registration] =
  getNextExamSession(topic).flatMap { examSession =>
    register(student, examSession)
  }
```

When performing a `map` operation followed by a `flatten` one, you should combine them and invoke the `flatMap` function instead. A few more examples of how to use it are the following:

```
scala> import scala.util._
import scala.util._

scala> Try(1).flatMap(n => Try(2/n))
```

```

res0: scala.util.Try[Int] = Success(2)
scala> Try(0).flatMap(n => Try(2/n))
res1: scala.util.Try[Int] = Failure(java.lang.ArithmetricException: / by zero)

```

QUICK CHECK 40.5

Imagine you have implemented a function to find a student by an id:

```

import scala.util._

case class Student(id: Int, name: String, topics: Set[String])
def findStudent(id: Int): Try[Student] = ???

```

Implement another function with the same name that takes a string as its parameter, and it returns an instance of `Try[Student]` by reusing the existing `findStudent(id: Int)` function.

```
def findStudent(id: String): Try[Student] = ???
```

HINT: You can parse a string instance into an integer using the `toInt` function, which throws an exception if this is not possible.

Have a look at table 40.1 for a summary of the `map`, `flatten`, `flatMap` functions acting on an instance of type `Try`.

Table 40.1: Technical recap of the three fundamental operations on Try. The function map applies a given function to the successful result of a computation, while the method flatten creates a new instance of Try by merging two nested structures. The flatMap operation combines the map and flatten ones to chain values together.

	Acts on	Signature	Usage
<code>map</code>	<code>Try[A]</code>	<code>map(f: A => B): Try[B]</code>	It applies a function to the successful result of the computation, if any.
<code>flatten</code>	<code>Try[Try[A]]</code>	<code>flatten: Try[A]</code>	It merges two nested Try instances into one.
<code>flatMap</code>	<code>Try[A]</code>	<code>flatMap(f: A => Try[B]): Try[B]</code>	The combination of map followed by flatten. It chains Try instances together.

40.4 For-comprehension

Let's look at the function you have implemented in listing 40.8 to register a student for the next exam session available. You could rewrite it using for-comprehension:

Listing 40.9: Registering for the next exam session using for-comprehension

```
import scala.util.Try

def registerForNextExamSession(student: Student,
                                topic: String): Try[Registration] =
  for {
    examSession <- getNextExamSession(topic) ①
    registration <- register(student, examSession) ②
  } yield registration
```

- ① It extracts the found exam session, if any
 ② It binds the successful registration the registration variable, if any

You can use for-comprehension to express a chain of `flatMap` and `map` operations to improve their readability. For example, you can rewrite the expression `Try(1).flatMap(n => Try(2/n))` using for-comprehension:

```
for {
  n <- Try(1)
  res <- Try(2/n)
} yield res
```

QUICK CHECK 40.6

Rewrite the function `findStudent(id: String)` that you have implemented in quick check 40.5 using for-comprehension.

40.5 Other operations on Try

Suppose that your program has logic to select the next available exam session for a given topic, and you'd like to perform the following operations on it:

- Check if it was able to find an exam session
- Retrieve the selected one or provide an alternative to the student

Listing 40.10 shows you how to implement this.

Listing 40.10: Checking if registration is successful

```
import java.time.LocalDate
import scala.util.Try

def exists(nextSession: Try[ExamSession]): Boolean =
  nextSession.isSuccess

private val defaultExamSession: ExamSession = ???
def getExamSession(nextSession: Try[ExamSession]): ExamSession =
  nextSession.getOrElse(defaultExamSession)
```

The class `Try` offers a few helper functions that you can use as alternatives to pattern matching. The most common listed below:

- `isSuccess` – The function `isSuccess` returns `true` if the instance is of type `Success`.

```
scala> Try(5/2).isSuccess
res0: Boolean = true
```

- `isFailure` – The method `isFailure` is complementary to `isSuccess`: it returns `true` if the instance is of type `Failure`.

```
scala> Try(5/2).isFailure
res0: Boolean = false
```

- `getOrElse` – For an instance of `Try[T]`, the function `getOrElse` takes an expression that produces a value of type `T` as its parameter. It returns its value if your instance is of type `Success`. In case of failure, it evaluates the given parameter to produce an alternative value to return only if needed to handle side effects correctly.

```
scala> Try(5/2).getOrElse { println("Side Effect!"); 42 }
res9: Int = 2

scala> Try(5/0).getOrElse { println("Side Effect!"); 42 }
Side Effect!
res10: Int = 42
```

QUICK CHECK 40.7

The class `String` offers a method called `toBoolean` to convert text into a boolean value, which is unsafe because it throws an exception for any string that does not match "true" or "false". Implement a function called `toSafeBoolean` by reusing `toBoolean` and returning `false` rather than throwing an exception.

```
def toSafeBoolean(text: String): Boolean = ???
```

40.6 Summary

In this lesson, my objective was to teach you about the type `Try`.

- You have learned how to create a `Try` instance by initializing one of its implementations or by using its `apply` method.
- You have seen how to manipulate and chain its values using the `map`, `flatten`, `flatMap` operations, and for-comprehensions.
- Also, you have discovered how to check if a `Try` instance is successful and how to retrieve its value.

Let's see if you got this!

TRY THIS

Imagine you are developing software to read and manipulate data from a file. Implement a function to parse a string into an instance of `Person`:

```
case class Person(age: Int, name: String)
```

For example, the text "35, Jane Doe" should result in a `Person` instance equal to `Person(35, "Jane Doe")`. Make sure not to throw exceptions if your function cannot parse the given text by returning an informative failure without throwing exceptions.

HINT: You can use the method `split` to tokenize a string.

```
scala> "a-b-c".split("-").toList
res0: List[String] = List(a, b, c)

scala> "a-b-c".split("-", 2).toList // limiting to up to two tokens
res1: List[String] = List(a, b-c)
```

40.7 Answers to Quick Checks

QUICK CHECK 40.1

A possible implementation for the function `toTry` is the following:

```
import scala.util._

def toTry[T](either: Either[Throwable, T]): Try[T] =
  either match {
    case Left(ex) => Failure(ex)
    case Right(t) => Success(t)
  }
```

QUICK CHECK 40.2

You can implement the method `toEither` as the following:

```
import scala.util._

def toEither[T](tryT: Try[T]): Either[Throwable, T] =
  tryT match {
    case Success(t) => Right(t)
    case Failure(ex) => Left(ex)
  }
```

QUICK CHECK 40.3

A possible implementation for the function `toFloat` is the following:

```
import scala.util.Try

def toFloat(d: Try[Double]): Try[Float] = d.map(_.toFloat)
```

QUICK CHECK 40.4

You can implement the function `superFlatten` by invoking the method `flatten` twice:

```
import scala.util._

def superFlatten[T](tryT: Try[Try[Try[T]]]): Try[T] = tryT.flatten.flatten
```

QUICK CHECK 40.5

A possible implementation for the function `findStudent` is the following:

```
def findStudent(id: String): Try[Student] =
  Try(id.toInt).flatMap(findStudent)
```

QUICK CHECK 40.6

Your function `findStudent` should look similar to the following:

```
def findStudent(id: String): Try[Student] =  
  for {  
    n <- Try(id.toInt)  
    student <- findStudent(n)  
  } yield student
```

QUICK CHECK 40.6

A possible implementation for the function `toSafeBoolean` is the following:

```
import scala.util.Try  
  
def toSafeBoolean(text: String): Boolean =  
  Try(text.toBoolean).getOrElse(false)
```

41

The Library Application

In this capstone, you will:

- Define an ordered sequence of items using `Set`
- Add and remove elements to it to track book loans
- Create book instances from a parsed CSV file using a `Map`
- Handle parsing failures using `Try`
- Validate book loans or returns by providing expressive error messages using `Either`

In this capstone, you'll create an application to keep track of books loaned in a library. Each book has a unique id, and it can only be taken by one user at a time but returned by anyone. Each user can take up to 5 books. You will need to write the code to read and parse the available books from a CSV file. In particular, you'll use a publicly accessible dataset called "goodbooks-10k" by Zajac Zygmunt. You can find its latest version as well as a detailed description of its content on its website:

The dataset contains six million ratings for ten thousand most popular books (with most ratings). There are also:

- books marked to read by the users
- book metadata (author, year, etc.)
- tags/shelves/genres

From <http://fastml.com/goodbooks-10k>

For this capstone, you'll focus on the book metadata only in the file `books.csv`.

The file provides lots of information for each record. You'll need only its title, authors, image URL, and its goodreads id: table 41.1 summarizes their features. GoodReads is an online platform for book recommendations (see <http://goodreads.com>). For simplicity, let's assume the library has

only one copy of each book so that you can use the goodreads id as a unique identifier for each of its physical copies.

Table 41.1 Summary of the book features you'll consider from the “goodbooks-10k” dataset by Zajac Zygmunt. For a full list of the analyzed characteristics, have a look at <http://fastml.com/goodbooks-10k> or the header of file books.csv.

Feature	Description	Format	Nullable
goodreads_book_id	The unique id for the book, taken from the GoodReads online platform.	Long	NO
title	Its English title	String	NO
authors	The comma-separated list of its authors	String	NO
image_url	A link to the image of its cover	String	YES

41.1 Download the base project

Let's use `git` to download the dataset together with a baseline sbt project containing the needed external dependencies and some ready to implement classes to help you get started. Navigate to an empty folder of your choice and checkout the remote branch containing the code:

```
$ git init
$ git remote add daniela https://github.com/DanielaSfregola/get-programming-with-scala.git
$ git fetch daniela
$ git checkout -b my_lesson41 daniela/baseline_unit6_lesson41
```

The files `project/build.properties` and `build.sbt` provide information on the sbt and Scala versions to use and which external dependencies to use. The `src/main/resources/books.csv` is the resource containing the dataset you'll parse to create the library's book collection. Finally, the `src/scalar` folder contains some base classes to implement:

- `org.example.books.entities.Book` provides the structure of a book.

```
case class Book(id: Long,
               title: String,
               authors: List[String],
               imageUrl: Option[URL])
```

You will implement a function in its companion object called `parse` that tries to create a `Book` instance movie from a dictionary containing fields and values.

- `org.example.books.entities.User` has an id and a full name to represent a user reserving a book from the library.

```
case class User(id: Long, fullName: String)
```

- `org.example.books.entities.BookLoan` keeps track of which user has taken a given book.

```
case class BookLoan(book: Book, user: User)
```

- `org.example.books.BookParser` uses the `scala-csv` external dependency to read a CSV file and produce an instance of type `List[Map[String, String]]` to represent the fields and values of each of its lines. You'll parse each line to create a `Book` instance tentatively.
- `org.example.books.BookService` contains the function signatures representing the business functionalities that your library must offer: you can search a book, request a book loan, and return a book. You'll implement them as part of this lesson.

Execute the command `sbt compile` to download all the external dependencies and compile the existing code. You are now ready to start coding!

41.2 Parsing a Book Entity

Let's begin by implementing a function that tentatively converts a dictionary of headers and values into a book presentation:

```
object Book {
  def parse(row: Map[String, String]): Try[Book] = ???
```

For each component of our book representation, you'll need to retrieve the value matching the correct headers and convert it into the expected type. The possible failures are that a header may be missing or that your program cannot convert it into the expected type. These failures apply to potentially any parsing scenario, so let's abstract them using higher-order functions:

Listing 41.1: Handling parsing failures

```
// file src/main/scala/org/example/books/entities/Book.scala

// [...]

object Book {

  // [...]

  private def parseAs[T](row: Map[String, String],
                        key: String, parser: String => T): Try[T] = ❶
    for {
      value <- getValue(row, key)
      t <- Try(parser(value)) ❷
    } yield t

  private def getValue(row: Map[String, String],
                      key: String): Try[String] = ❸
    row.get(key) match { ❹
      case Some(value) => Success(value)
      case None => Failure(new IllegalArgumentException(
        s"Couldn't find column $key in row - row was $row"))
    }
}
```

❶ It combines all the parsing failures

❷ Evaluating the parser function inside a Try as it may fail

- ③ It retrieves a value associated with a key, or it fails with a meaningful message
 ④ The function `get` reminds you to handle the case of missing value by returning an optional value.

You can now define functions to parse a value associated with a header into the types you need and combine them to create a `Book` instance:

Listing 41.2: Parsing a Book instance from a dictionary

```
// file src/main/scala/org/example/books/entities/Book.scala

package org.example.books.entities

import scala.util.{Failure, Success, Try}
import java.net.URL

case class Book(id: Long,
               title: String,
               authors: List[String],
               imageUrl: Option[URL]) {

  def toPrettyString: String =
    s"[${id} ${title} ${authors.mkString(", ", ", ", ")})"
}

object Book {

  def parse(row: Map[String, String]): Try[Book] = ①
    for {
      id <- parseLong(row, "goodreads_book_id")
      title <- paramString(row, "title")
      authors <- parseStrings(row, "authors")
    } yield {
      // optional fields
      val imageUrl = parseURL(row, "image_url").toOption ②
      Book(id, title, authors, imageUrl)
    }

  private def parseLong(row: Map[String, String],
                        key: String): Try[Long] =
    parseAs(row, key, _.toLong)

  private def paramString(row: Map[String, String],
                         key: String): Try[String] =
    parseAs(row, key, x => x) ③

  private def parseStrings(row: Map[String, String],
                          key: String): Try[List[String]] =
    parseAs(row, key, _.split(",").map(_.trim).toList) ④

  private def parseURL(row: Map[String, String],
                       key: String): Try[URL] =
    parseAs(row, key, s => new URL(s))

  // [...]
}
```

- ① It combines all the parsers to create a book

- ② It returns None if parsing an image URL was not possible. It is an optional field, so you can still create a book instance if not available.
- ③ A String is already your target type, so you return it as is.
- ④ It applies a map operation over an Array and converts it into a list.

You can now apply your `Book.parse` function to each line of the read CSV file and skip those unsuccessful into the `BookParser` class:

Listing 41.3: Parsing each line of the CSV file

```
// file src/main/scala/org/example/books/BookParser.scala

package org.example.books

import org.example.books.entities.Book
import com.github.tototoshi.csv._
import org.slf4j.{Logger, LoggerFactory}

import scala.io.Source
import scala.util.{Failure, Success, Try}

class BookParser(filePath: String) {

    private val logger: Logger = LoggerFactory.getLogger(this.getClass)

    val books: List[Book] = {
        loadCSVFile(filePath).flatMap { rowData => ①
            Book.parse(rowData) match {
                case Success(book) => Some(book)
                case Failure(ex) =>
                    logger.warn(s"Skipping book: Unable to parse row ②
because of ${ex.getMessage} - row was $rowData")
                    None
            }
        }
    }

    private def loadCSVFile(path: String): List[Map[String, String]] = { ③
        logger.info(s"Processing file $path...")
        val file = Source.fromResource(path)
        val reader = CSVReader.open(file)
        val data = reader.allWithHeaders()
        logger.info(
            s"Completed processing of file $path! ${data.size} records loaded")
        data
    }
}
```

- ① It transforms each dictionary with fields and values of a line of the CSV file.
- ② It logs a warning when a CSV record when finding an unparsable record
- ③ It uses the external dependency scala-csv to read the CSV file into a list of dictionaries.

41.3 The Business Logic Layer

After implementing the logic to parse the books from a file, let's see how to implement your library's business functionalities: search for, reserve, and return a book. The base structure of BookService has given you a possible signature for these functions:

Listing 41.4: The base structure of BookService

```
// file src/main/scala/org/example/books/BookService.scala

package org.example.books

import org.example.books.entities._
import org.slf4j.{Logger, LoggerFactory}

class BookService(bookCatalogPath: String) {
    private val logger: Logger = LoggerFactory.getLogger(this.getClass)

    private val books: List[Book] = new BookParser(bookCatalogPath).books ①

    private var bookLoans: Set[BookLoan] = ??? ②

    def search(title: Option[String] = None,
              author: Option[String] = None): List[Book] = ???

    def reserveBook(bookId: Long, user: User): Either[String, BookLoan] = ???

    def returnBook(bookId: Long): Either[String, BookLoan] = ???

}
```

① It loads the books for the book parser

② It tracks book loans with a private mutable assignment

41.3.1 Performing a book search

Let's implement the search by filtering on the books based on their title or any of its authors.

Have a look at listing 41.5 for its implementation:

Listing 41.5: Searching for a book by title or author

```
// file src/main/scala/org/example/books/BookService.scala

package org.example.books

import org.example.books.entities._
import org.slf4j.{Logger, LoggerFactory}

class BookService(bookCatalogPath: String) {
    private val logger: Logger = LoggerFactory.getLogger(this.getClass)

    private val books: List[Book] = new BookParser(bookCatalogPath).books

    // [...]

    def search(title: Option[String] = None,
```

```

        author: Option[String] = None): List[Book] =
    books.filter { book =>
      title.forall(t => containsCaseInsensitive(book.title, t)) &&
      author.forall(a => book.authors.exists(
containsCaseInsensitive(_, a)))
    }

  private def containsCaseInsensitive(text: String,
                                      substring: String): Boolean = ①
text.toLowerCase.contains(substring.toLowerCase)
// [...]
}

```

① Helper function to make the search case insensitive.

The function `forall` is a function defined in the `Option` class that returns `true` if it contains a value that respects the given predicate or it is empty. It is equivalent to performing a `map` operation following by a `getOrElse(true)` invocation:

```

title.forall(t => book.title.contains(t))
// ...is equivalent to...
title.map(t => book.title.contains(t)).getOrElse(true)

```

41.3.2 Reserving a book

Let's implement the functionality for a user to reserve a book. It has the following business requirements:

- A user cannot reserve more than five books at the same time.
- The requested book must exist.
- It must be available; in other words, no other user has it currently.

You can implement a function representing each check and then chain them together with for-comprehension:

Listing 41.6: Reserving a book

```

// file src/main/scala/org/example/books/BookService.scala

package org.example.books

import org.example.books.entities._
import org.slf4j.{Logger, LoggerFactory}

class BookService(bookCatalogPath: String) {
  private val logger: Logger = LoggerFactory.getLogger(this.getClass)

  private val books: List[Book] = new BookParser(bookCatalogPath).books

  private var bookLoans: Set[BookLoan] = Set.empty ①

  // [...]

  def reserveBook(bookId: Long, user: User): Either[String, BookLoan] = {
    val res = for {
      _ <- checkReserveLimits(user)

```

```

book <- checkBookExists(bookId)
    _ <- checkBookIsAvailable(book)
} yield registerBookLoan(book, user)
logger.info(s"Book $bookId - User ${user.id} - Reserve request:
${outcomeMsg(res)}") ②
res
}

private def outcomeMsg[T](res: Either[String, T]): String =
  res.left.getOrElse("OK")

private val loanLimit = 5
private def checkReserveLimits(user: User): Either[String, User] = ③
  if (bookLoans.count(_.user == user) < loanLimit) Right(user)
  else Left(
    s"You cannot loan more than $loanLimit books at the same time")

private def checkBookExists(bookId: Long): Either[String, Book] = ④
  books.find(_._id == bookId) match {
    case Some(book) => Right(book)
    case None => Left(s"Book with id $bookId not found")
  }

private def checkBookIsAvailable(book: Book): Either[String, Book] = ⑤
  findBookLoan(book) match {
    case Some(_) => Left(s"Another user has book ${book.id}")
    case None => Right(book)
  }

private def findBookLoan(book: Book): Option[BookLoan] =
  bookLoans.find(_.book == book)

private def registerBookLoan(book: Book, user: User): BookLoan = { ⑥
  val bookLoan = BookLoan(book, user)
  updateBookLoans(books => books + bookLoan)
  bookLoan
}

// [...]

private def updateBookLoans(f: Set[BookLoan] => Set[BookLoan]): Unit =
  synchronized { bookLoans = f(bookLoans) } ⑦
}

```

- ① Assuming that there are no book loans initially
- ② It logs the request and its outcome as info level
- ③ It checks if the user is within the maximum loan limit
- ④ It checks if the book exists
- ⑤ It checks if the book is available for loan
- ⑥ It registers the book loan
- ⑦ The method synchronized ensures that no simultaneous updates can happen for the mutable assignment bookLoans, causing data inconsistencies.

The function `synchronized` mimics the behavior of the Java qualifier `synchronized`: it prevents the execution of a block of code more than ones simultaneously. Imagine a scenario in which two requests invoke the function `updateBookLoans` at the same time. They both read the current assignment for `bookLoans`: suppose this is the empty set. They both produce a new instance for

`bookLoans`: let's imagine these are `Set(bookLoanA)` and `Set(bookLoanB)`, respectively. The requests will override each other's assignments: if the first one goes after the second one, `bookLoans` will contain `bookLoanA` but not `bookLoanB`, and vice versa. The method `synchronized` ensures that one request can read, modify, and reassign the mutable assignment `bookLoans` at any time, which prevents the described data inconsistency.

41.3.3 Returning a book

Finally, let's define the logic to return a book. You'll need to check the following:

- The book must exist.
- It must be out for a loan.

Its implementation reuses some of the checks you are already implemented for reserving a book:

Listing 41.7: Returning a book

```
// file src/main/scala/org/example/books/BookService.scala

package org.example.books

import org.example.books.entities._
import org.slf4j.{Logger, LoggerFactory}

class BookService(bookCatalogPath: String) {
    private val logger: Logger = LoggerFactory.getLogger(this.getClass)

    private val books: List[Book] = new BookParser(bookCatalogPath).books

    private var bookLoans: Set[BookLoan] = Set.empty

    // [...]

    def returnBook(bookId: Long): Either[String, BookLoan] = {
        val res = for {
            book <- checkBookExists(bookId)
            user <- checkBookIsTaken(book)
        } yield unregisterBookLoan(book, user)
        logger.info(s"Book $bookId - Return request: ${outcomeMsg(res)}") ①
        res
    }

    // [...]

    private def checkBookIsTaken(book: Book): Either[String, User] = ②
        findBookLoan(book) match {
            case Some(BookLoan(_, user)) => Right(user)
            case None => Left(s"Book ${book.id} does not result out on loan")
        }
    }

    private def unregisterBookLoan(book: Book, user: User): BookLoan = {
        val bookLoan = BookLoan(book, user)
        updateBookLoans(books => books - bookLoan)
        bookLoan
    }
}
```

```
// [...]
}
```

- ① It logs the request and its outcome as info level
- ② It implements a new function rather than reusing the existing one called `checkBookIsAvailable` to have a meaningful validation message.

You have now completed your library's implementation: let's see it in action in a simple scenario.

4.1.4 Let's give it a try!

Let's play a simple scenario to demonstrate how to use your library program using the sbt interactive console. Navigate to the root directory of your project and execute the `sbt console` command to compile and load your code in memory:

```
$ sbt console
[...]
scala>
```

Let's initialize your library and search for books with titles containing the words "Harry Potter":

```
scala> import org.example.books.BookService
import org.example.books.BookService

scala> val library = new BookService("books.csv")
12:19:34.671 [run-main-0] INFO org.example.books.BookParser - Processing file books.csv...
12:19:35.475 [run-main-0] INFO org.example.books.BookParser - Completed processing of file
books.csv! 10000 records loaded
library: org.example.books.BookService = org.example.books.BookService@155bee8d

scala> val books = library.search(title = Some("Harry Potter"))
books: List[org.example.books.entities.Book] = List(Book(3,Harry Potter and the Sorcerer's Stone
(Harry Potter, #1),List(J.K. Rowling...

scala> books.size
res0: Int = 22
```

Your search returned 22 books. Let's focus on the first two of them and create two users:

```
scala> val bookA = books(0)
bookA: org.example.books.entities.Book = Book(3,Harry Potter and the Sorcerer's Stone (Harry
Potter, #1),List(J.K. Rowling, Mary GrandPré),Some(https://images.gr-
assets.com/books/1474154022m/3.jpg))
// you can safely use the method apply here
// because you have verified that the collection books has 22 items

scala> val bookB = books(1)
bookB: org.example.books.entities.Book = Book(5,Harry Potter and the Prisoner of Azkaban (Harry
Potter, #3),List(J.K. Rowling, Mary GrandPré, Rufus Beck),Some(https://images.gr-
assets.com/books/1499277281m/5.jpg))
// you can safely use apply here cause you know that books has 22 items!

scala> import org.example.books.entities.User
import org.example.books.entities.User
```

```
scala> val martin = User(1, "Martin Odersky")
martin: org.example.books.entities.User = User(1,Martin Odersky)

scala> val daniela = User(2, "Daniela Sfregola")
daniela: org.example.books.entities.User = User(2,Daniela Sfregola)
```

Martin takes the first two books:

```
scala> library.reserveBook(bookA.id, martin)
12:50:58.412 [run-main-0] INFO org.example.books.BookService - Book 3 - User 1 - Reserve request:
          OK
[...]

scala> library.reserveBook(bookB.id, martin)
12:51:10.465 [run-main-0] INFO org.example.books.BookService - Book 5 - User 1 - Reserve request:
          OK
[...]
```

Then, Daniela tries to reserve a book that Martin has and a book that does not exist:

```
scala> library.reserveBook(bookA.id, daniela)
12:53:53.131 [run-main-0] INFO org.example.books.BookService - Book 3 - User 2 - Reserve request:
          Another user has book 3
[...]

scala> library.reserveBook(-1, daniela)
12:55:55.292 [run-main-0] INFO org.example.books.BookService - Book -1 - User 2 - Reserve
          request: Book with id -1 not found
[...]
```

After Martin returns the book, Daniela should be able to reserve it:

```
scala> library.returnBook(bookA.id)
12:58:21.783 [run-main-0] INFO org.example.books.BookService - Book 3 - Return request: OK
[...]

scala> library.reserveBook(bookA.id, daniela)
12:58:28.352 [run-main-0] INFO org.example.books.BookService - Book 3 - User 2 - Reserve request:
          OK
[...]
```

Try to write a new example scenario by searching for your favorite book, reserving it, and returning it. Can you spot any bug and disadvantages of your implementation?

41.5 The ugly bits of our solution

Congratulations on completing the implementation. Although it respects the requirements, it has a few aspects that you could improve: let's see what these are and what techniques will help you overcome them.

LACK OF PERSISTENT STORAGE

All your book loans are in memory: as soon as your application shuts down, it will lose them. You should consider storing them in a more permanent storage solution that you can persist and share between multiple applications. In the next unit, you'll learn how to connect to a PostgreSQL database using a library called Quill.

APPLICATION LOAD TIME

Your application parses and reads a file to populate its collection of books on startup: until this operation is complete, its users cannot perform any action with it. In this capstone, you parsed 10 thousand records in seconds, but this could take a lot longer if using a much bigger file or multiple ones. Ideally, you'd like to allow your software to be ready to use as soon as possible and perform expensive operations in the background. In the next unit, you'll learn about the type `Future` to handle asynchronous computations.

VARS ARE EVIL

You should avoid using `vars` as they make your code more complex and potentially unsafe. You had to use the `synchronized` method to ensure your program doesn't lose data when multiple requests try to update a mutable assignment simultaneously.

Unfortunately, the use of `synchronized` is not enough to guarantee that your program will always perform as expected. One of the business requirements demands a book to be loaned by one user at a time. Imagine two users request to lend the same book simultaneously. Both of them pass the checks: both users are within their loan limits, the book exists, and nobody has completed a request for it yet. They then both register a book loan: you now have the same book reserved by two users – which the business requirements explicitly forbid. Concurrency is hard, and mutability makes it even more challenging. Storing your book loans in a database can help mitigate this problem, as you can express at the database table level that the combination of user and book must be unique.

41.6 Summary

In this capstone, you have created an application to search, reserve, and return books.

- You have implemented the logic to parse a book record from a file using the classes `Try` and `Map`.
- You have represented its unordered book loans using the `Set` collection.
- You have created a function to search a book by its title or author.
- You have used the type `Either` to provide meaningful validation messages when rejecting reserve and return book requests.

Unit 7

Concurrency

In Unit 6, you have learned how to use different collections and handle errors in a more functional style. In this unit, you'll discover how to retrieve and store data in a database asynchronously. In the capstone, you'll implement the data access layer of an application to create and answer quizzes. In particular, you'll learn about the following subjects:

- Lesson 42 introduces you to `implicit`, a feature in the language that allows the compiler to enrich and expand your code at compile time. You'll also learn about the type class pattern to define more powerful abstractions.
- Lesson 43 gives you an overview of synchronous versus asynchronous computations and how they can impact your application's runtime performance. You'll learn how to represent asynchronous computations using the type `Future`.
- Lesson 44 shows you how to apply the `map`, `flatten`, and `flatMap` operations on a `Future` instance to manipulate and transform its value.
- Lesson 45 teaches you how to efficiently coordinate multiple asynchronous computations by running them in sequence or parallel.
- Lesson 46 introduces you to Quill, a popular library that provides a Domain Specific Language (DSL) to generate and run database queries. You'll use one of its asynchronous modules to connect to a PostgreSQL instance and execute queries on it.
- Finally, you'll implement the data access layer of a quiz application in lesson 47. You'll connect to its database and define the queries to run to store and retrieve categories and questions assigned to them.

After learning about asynchronous computations, you'll learn about JSON serialization and deserialization in unit 8.

42

Implicit and Type Classes

After reading this lesson, you will be able to:

- Define a function that takes implicit parameters using the keyword `using`
- Mark values as `given`
- Use the type class pattern to express ad-hoc polymorphism.

In the previous unit, you have learned about collections and error handling. In this lesson, you'll discover a feature of the Scala language called implicit. It is one of its most controversial traits. On one side, it allows you to write less code and express extremely powerful abstractions. On the other hand, it can make your program more difficult to understand, and it increases its compilation time if misused. In this lesson, rather than introducing you to all the uses that implicits have, you'll focus on its primary usage. Implicits also have a slightly different syntax between Scala 2 and Scala 3, but their core principles stay the same. You'll define a function that takes implicit parameters and mark a value as implicit in Scala 2 or as `using` in Scala 3. You'll learn how the compiler searches for a match for an implicit parameter, which is a process called implicit resolution. You'll also see how to express ad hoc polymorphism using a pattern called type class using the keywords `implicit` in Scala 2 and `given...`with in Scala 3. In the capstone, you will use implicits to define the number of threads to use when reading and writing the questions and answers to the database for your quiz application.

Consider this

Suppose you are developing an application to manage a bank account. Only bank employees with a manager role can perform critical operations, such as money withdraws over a certain amount. How would you structure your code to ensure that this business requirement is respected?

42.1 Implicit Parameters and Values

Imagine you are developing an application to place orders to deliver for a store and that you have written the following group of functions to validate and place an order:

Listing 42.1: Placing an order

```
case class User(id: Int)
case class UserContext(id: Int, details: PersonalDetails, account: Account)
case class ProductSelection(productIds: List[Int])

def purchase(userId: Int, selection: ProductSelection):
  Either[String, Int] = {
    val userContext = getUserContext(userId)
    for {
      _ <- validateAddressWithinDistance(userContext)
      _ <- validateSelection(selection, userContext)
      _ <- validateBalance(selection, userContext)
    } yield placeOrder(selection, userContext)
  }

private def getUserContext(userId: Int): UserContext = ??? ①

private def validateBalance(
  selection: ProductSelection, userContext: UserContext):
  Either[String, Double] = ??? ②

private def validateAddressWithinDistance(
  userContext: UserContext): Either[String, UserContext] = ??? ③

private def validateSelection(
  selection: ProductSelection, userContext: UserContext):
  Either[String, ProductSelection] = ??? ④

private def placeOrder(
  selection: ProductSelection, userContext: UserContext):
  Int = ??? ⑤
```

- ① It retrieves all the user information
- ② It checks the user balance is enough to buy the selected product
- ③ It ensures the user address is
- ④ It verifies that the selected products are available and age-appropriate for the user
- ⑤ It returns the id of the placed order

In Scala, functions can have function parameters split into different groups. You decide to change your code so that the user context is on a different parameter group than the product selection where applicable. Listing 42.2 shows you how the code looks like after your refactoring:

Listing 42.2: Placing an order using due groups of parameters

```
case class User(id: Int)
case class UserContext(id: Int, details: PersonalDetails, account: Account)
case class ProductSelection(productIds: List[Int])

def purchase(userId: Int, selection: ProductSelection):
  Either[String, Int] = {
    val userContext = getUserContext(userId)
```

```

for {
  _ <- validateAddressWithinDistance(userContext)
  _ <- validateSelection(selection)(userContext)
  _ <- validateBalance(selection)(userContext)
} yield placeOrder(selection)(userContext)
}

private def getUserId(userId: Int): UserContext = ???

private def validateBalance(selection: ProductSelection)
  (userContext: UserContext):
    Either[String, Double] = ???

private def validateAddressWithinDistance(
  userContext: UserContext): Either[String, UserContext] = ???

private def validateSelection(selection: ProductSelection)
  (userContext: UserContext):
    Either[String, ProductSelection] = ???

private def placeOrder(selection: ProductSelection)
  (userContext: UserContext): Int = ???

```

All the functions you specified rely on the presence of an instance of `UserContext` to provide information about the user placing the order. You could avoid this visual repetition when calling each function by declaring it as an implicit parameter:

Listing 42.3: Placing an order using implicit

```

case class User(id: Int)
case class UserContext(id: Int, details: PersonalDetails, account: Account)
case class ProductSelection(productIds: List[Int])

def purchase(userId: Int, selection: ProductSelection):
Either[String, Int] = {
  // In Scala 2: implicit val userContext = getUserContext(userId) ①
  given UserContext: UserContext = getUserContext(userId) ②
  for {
    _ <- validateAddressWithinDistance ③
    _ <- validateSelection(selection) ③
    _ <- validateBalance(selection) ③
  } yield placeOrder(selection)
}

private def getUserId(userId: Int): UserContext = ???

private def validateBalance(selection: ProductSelection)
  (using userContext: UserContext): ④
    Either[String, Double] = ???

private def validateAddressWithinDistance(
  using userContext: UserContext): ④
Either[String, UserContext] = ???

private def validateSelection(selection: ProductSelection)
  (using userContext: UserContext): ④
    Either[String, ProductSelection] = ???

```

```
private def placeOrder(selection: ProductSelection)
    (using userContext: UserContext): Int = ??? ④
```

- ① Declaring a value as implicit in Scala 2
- ② Declaring a value as implicit in Scala 3
- ③ Invoking the function without explicitly providing a user context parameter
- ④ Declaring a function parameter as implicit in Scala 3. In Scala 2, use the keyword `implicit` instead of the keyword `using`.

When defining a function, you can declare its last groups of parameters as implicit using the keyword `using` in Scala 3 or the keyword `implicit` in Scala 2. You can either specify or omit its implicit parameters. If missing, the compiler will try to fill the gaps for you and find a suitable match at compile time to pass to your function. It will search into the elements marked as `given` or `implicit` if using Scala 2, within your function's scope. If it finds no unique and unambiguous match, the compiler will fail with a compilation error.

Let's see a simple example in action:

```
// In Scala 2: def welcome(name: String)(implicit msg: String): Unit
scala> def welcome(name: String)(using msg: String): Unit =
|   println(s"$msg, $name!")
welcome: (name: String)(using msg: String)Unit
// the function welcome requires an implicit parameter of type String

// In Scala 2: welcome("Jane")("Hello")
scala> welcome("Jane")(using "Hello")
Hello, Jane!
// You need to provide the keyword using when passing an implicit parameter
// explicitly in Scala 3. You do not need to pass any keyword in Scala 2.

scala> welcome("Jane")
<console>:13: no implicit argument of type String was found for parameter msg of method welcome
          ^
// You didn't provide a value for msg, so the compiler searches one to use.
// It fails to find one (there are no implicit elements!)

scala> val hi = "Hi"
hi: String = Hello

scala> welcome("Jane")
<console>:13: no implicit argument of type String was found for parameter msg of method welcome
          ^
// The compiler does not detect hi as a valid match
// because it is not marked as given (or implicit in Scala 2).

// In Scala 2: implicit val hi2 = "Hi"
scala> given hi2: String = "Hi"
lazy val hi2: String = Hi

scala> welcome("Jane")
Hi, Jane!
// The compiler passes the value hi2 implicitly

// In Scala 2: implicit val hola = "Hola"
scala> given hola: String = "Hola"
lazy val hola: String = Hola
scala> welcome("Jane")
<console>:15: error: ambiguous implicit arguments: both method hola and method hi match type
          String of parameter msg of method welcome
```

```
// The compiler finds two values of type String  
// and it cannot pick one unambiguously.
```

Figures 42.1 and 42.2 summarize how to declare implicit parameters and values in Scala 2 and 3.

Scala 2 - Implicit Value

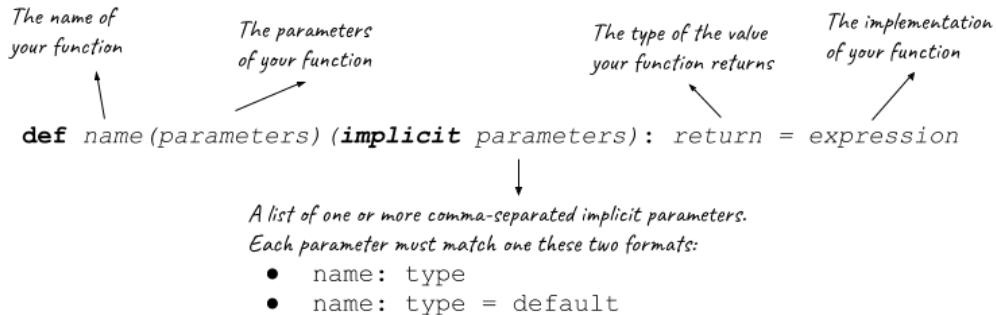
the name of your implicit value *an assignment for your implicit value*
implicit val *name: type = expression*
 ↓
the type of your implicit value

Scala 3 - Implicit Value

the name of your implicit value *an assignment for your implicit value*
given *name: type = expression*
 ↓
the type of your implicit value

Figure 42.1: Syntax summary of the implicit values in Scala 2 and Scala 3.

Scala 2 - Implicit Parameters



Scala 3 - Implicit Parameters

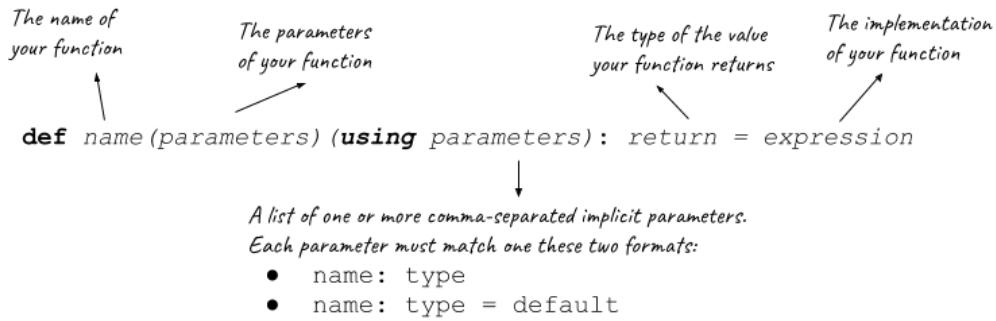


Figure 42.2: Syntax summary of how to define implicit parameters in Scala 2 and Scala 3.

QUICK CHECK 42.1

Consider the following snippet of code. Does it compile? If not, how would you fix it?

```
def plusOne(using n: Int): Int = n + 1
plusOne(using 3)
```

42.2 Implicit Resolution

When invoking a function, you can omit its implicit parameters. The compiler will search for suitable matches to automatically pass for you at compile time: this process is called *implicit resolution*. Let's outline how it performs its search.

First, the compiler looks at the *current scope*:

- Your function's local scope (i.e., the portion of code accessible without imports)

```
// In Scala 2: def pow(exp: Int)(implicit base: Int): Double = ...
```

```
def pow(exp: Int)(using base: Int): Double = Math.pow(base, exp)

    // In Scala 2: implicit val a: Int = ...
given b: Int = 2 // implicit found in the local scope[1]

pow(5)
```

- the code you have imported.

```
// In Scala 2: def pow(exp: Int)(implicit base: Int): Double = ...
def pow(exp: Int)(using base: Int): Double = Math.pow(base, exp)[1][2][3]

object Base {

    // In Scala 2: implicit val b: Int = 2
    given b: Int = 2
}

// In Scala 2: import Base._
import Base.given // importing all the implicit instances in Base

// implicit found in the imported code[1][2][3]
pow(5)
```

If it finds no match, it searches into the *associated types*:

- The companion object of the type marked as `given` (or `implicit` in Scala 2).

```
trait Name[A] {
    def name(): String
}

object Name {

    // implicit selected
    // In Scala 2: implicit val intName: Name[Int] = new Name[Int] { ... }
    given intName: Name[Int] with {
        def name() = "integer"
    }
}

// In Scala 2: def describe[T](implicit t: Name[T]): String = ...
def describe[T](using t: Name[T]): String = t.name()

describe[Int]
```

The compiler cannot find an implicit instance of `Name[Int]` into the current scope. It then looks at the companion object `Name`, and it selects the value `intName`.

- The companion object of any type parameter of the type marked as `given`, or `implicit` in Scala 2.

```
trait Name[A] {
    def name(): String
}
```

```

class Test()
object Test {

    // implicit selected[T]
    // In Scala 2: implicit val name: Name[Test] = new Name[Test] {...}
    given name: Name[Test] with {
        override def name(): String = "my-test"
    }
}

// In Scala 2: def describe[T](implicit t: Name[T]): String = ...
def describe[T](using t: Name[T]): String = t.name()

describe[Test]

```

The compiler fails to find any implicit value in the current scope. Then, it finds a valid match of type `Name[Test]` in the companion object of the class `Test`.

Figure 42.3 provides a summary of how the compiler searches implicit matches to use in your code.

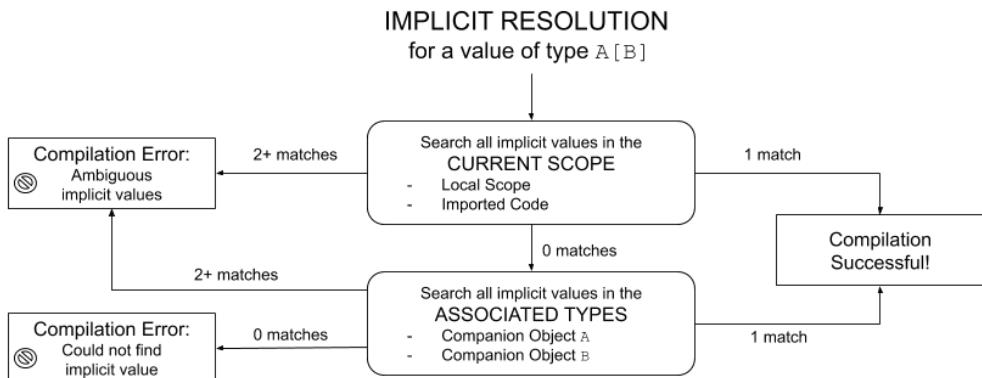


Figure 42.3: The implicit resolution algorithm that the compiler uses to find an implicit match of type `A[B]`. First, it looks at the current scope. Then it searches into the companion objects of the associated types `A` and `B`. If it finds one match, it uses the code element as a parameter. Otherwise, the compiler errors with either an implicit not found or an ambiguous implicit values error.

QUICK CHECK 42.2

Consider the following snippet of code. Does it compile? If not, how would you fix it?

```

class A {
  def test(using n: Int): String = n.toString
}

object A {
  given n: Int = 2
}

(new A()).test

```

42.3 Type Classes

Let's consider your application to deliver store orders again. Imagine you have a requirement to consistently sort your orders across all your program's functionalities by their id in reverse order. Listing 42.4 shows you how to achieve this:

Listing 42.4: Sorting by id in inverse order

```

case class Order(id: Int)

object Order {
// In Scala 2:
// implicit val ordering: Ordering[Order] = new Ordering[Order] { ... }
  given ordering: Ordering[Order] with { ①
    override def compare(x: Order, y: Order): Int =
      - x.id.compare(y.id) ②
  }
}

```

① Ordering is the interface that Scala requires you to implement to define sorting rules.

② The minus symbol expresses the inverse order.

After providing a default implicit implementation for `Ordering[Order]`, you can now use it in your code. For example, you have learned that the `List` collection has a few methods, such as `sorted`, `min`, `max`, that use it to define how to compare instances of the same type. Another example you have encountered is the trait `Numeric[T]`, which is used by its method `sum`.

Scala offers *ad hoc polymorphism*. Polymorphism is the concept of having an interface that defines the form of many types. It is *ad hoc* because you can override the behavior by providing an implicit in your current scope rather than using the one you defined in the associated types. This behavior is possible because the compiler only searches into the associated types if it finds no matches in the current scope. This pattern is called *type class*, and it is popular when coding in a functional programming style.

Suppose you need to ensure that any information your program displays to a user must be human-readable. The function `toString` is not fit for purpose:

```

scala> class A; (new A()).toString
defined class A
res0: String = A@55b7f0d
// Incomprehensible for a human!

scala> List(1, 2, 3).toString
res1: String = List(1, 2, 3)
// Humans may find "1, 2, 3" simpler to read

```

Let's define a trait with a method `show` to produce a human-readable text from a given instance and provide a few default implementations for some of the basic types:

Listing 42.5: The trait Show

```
trait Show[T] {

    def show(t: T): String
}

object Show {

    // In Scala 2:
    // implicit val stringShow: Show[String] = new Show[String] { ... }
    given stringShow as Show[String] with {
        override def show(s: String): String = s
    }

    // In Scala 2:
    // implicit val intShow: Show[Int] = new Show[Int] { ... }
    given intShow: Show[Int] with {
        override def show(i: Int): String = i.toString
    }

    // In Scala 2:
    // implicit def listShow[T]: Show[List[T]] = new Show[List[T]] { ... }
    given listShow[T]: Show[List[T]] with {
        override def show(l: List[T]): String = l.mkString(", ")
    }
}
```

You can now define a function `prettyPrintln` and invoke it every time you need to display data to the user:

```
// In Scala 2: def prettyPrintln[T](t: T)(implicit s: Show[T]): Unit = ...
def prettyPrintln[T](t: T)(using s: Show[T]): Unit =
    println(s.show(t))

prettyPrintln(1) // prints "1"
prettyPrintln("hello") // prints "hello"
prettyPrintln(List(1, 2, 3)) // prints "1, 2, 3"
```

The trait `Show` is part of an external Typelevel library called `cats`. It offers a collection of useful type classes: a full list is available in their documentation at <https://typelevel.org/cats>.

QUICK CHECK 42.3

Consider the following class `Person`:

```
case class Person(name: String, age: Int)
```

Define a default implementation for `Show[Person]` so that your function `prettyPrintln` can print the person's name followed by its age. For example, the expression `prettyPrintln(Person("Jon Doe", 25))` should print "Jon Doe (25)" to the console.

42.4 Summary

In this lesson, my objective was to teach you the basics of implicits.

- You have seen how to define implicit parameters and values.
- You have discovered the implicit resolution algorithm that the compiler uses to find a valid implicit match.
- You have learned how to use the type class pattern to express ad hoc polymorphism.

Let's see if you got this!

TRY THIS

Suppose your program needs to perform monetary calculations. Your money representation looks like the following:

```
import java.util.Currency
case class Money(amount: Double, currency: Currency)
```

Define an instance of `Numeric[Money]` so that Scala can recognize it as a numeric value and perform calculations on them, such as summing two `Money` instances. Ensure to forbid operations between monetary amounts that have different currencies.

42.5 Answers to Quick Checks

QUICK CHECK 42.1

The code compiles, and it returns 4. The compiler doesn't search for an implicit parameter because you provided a value for it explicitly.

QUICK CHECK 42.2

The compiler fails in finding an implicit value for the parameter `n`:

```
<console>:13: no implicit argument of type Int was found for parameter n of method test in class
      A
      |
      |The following import might fix the problem:
      |
      |  import A.n
```

First, it searches in the current scope: this is the code accessible from `(new A()).test` and from `def test(using n: Int): String`. It fails to find a good match, so it looks at the companion object of `Int`, which has no accessible implicit value of type `Int`. The compiler stops the implicit resolution, and it fails the compilation. You need to include the implicit value `A.n` into the current scope using an import to fix the compilation error:

```
// In Scala 2: import A._

import A.given
(new A()).test
```

Alternatively, you can also do the following:

```
class A {  
    // In Scala 2: import A._  
    import A.given  
  
    // In Scala 2: def test(implicit n: Int): String = ...  
    def test(using n: Int): String = n.toString  
}
```

QUICK CHECK 42.3

Your implementation of `Show[Person]` should live in the companion object of `Person`, and it should look similar to the following:

```
object Person {  
  
    // In Scala 2: implicit val show: Show[Person] = new Show[Person] { ... }  
    given show: Show[Person] with {  
        override def show(p: Person): String = s"${p.name} (${p.age})"  
    }  
}
```

43

Future

After reading this lesson, you will be able to:

- Represent an asynchronous computation using Future
- Process its result on completion

In the previous lesson, you have learned about implicit parameters and values: you'll now see them in action when defining an instance of type Future. You'll discover the difference between synchronous and asynchronous computations and how they can affect your program's performance. You'll see how you can use the type Future to define asynchronous computations that can fail. An operation is asynchronous if your program can continue to run without waiting for its outcome. You'll also process its result once completed. In the capstone, you'll use the type Future to read and write data to a database asynchronously for your quiz application.

Consider this

Imagine you are developing an application to book tickets for events. It needs to accept and process as many booking requests as possible at the same time. How would you structure your application so that it can maximize its available resources?

43.1 Why Future?

Latency is the time spent between requesting an operation and getting a response back. Brendan Gregg discusses how system operations can differ in duration in his book *Systems Performance: Enterprise and the Could* (Prentice Hall, 2014). For example, a 3.3 GHz processor can execute on average one CPU cycle in 0.3 nanoseconds (ns) and a rotational hard disk I/O operation in up to 10 milliseconds (ms). As humans, we struggle to grasp how different these numbers are, so he decided to scale these and other system events to a more understandable scale: you can see them

in table 43.1. If you scale one CPU cycle up to 1 second, a single I/O operation can last up to 12 months!

Table 43.1: Example Time Scale of System Latencies, taken from Brendan Gregg's *Systems Performance: Enterprise and the Cloud* (Prentice Hall, 2014), page 20.

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50-150 µs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1-3 s	105-317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

Programs traditionally perform operations synchronously: they wait for an instruction to finish before starting with the next one. This approach is a popular strategy, but it has a considerable performance cost when dealing with procedures with significant latency. Another option is to perform operations asynchronously: starting the next instruction without waiting for the current one to be complete. This approach can make your program do more with its resources as it will not spend as much time waiting for operations to complete, but its execution flow will be more difficult to predict. Your CPU can split a program into smaller independent sequences of instructions called

threads of executions, or just threads. A multithreaded platform, such as the JVM, allows your program to run more than one thread simultaneously, making asynchronous computations possible.

The type `Future` allows you to define an operation to execute asynchronously: the JVM delegates its execution to another thread, it moves on to other instructions, and it returns to it when notified of its completion.

43.2 Creating an instance of Future

Imagine you are developing a program to handle orders for a store. You need to confirm a product's availability with its warehouse via an HTTP API, a known bottleneck of your application. You can perform it asynchronously using `Future` to improve its overall performance:

Listing 43.1: Checking product availability

```
import scala.concurrent.Future ①
import scala.concurrent.ExecutionContext.Implicits.global ②

def isProductAvailable(productId: Int,
                      quantity: Double): Future[Boolean] = Future { ③
    requestAvailability(productId, quantity)
}

private def requestAvailability(productId: Int,
                               quantity: Double): Boolean = ???
```

① Importing the class `Future`

② Adding the implicit Global Execution Context to your scope. It contains information about the resources available to your program

③ Calling the function `Future.apply`

In Scala, the class `Future` allows you to specify an operation to execute asynchronously: a separate thread eventually completes its execution while your program's current thread processes other instructions. The method `Future.apply` allows you to create an instance of `Future`: listing 43.2 shows you its signature.

Listing 43.2: The method Future.apply

```
package scala.concurrent

object Future {

    def apply[T](r: => T) ①
        (using ec: ExecutionContext): Future[T] = { ②
            ??? ③
        }
}
```

① The parameter is not evaluated until needed

② The `ExecutionContext` defines the available set of threads to use

③ Implementation omitted as non-trivial

First, include the companion object Future into your scope by adding `import scala.concurrent.Future`. Then, invoke its method `Future.apply` by providing the instructions to execute asynchronously and an execution context. Figure 43.1 compares the signature of `Try.apply` with `Future.apply`: the two methods look very similar. However, `Future.apply` requires an extra implicit parameter of type `ExecutionContext`, which provides information that makes the asynchronous computation possible.

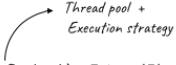
SYNCHRONOUS computation that can fail	versus	ASYNCHRONOUS computation that can fail
<pre>package scala.util object Try { def apply[T](r: => T): Try[T] = ??? }</pre>		<pre>package scala.concurrent object Future { def apply[T](r: => T) (using ec: ExecutionContext): Future[T] = ??? }</pre> 

Figure 43.1: A comparison between the `apply` method for `Try` and `Future`: the first represents synchronous computations that can fail, the second asynchronous ones. They both delay the execution of an expression. However, the `Future` one requires an instance of `ExecutionContext`, which contains information about the resources and execution strategy.

The `ExecutionContext` class gives your `Future` instance a thread pool to use and an execution strategy to follow. Scala offers a default implementation for it called `ExecutionContext.global`, which is designed to work for the majority of the cases. You can define a custom execution context, but it goes beyond this book's scope: this is non-trivial and can cause significant performance degradation to your application if not done carefully. You can include `ExecutionContext.global` as implicit into your scope by adding the following instruction:

```
import scala.concurrent.ExecutionContext.Implicits.global
```

This expression is equivalent to defining the following implicit value:

```
import scala.concurrent.ExecutionContext
using val ec = ExecutionContext.global
```

ExecutionContext as an implicit parameter rather than as an import

When developing a large application, you should prefer passing your `ExecutionContext` as an implicit parameter to your classes and functions rather than directly using the global execution context. You could then select which execution context to use in one place of your codebase, usually its entry point. This approach gives you more control over which execution context your program uses and easier to customize if needed.

Let's see a few more examples of how you can use the method `Future.apply`:

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> Future(10/2)
```

```

<console>:13: error: Cannot find an implicit ExecutionContext. You might pass
an (using ec: ExecutionContext) parameter to your method
or import scala.concurrent.ExecutionContext.Implicits.global.
    Future(10/2)
           ^
// you need to provide an ExecutionContext to Future.apply!

scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> Future(10/2)
res0: scala.concurrent.Future[Int] = Future(Success(5))
// Its execution has completed with success
// before the REPL displays its content

scala> val tenOverZero = Future(10/0)
tenOverZero: scala.concurrent.Future[Int] = Future(<not completed>)
// This time, its execution has not completed yet

scala> tenOverZero res1
tenOverZero: scala.concurrent.Future[Int] = Future(Failure(java.lang.ArithmeticException: / by
zero))
// Checking its content again after some time,
// it has completed with a failure

scala> val foo = {
    |   println("Hello")
    |   Future(println("World"))
    |   println("!")
    | }
Hello
!
World
val foo: Unit = ()
// Your program does not wait for the Future instance to complete
// and it moves on to print the text "!"

```

QUICK CHECK 43.1

Consider the following function: does it compile? What output does it produce when invoked?

```

import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def test(): Unit = {
  Future(('a' to 'c').map(print))
  (0 to 2).map(print)
}

```

43.3 Processing Future on completion

Consider your program to place orders in a store. Suppose that the third party API can provide information on the quantity available for a certain product: you want to keep track of it to help the store decide on the items to restock. Listing 43.3 shows you how to do this with the `onComplete` method:

Listing 43.3: Tracking the availability of a product

```
import scala.concurrent.Future
import scala.util.{Failure, Success}
import scala.concurrent.ExecutionContext.Implicits.global

case class Availability(id: Int, quantity: Double)

def trackAvailability(availability: Future[Availability]): Unit =
  availability.onComplete { ①
    case Success(p) if p.quantity <= 0 =>
      println(s"Product ${p.id} is not available")
    case Success(p) =>
      println(s"Product ${p.id} has available quantity ${p.quantity}")
    case Failure(ex) =>
      println(s"Couldn't get the availability because of ${ex.getMessage}")
  }
```

① It processes the result of a completed future as an instance of Try

You can use pattern matching on many of the types you have encountered: unfortunately, `Future` is not one of them. A pattern matching operation is synchronous as it requires the value wrapped inside your type to be available. For an instance of `Future[T]`, the method `onComplete` allows you to execute a function once completed:

```
def onComplete[U](f: Try[T] => U)
  (using executor: ExecutionContext): Unit
```

The function `onComplete` returns a value of type `Unit`: any result that your callback function `f` produces will be lost. If you need to retain its result, you should consider using its method `map` instead: you'll learn more about this in the next lesson. A few more examples of how to use the method `onComplete` are the following:

```
scala> Future(10/2).onComplete { value =>
  |   if (value.isFailure) println("bad!")
  |   else println("good!")
  | }
good!
// it prints "good!" to the console

scala> Future(10/2).onComplete { value =>
  |   if (value.isFailure) "bad!"
  |   else "good!"
  | }
// it produce no value because it discards the string value
```

QUICK CHECK 43.2

Consider the following function: does it compile? What output does it produce when invoked?

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future

def isSuccess[T](f: Future[T]): Unit = f.onComplete(_.isSuccess)
```

43.4 Summary

In this lesson, my objective was to introduce you to asynchronous computation using the type `Future`.

- You have learned that an asynchronous computation allows your program to execute other instructions without waiting for it to complete first.
- You have seen to create an instance of type `Future` using the global execution context.
- You have also seen how to use the method `onComplete` to process a value provided by an asynchronous computation.

Let's see if you got this!

TRY THIS

The following snippet of code prints all the files in the current directory to the terminal: change its code to execute it asynchronously.

```
import java.io.File
new File(".").listFiles().foreach(println)
```

43.5 Answers to Quick Checks

QUICK CHECK 43.1

The function `test` compiles: an instance of `ExecutionContext.global` is available through the import of `scala.concurrent.ExecutionContext.Implicits.global`. Its output is non-deterministic: it changes based on when how the executor schedules its threads.

```
scala> import scala.concurrent.Future
|   import scala.concurrent.ExecutionContext.Implicits.global
|
|   def test(): Unit = {
|     Future((`a` to `c`).map(print))
|     (0 to 2).map(print)
|   }
|
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
test: ()Unit

scala> test()
abc012
scala> test()
0abc12
scala> test()
012abc
```

QUICK CHECK 43.2

The function `isSuccess` compiles, but it returns no value: when your `Future` instance completes, the callback produces a boolean value that the method `onComplete` then discards.

```
scala> import scala.concurrent.ExecutionContext.Implicits.global
|   import scala.concurrent.Future
```

```
| def isSuccess[T](f: Future[T]): Unit = f.onComplete(_.isSuccess)
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
isSuccess: [T](f: scala.concurrent.Future[T])Unit
scala> isSuccess(Future("hello"))
scala> isSuccess(Future(throw new Exception("BOOM!")))
```

44

Working with Future: map and flatMap

After reading this lesson, you will be able to:

- Manipulate the result of an asynchronous computation using the `map` operation.
- Merge two nested asynchronous computations using the `flatten` method.
- Combine multiple asynchronous operations using the `flatMap` function.

In the previous lesson, you have learned the basics of expressing asynchronous computations using the type `Future`. In this lesson, you'll learn how to use the methods `map`, `flatten`, and `flatMap` for an instance of `Future`. You'll notice that they share many commonalities with the `map`, `flatten`, and `flatMap` methods you have mastered for other types. The `map` function allows you to transform the value that an asynchronous computation produces. The `flatten` method merges two nested instances of `Future` into one. The `flatMap` operation is the composition of the methods `map` and `flatten`, and it allows you to chain multiple asynchronous instances. In the capstone, you'll need to coordinate several asynchronous calls to read or write questions to a database for your quiz application.

Consider this

Imagine that your application to book tickets for events performs an asynchronous computation to produce a registration receipt. After its completion, you'd like to show the user a message to confirm its registration number and provide more details about the event. How would you achieve this?

44.1 The map, flatten, and flatMap operations

The type `Future` has an implementation for the `map`, `flatten`, and `flatMap` methods: you'll see them in action in the following subsections. You will notice that they have many similarities with those you are already encountered, such as `Option`, `List`, and `Try`.

44.1.1 The map function

Let's consider again your program to place orders in a store. Suppose that after checking for a product's availability, you'd like to either place an order or reject the request. Listing 44.1 shows you a possible way of achieving this:

Listing 44.1: Placing an order if the product is available

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext

case class Availability(id: Int, quantity: Double)
case class Order(id: Int, customerId: Int,
    productId: Int, quantity: Double)

private def getAvailability(productId: Int)
(using ec: ExecutionContext):
    Future[Availability] = ???

private def createOrder(customerId: Int,
    productId: Int,
    quantity: Double): Order = ???

def placeOrder(customerId: Int,
    productId: Int,
    quantity: Double)
(using ec: ExecutionContext): Future[Order] = {
    getAvailability(productId).map { availability => ①
        if (quantity <= availability.quantity)
            createOrder(customerId = customerId, productId = productId, quantity)
        else throw new IllegalStateException(②
            s"Product $productId unavailable:
requested $quantity, available ${availability.quantity}")
    }
}
```

① It accesses the value wrapped into a `Future` instance, and it applies a function to it.

② `Future` catches any exception: you can throw them knowing that `Future` will contain them.

When working with an asynchronous computation, you can use its method `map` to transform its produced result. For an instance of `Future[T]`, the function `map` takes one parameter `f` of type `T => S` and an implicit execution context to produce an instance of type `Future[S]`. It has the following signature:

```
def map[S](f: T => S)(using ec: ExecutionContext): Future[S]
```

If your `Future[T]` instance has completed successfully, it will apply the parameter `f` to its result to produce a value of type `Future[S]`. Nothing happens if your `Future[T]` instance has completed with a failure. A few examples of how to use it are the following:

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> Future(12/2).map(_ * 3)
res0: scala.concurrent.Future[Int] = Future(Success(18))

scala> Future(12/0).map(_ * 3)
res1: scala.concurrent.Future[Int] = Future(Failure(java.lang.ArithmeticException: / by zero))

scala> Future(12/2).map { n =>
    |   if (n > 10) n
    |   else throw new Exception(s"Too small: $n")
    |
}
res2: scala.concurrent.Future[Int] = Future(Failure(
java.lang.Exception: Too small: 6))
```

QUICK CHECK 44.1

Define a function called `toInt` to parse a value of type `Future[String]` into one of `Future[Int]`. Make sure to provide an execution context as an implicit parameter, rather than importing one directly.

44.1.2 The flatten function

Consider the function you have written in listing 44.1 to check the availability for a product and create an order: its function `createOrder` returns a value of type `Order`. Imagine the function `createOrder` now needs to write to a database asynchronously, and that you need to change its return type from `Order` to `Future[Order]`: this causes its function `placeOrder` to return a value of type `Future[Future[Order]]`:

Listing 44.2: Placing an order by writing to a database

```
private def getAvailability(productId: Int)
    (using ec: ExecutionContext):
Future[Availability] = ???

private def createOrder(customerId: Int,
    productId: Int,
    quantity: Double)
    (using ec: ExecutionContext): Future[Order] = ???

def placeOrder(customerId: Int,
    productId: Int,
    quantity: Double)
    (using ec: ExecutionContext): Future[Future[Order]] = {
    getAvailability(productId).map { availability =>
        if (quantity <= availability.quantity)
            createOrder(customerId = customerId, productId = productId, quantity)}
```

```
    else throw new IllegalStateException(
      s"Product $productId unavailable:
requested $quantity, available ${availability.quantity}")
  }
}
```

The type `Future[Future[Order]]` represents two nested asynchronous computations that will eventually either return an instance of type `Order` or fail. The function `flatten` can simplify this expression by considering the two nested operations as one: it will now produce a value of type `Future[Order]` instead.

Listing 44.3: Placing an order by writing to a database using flatten

```
private def getAvailability(productId: Int)
    (using ec: ExecutionContext):
Future[Availability] = ???

private def createOrder(customerId: Int,
                      productId: Int,
                      quantity: Double)
    (using ec: ExecutionContext): Future[Order] = ???

def placeOrder(customerId: Int,
               productId: Int,
               quantity: Double)
    (using ec: ExecutionContext): Future[Order] = {
  getAvailability(productId).map { availability =>
    if (quantity <= availability.quantity)
      createOrder(customerId = customerId, productId = productId, quantity)
    else throw new IllegalStateException(
      s"Product $productId unavailable:
requested $quantity, available ${availability.quantity}")
  }.flatten
}
```

The method `flattenOnFuture` allows you to transform an instance of `Future[Future[T]]` into one of type `Future[T]`. A few examples are following:

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> val twelveOverZero = Future(Future(12/0)).flatten
twelveOverZero: scala.concurrent.Future[Int] = Future(<not completed>)

scala> twelveOverZero
twelveOverZero: scala.concurrent.Future[Int] = Future(Failure(java.lang.ArithmetricException: / by
    zero))

scala> Future(Future(12/2)).flatten
res0: scala.concurrent.Future[Int] = Future(Success(6))

scala> Future(5).flatten
<console>:14: error: Cannot prove that Int <::> scala.concurrent.Future[S].
      Future(5).flatten
```

```
// you can only invoke the method flatten on nested structured
```

QUICK CHECK 44.2

Consider the following snippet of code:

```
import scala.concurrent.{ExecutionContext, Future}

case class Account(id: String)
case class User(name: String)

def getAccount(orderId: Int)
  (using ec: ExecutionContext): Future[Account] = ???

def getUser(accountId: String)
  (using ec: ExecutionContext): Future[User] = ???
```

Use the functions `getAccount` and `getUser` to create a new function that will return the user associated with a given order id. This function should have the following signature:

```
def getUser(orderId: Int)(using ec: ExecutionContext): Future[User]
```

44.1.3 The flatMap function

Consider the snippet of code you have written in listing 44.3. A more elegant way of achieving the same is the following:

Listing 44.4: Placing an order by writing to a database using flatMap

```
private def getAvailability(productId: Int)
  (using ec: ExecutionContext):
Future[Availability] = ???

private def createOrder(customerId: Int,
                      productId: Int,
                      quantity: Double)
  (using ec: ExecutionContext): Future[Order] = ???

def placeOrder(customerId: Int,
               productId: Int,
               quantity: Double)
  (using ec: ExecutionContext): Future[Order] = {
  getAvailability(productId).flatMap { availability =>
    if (quantity <= availability.quantity)
      createOrder(customerId = customerId, productId = productId, quantity)
    else throw new IllegalStateException(
      s"Product $productId unavailable:
requested $quantity, available ${availability.quantity}")
  }
}
```

The method `flatMap` is the combination of the `map` and `flatten` operations. For an instance of `Future[T]`, the function `flatMap` takes one parameter `f` of type `T => Future[S]` and an implicit execution context to produce an instance of type `Future[S]`. It has the following signature:

```
def flatMap[S](f: T => Future[S])(using ec: ExecutionContext): Future[S]
```

If your instance of `Future[T]` has completed successfully, it will apply the parameter `f` to produce a value of type `Future[S]`. Nothing happens if your instance has completed with a failure. A few examples of how to use it are the following:

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> val twelveOverTwo = Future(12/2).flatMap(n => Future(n.toString))
twelveOverTwo: scala.concurrent.Future[String] = Future(<not completed>)
// twelveOverTwo has not completed yet - let give it another try

scala> twelveOverTwo
twelveOverTwo: scala.concurrent.Future[String] = Future(Success(6))
// twelveOverTwo has now completed successfully

scala> Future(12/0).flatMap(n => Future(n.toString))
res0: scala.concurrent.Future[String] = Future(Failure(java.lang.ArithmetricException: / by zero))
```

The `flatMap` method allows you to express an execution dependency asynchronous computations. For example, the `placeOrder` function you have implemented in listing 44.4 defines that the product availability check must complete successfully before creating an order. You will learn more about this in the next lesson in which you will master how to use for-comprehension on instances of type `Future`.

QUICK CHECK 44.3

In quick check 44.2, you have implemented a function `getUser(orderId: Int)` using the function `flatten`: refactor it to use the `flatMap` method instead.

Table 44.1 summarizes the signature and usage of the methods `map`, `flatten`, and `flatMap` acting on `Future`.

Table 44.1: Technical recap of the three fundamental operations on the type Future. The function `map` transforms the result of an asynchronous computation, while `flatten` merges two executions. The `flatMap` function combines the `map` and `flatten` operations to define an execution order between values.

	Acts on	Signature	Usage
<code>map</code>	<code>Future[T]</code>	<code>map(f: T => S)(using ec: ExecutionContext): Future[S]</code>	It applies a function to the value the future produced.
<code>flatten</code>	<code>Future[Future[T]]</code>	<code>flatten: Future[T]</code>	It merges two nested futures into one.
<code>flatMap</code>	<code>Future[T]</code>	<code>flatMap(f: T => S)(using ec: ExecutionContext): Future[S]</code>	The combination of <code>map</code> followed by <code>flatten</code> . It chains futures together.

44.2 Summary

In this lesson, my objective was to teach you the fundamental operations you can perform on an instance of type `Future`.

- You have seen how to use the function `map` to transform the value your asynchronous computation produces.
- You have learned how to merge two nested instances of `Future` into one using the `flatten` operation.
- You have discovered how the `flatMap` method allows you to express an execution dependency between two asynchronous computations.

Let's see if you got this!

TRY THIS

Consider the following snippet of code that defines a function to list the content in a given directory:

```
import java.io.File
import scala.concurrent.{ExecutionContext, Future}

def contentInDir(path: String)
    (using ec: ExecutionContext): Future[List[String]] =
  Future {
  val file = new File(path)
  if (file.isDirectory)
    // unfortunately, listFiles returns null
```

```
// if invoked on a file that is not directory
    file.listFiles().toList.map(_.getAbsolutePath)
  else List.empty
}
```

Define a new function that invokes the function `contentInDir` to count the number of items in a directory.

44.3 Answers to Quick Checks

QUICK CHECK 44.1

Your function `toInt` should have a signature and implementation similar to the following:

```
import scala.concurrent.{ExecutionContext, Future}

def toInt(f: Future[String])(using ec: ExecutionContext): Future[Int] =
  f.map(_.toInt)
```

QUICK CHECK 44.2

A possible implementation for the function `getUser(orderId: Int)` is the following:

```
def getUser(orderId: Int)(using ec: ExecutionContext): Future[User] =
  getAccount(orderId).map(account => getUser(account.id)).flatten
```

QUICK CHECK 44.3

You should refactor your function `getUser(orderId: Int)` as follows:

```
def getUser(orderId: Int)(using ec: ExecutionContext): Future[User] =
  getAccount(orderId).flatMap(account => getUser(account.id))
```

45

Working with Future: for-comprehension and other operations

After reading this lesson, you will be able to:

- Define a chain of asynchronous operations using for-comprehension
- Select the first `Future` instance to complete, either successfully or not.
- Find the fastest asynchronous operation to produce a value with specific properties.
- Run independent `Future` instances in parallel and collect their results in a sequence.

You have discovered how you can use the `map`, `flatten`, and `flatMap` operations to manipulate the value that asynchronous computations produce. In this lesson, you'll learn how to coordinate multiple `Future` instances using for-comprehension. In particular, you'll see how to provide requirements on when each asynchronous computation should start. You'll discover how to select the first `Future` instance to complete among many and find the one that successfully produces a value with given features. You'll also see how to combine the results of several asynchronous independent computations running in parallel into one sequence. In the capstone, you'll use for-comprehension to define asynchronous computations to execute in a given order for your quiz program.

Consider this

Suppose you are developing a program that depends on a third-party API, which has servers deployed in several regions, such as London, Virginia, and Hong Kong. The servers vary in performance during the day based on their requests volume. You'd like to send the same call to all its servers, get a response from any of them. How would you implement this?

45.1 For-comprehension

Let's consider your program to order items in a store and imagine that you now need to retrieve the full details of an existing order. You already have code to retrieve information of an entity based on its id:

Listing 45.1: Retrieving an order, user, and product by id

```
import scala.concurrent.{ExecutionContext, Future}

case class Order(id: Int, userId: Int, productId: Int, quantity: Double)
case class Product(id: Int, description: Int, price: Double)
case class User(id: Int, fullname: String, email: String)

def getOrder(id: Int)(using ec: ExecutionContext): Future[Order] = ???
def getUser(id: Int)(using ec: ExecutionContext): Future[User] = ???
def getProduct(id: Int)
    (using ec: ExecutionContext): Future[Product] = ???
```

You can combine these functions using the `map` and `flatMap` to retrieve the full details of an order:

Listing 45.2: Retrieving the order details by an order id using map and flatMap

```
case class OrderDetails(order: Order, user: User, product: Product)

def getOrderDetails(orderId: Int)
    (using ec: ExecutionContext): Future[OrderDetails] =
  getOrder(orderId).flatMap { order => ①
    getUser(order.userId).flatMap { user => ②
      getProduct(order.productId).map { product => ③
        OrderDetails(order, user, product) ④
      }
    }
  }
```

- ① Extracting an order
- ② Extracting its user
- ③ Extracting its product
- ④ Combining the three entities to create an `OrderDetails` instance

Listing 45.3 shows you how you can refactor its implementation in a more elegant and readable style using for-comprehension:

Listing 45.3: Retrieving the order details by an order id using for-comprehension

```
case class OrderDetails(order: Order, user: User, product: Product)

def getOrderDetails(orderId: Int)
    (using ec: ExecutionContext): Future[OrderDetails] =
  for {
    order <- getOrder(orderId) ①
    user <- getUser(order.userId) ②
    product <- getProduct(order.productId) ③
  } yield OrderDetails(order, user, product) ④
```

- ① Extracting an order
- ② Extracting its user
- ③ Extracting its product
- ④ Combining the three entities to create an OrderDetails instance

In Scala, for-comprehension allows you to express the combination of the `map` and `flatMap` operations in a more elegant and readable way. The following expressions are equivalent:

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> Future(12/2).flatMap(n => Future(n.toString()))
res0: scala.concurrent.Future[String] = Future(Success(6))

scala> for {
    |   n <- Future(12/2)
    |   res <- Future(n.toString())
    | } yield res
res1: scala.concurrent.Future[String] = Future(Success(6))
```

QUICK CHECK 45.1

Consider the following snippet of code: refactor the function `getUserId` to use for-comprehension.

```
import scala.concurrent.{ExecutionContext, Future}

case class Order(id: Int, userId: Int, productId: Int, quantity: Double)

def getOrder(id: Int)(using ec: ExecutionContext): Future[Order] = ???

def getUserId(orderId: Int)(using ec: ExecutionContext): Future[Int] =
  getOrder(orderId).map(_.userId)
```

Let's have another look at the code you have written in listing 45.3, in which you are chaining three asynchronous computations. First, the operation to get the order by id will run. Once it completes, the one to retrieve the user will commence. Finally, the third one to recover the product starts after the user entity is received. If any of these computations fails, the entire chain will result in a failed asynchronous computation. Running all of them in sequence may seem the most efficient option. However, it is not because the functions `getUser` and `getProduct` are independent: you could execute them in parallel after retrieving the order. Listing 45.4 shows you how to do this:

Listing 45.4: Retrieving order details more efficiently

```
def getOrderDetails(orderId: Int)
    (using ec: ExecutionContext): Future[OrderDetails] =
  for {
    order <- getOrder(orderId)
    futureUser = getUser(order.userId) ①
    futureProduct = getProduct(order.productId) ①
    user <- futureUser ②
    product <- futureProduct ②
  } yield OrderDetails(order, user, product)
```

- ① Scheduling the operation to start. You can omit the keyword `val` here because inside a `for-comprehension` construct.
 ② Extracting the result of the computation once completed successfully

Alternatively, you can also do the following:

Listing 45.5: Another way of retrieving order details more efficiently

```
def getOrderDetails(orderId: Int)
    (using ec: ExecutionContext): Future[OrderDetails] =
  getOrderId.flatMap { orderId => ①
    val futureUser = getUser(orderId.userId) ②
    val futureProduct = getProduct(orderId.productId) ②
    for {
      user <- futureUser ③
      product <- futureProduct ③
    } yield OrderDetails(order, user, product)
  }
```

- ① Using `flatMap` to attach an asynchronous callback to `getOrder`
 ② Scheduling the operation to start
 ③ Extracting the result of the computation once completed successfully

Thanks to this minor change, your program retrieves the order details more efficiently: it fetches the user and product data in parallel instead of waiting for the user entity before requesting the product entity.

QUICK CHECK 45.2

Consider the following snippet of code: improve the execution time of the function `myOp` by running asynchronous operations in parallel where possible.

```
import scala.concurrent.{ExecutionContext, Future}

def opA(a: String)(using ec: ExecutionContext): Future[Long] = ???
def opB(b: Int)(using ec: ExecutionContext): Future[Long] = ???
def opC(c: Long)(using ec: ExecutionContext): Future[Long] = ???

def myOp(text: String)(using ec: ExecutionContext): Future[Long] =
  for {
    a <- opA(text)
    b <- opB(text.length)
    c <- opC(a - b)
  } yield a * b * c
```

45.2 Retrieving the first Future to complete

Consider your program to place an order, and imagine that your store now has multiple warehouses that can supply a product. When checking the availability of a product, you'd like to do so for all warehouses and pick the first one that successfully replied with enough available stocks. Listing 45.6 shows you how to achieve this:

Listing 45.6: Getting the availability of a product from multiple warehouses

```
import scala.concurrent.{ExecutionContext, Future}

sealed trait Warehouse
case object London extends Warehouse
case object Brighton extends Warehouse
case object Leeds extends Warehouse

object Warehouse {
    val all: List[Warehouse] = List(London, Brighton, Leeds)
}

case class Availability(productId: Int,
                      quantity: Double,
                      location: Warehouse)

def checkAvailability(productId: Int, warehouse: Warehouse)
                      (using ec: ExecutionContext):
Future[Availability] = ????

def getAvailability(productId: Int, quantity: Double)
                      (using ec: ExecutionContext):
Future[Option[Availability]] =
  Future.find(❶
    Warehouse.all.map(checkAvailability(productId, _)) ❷
  )(availability => availability.quantity >= quantity) ❸
```

❶ It returns the first asynchronous computation to produce a value with given features if any.

❷ A sequence of asynchronous operations run in parallel.

❸ This is the predicate to select a successfully produced result.

When working with several asynchronous computations that run independently in parallel, Scala offers you several methods to help coordinate them. The most common are the following:

- `firstCompletedOf` – The function `Future.firstCompletedOf[T]` has one parameter of type `Iterable[Future[T]]`, and it returns a value of type `Future[T]` containing the first instance in the sequence to complete, either successfully or unsuccessfully.

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> def futureA = Future { Thread.sleep(42); 42 }
futureA: scala.concurrent.Future[Int]

scala> def futureB = Future(123/0)
```

```
futureB: scala.concurrent.Future[Int]

scala> Future.firstCompletedOf(Seq(futureA, futureB))
res0: scala.concurrent.Future[Int] = Future(Failure(java.lang.ArithmetricException: / by zero))
```

- **find** – The method `Future.find[T]` takes a sequence of asynchronous computations returning a value of type `T` and a predicate `T => Boolean`. It returns a new one containing the first value produced that respects the given requirement, if any.

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> def futureA = Future { Thread.sleep(42); 42 }
futureA: scala.concurrent.Future[Int]

scala> def futureB = Future(123/0)
futureB: scala.concurrent.Future[Int]

scala> val seqAB = Future.find(Seq(futureA, futureB))(_ > 10)
seqAB: scala.concurrent.Future[Option[Int]] = Future(<not completed>

scala> seqAB
seqAB: scala.concurrent.Future[Option[Int]] = Future(Success(Some(42)))

scala> Future.find(Seq(futureA, futureB))(_ > 100)
res0: scala.concurrent.Future[Option[Int]] = Future(Success(None))
// No Future instance in the sequence produces a value bigger than 100!
```

- **sequence** – The function `Future.sequence[T]` takes a sequence of asynchronous computations producing a value `T` and it returns a new asynchronous computation containing all their results in a sequence in the same order. If any of them fail, it completes the new asynchronous computation as a failure.

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> def futureA = Future { Thread.sleep(42); 42 }
futureA: scala.concurrent.Future[Int]

scala> def futureB = Future(123/0)
futureB: scala.concurrent.Future[Int]

scala> def futureC = Future(123)
futureC: scala.concurrent.Future[Int]

scala> Future.sequence(Seq(futureA, futureB, futureC))
res0: scala.concurrent.Future[Seq[Int]] = Future(Failure(java.lang.ArithmetricException: / by zero))
```

```
scala> val seqAC = Future.sequence(Seq(futureA, futureC))
seqAC: scala.concurrent.Future[Seq[Int]] = Future(<not completed>)

scala> seqAC
seqAC: scala.concurrent.Future[Seq[Int]] = Future(Success(List(42, 123)))
```

Have a look at table 45.1 for a summary of the functions you can use to coordinate a sequence of independent asynchronous computations.

Table 45.1: Summary of some of the operations you can use to combine a list of Future instances. The method `firstCompletedOf` returns the first instance of Future to complete, either successfully or unsuccessfully. The function `find` returns the first result of an asynchronous computation that respects a given predicate. The method `sequence` collects all the values that Future instances run in parallel produce.

	Acts on	Signature	Usage
<code>firstCompletedOf</code> <code>f</code>	<code>Future</code>	<code>firstCompletedOf[T](in: Iterable[Future[T]])(using ec: ExecutionContext): Future[T]</code>	It returns the first future to complete, either successfully or unsuccessfully.
<code>find</code>	<code>Future</code>	<code>find[T](in: Iterable[Future[T]])(p: T => Boolean)(using ec: ExecutionContext): Future[Option[T]]</code>	It returns the first future to successfully produce a value respecting the given predicate, if any.
<code>sequence</code>	<code>Future</code>	<code>sequence[T](in: Iterable[Future[T]])(using ec: ExecutionContext): Future[Iterable[T]]</code>	Combines all the values produced into a sequence into a new Future instance. If any of them fails, it returns a failed Future instance.

QUICK CHECK 45.3

Define a function called `firstSuccessful` that takes a list of Future instances and returns the first future instance to complete successfully, if any.

```
import scala.concurrent.{ExecutionContext, Future}

def firstSuccessful[T](in: List[Future[T]])
    (using ec: ExecutionContext): Future[Option[T]]
```

Blocking on Future

When coding using asynchronous computation, you should never block a thread waiting for a `Future` to complete. However, there are cases where this is needed – particularly when writing tests.

For example, you may want to write a test in which you assert that a certain asynchronous should complete, either with a failure or a success, within some time. You can achieve this by using the function `Await.ready`. It takes a future instance and a duration, and it either returns the completed future instance once it completes or throws an exception when the time has run out.

```
def myFuture: Future[Int] = Future{Thread.sleep(10 /*ms*/); 42 }
```

```
import scala.concurrent.Await
```

```
import scala.concurrent.duration._
```

```
Await.ready(myFuture, atMost = 5 milliseconds)
```

You may also want to assert that your `Future` instance can successfully produce a value within a given time. The method `Await.result` takes a future instance and a duration. It either returns the produced value produced or throws an exception on failure or on timeout.

```
import scala.concurrent.Await
```

```
import scala.concurrent.duration._
```

```
val n: Int = Await.result(myFuture, atMost = 2 seconds)
```

45.3 Summary

In this lesson, my objective was to teach about some of the operations you can perform on asynchronous computations.

- You have learned how to combine asynchronous computation in a sequence using for-comprehension.
- You have discovered how to select the first asynchronous computation to complete using the `Future.firstCompletedOf` method.
- You have mastered how to find the first to produce a value that respects given properties thanks to the function `Future.find`.
- You have seen how to execute independent asynchronous computations in parallel and collect their results into a sequence using `Future.sequence`.

Let's see if you got this!

TRY THIS

Let's consider the function `contentInDir` that you have seen in the Try This section of Lesson 44 to list the content of a given directory:

```
import java.io.File
import scala.concurrent.{ExecutionContext, Future}

def contentInDir(path: String)
  (using ec: ExecutionContext): Future[List[String]] =
  Future {
    val file = new File(path)
    if (file.isDirectory)
```

```
    file.listFiles().toList.map(_.getAbsolutePath)
  else List.empty
}
```

Write a function to asynchronously list the content of a directory and all its subdirectories. Visit each subdirectory independently to improve its runtime performance.

```
def allContentInDir(path: String)
  (using ec: ExecutionContext): Future[List[String]]
```

45.4 Answers to Quick Checks

QUICK CHECK 45.1

You should re-implement your `getUserID` function as follows:

```
def getUserId(orderId: Int)(using ec: ExecutionContext): Future[Int] =
  for {
    order <- getOrder(orderId)
  } yield order.userId
```

QUICK CHECK 45.2

You can improve the execution time of `myOp` by scheduling the functions `opA` and `opB` in parallel, because they are independent. Your solution should look like similar to the following:

```
def myOp(text: String)(using ec: ExecutionContext): Future[Long] = {
  val futureOpA = opA(text)
  val futureOpB = opB(text.length)
  for {
    a <- futureOpA
    b <- futureOpB
    c <- opC(a - b)
  } yield a * b * c
}
```

QUICK CHECK 45.3

A possible implementation for the function `firstSuccessful` is the following:

```
def firstSuccessful[T](in: List[Future[T]])
  (using ec: ExecutionContext): Future[Option[T]] =
  Future.find(in)(_ => true)
```

46

Database queries with Quill

After reading this lesson, you will be able to:

- Connect to a database and execute SQL queries asynchronously using the Quill library
- Match database tables to case classes
- Write code to generate and run queries to select, insert, update and delete records

After learning about the type Future, you'll discover how to connect and asynchronously perform queries to a database using a popular library called Quill. In particular, you'll see how to start a test PostgreSQL database instance and connect to it. You'll see how to execute SQL queries. You'll master how to define case classes that correspond to its tables and write code that generates SQL queries to retrieve, insert, update, and delete its records. In the capstone, you'll use a database to store the questions and user information for a quiz application.

Consider this

Suppose you are developing software to manage orders and bookings in a restaurant that uses a database to read and write data. How would you connect to it, retrieve and save records?

46.1 Project Setup

Rather than creating an sbt project from scratch, let's download a base project as your starting point: the following subsections provide instructions on how to do this.

46.1.1 Download the base project

First, you need to create an sbt project and download its external dependencies. Navigate to an empty folder and use the `git` command to download the base project for this lesson:

```
$ git init
$ git remote add daniela https://github.com/DanielaSfregola/get-programming-with-scala.git
$ git fetch daniela
$ git checkout -b my_lesson46 daniela/baseline_unit7_lesson46
```

In this lesson, you'll also use a library called testcontainers, which uses Docker to initialize a temporary PostgreSQL database instance for your application. Make sure that your machine has Docker installed by running the following command:

```
$ docker --version
Docker version 20.10.0, build 7287ab3
```

If you need to install Docker, please refer to the instructions in lesson 2, section 2.4.2.

After running these commands, your directory should contain a base sbt project including the following files:

- project/build.properties has the sbt version of your project.
- build.sbt defines your Scala version and the dependencies you are going to use. Listing 46.1 shows its content.

Listing 46.1: the build.sbt file

```
//build.sbt file

name := "get-programming-with-scala-lesson46"

version := "0.1"

scalaVersion := "2.13.1"

libraryDependencies ++= List(
  "io.getquill" %% "quill-async-postgres" % "3.5.2", ①
  "org.testcontainers" % "postgresql" % "1.13.0", ②
  "org.postgresql" % "postgresql" % "42.2.11", ③
  "ch.qos.logback" % "logback-classic" % "1.2.3" ④
)
```

① Quill allows you to connect and query a database from a Scala codebase

② Java library to spin database instances up for test purposes

③ The PostgreSQL driver

④ Logback manages the logs of your application

- src/main/resource/logback.xml specifies the logging format to use in your application.
- src/main/resource/init.sql is an SQL script to create the tables of your database and insert a few records.
- src/main/scala/org/example/registrations/PostgreSQL.scala defines a class to start test PostgreSQL database instance: you are going to learn about it in the next section.

Execute the command `sbt compile` to download all the external dependencies and compile the existing code.

46.1.2 Starting the PostgreSQL server

When developing a real-world application, you should rely on a database of appropriate size, secure, and periodically backed up: explaining how to do this is beyond this book's scope. Instead, you are going to run a small temporary PostgreSQL database using the `org.testcontainers` library. The `PostgreSQL` class allows you to configure, start, and stop a base Docker container containing a PostgreSQL database:

Listing 46.2: The PostgreSQL class

```
package org.example.registrations

import java.net.URL

import com.typesafe.config._
import org.testcontainers.containers.PostgreSQLContainer
import org.slf4j.LoggerFactory

class PostgreSQL(initScript: String) {

    private val logger = LoggerFactory.getLogger(this.getClass)

    private val container: PostgreSQLContainer[_] = { ❶
        val psql: PostgreSQLContainer[_] = new PostgreSQLContainer().withInitScript(initScript) ❷
        logger.info(s"Starting container...")
        psql.start() ❸
        psql
    }

    def stop() = { ❹
        logger.info("Stopping container...")
        container.stop()
    }

    val config: Config = { ❺
        val components = List(
            container.getJdbcUrl,
            s"user=${container.getUsername}",
            s"password=${container.getPassword}"
        )
        ConfigFactory.empty().withValue(
            "url", ConfigValueFactory.fromAnyRef(components.mkString("&"))
        )
    }
}
```

❶ A reference to the PostgreSQL server inside the docker container

❷ It initializes the database with a given script

❸ It starts the PostgreSQL server

❹ It stops the PostgreSQL server

❺ Details on how to connect to the PostgreSQL server

The library `org.testcontainers` is a popular Java library to spin up throwaway, lightweight instances of common databases for test purposes. Some of the supported databases are MySQL,

Cassandra, PostgreSQL, Kafka, Neo4j: their website testcontainers.org lists all the modules they offer.

In this lesson, you'll write queries to read and write data about customers of a store. The `init.sql` file defines a table to represent a user, and it inserts some sample records in it. Listing 46.3 shows you part of its content:

Listing 46.3: The `init.sql` script

```
// src/main/resource/init.sql file

CREATE TABLE IF NOT EXISTS customer (
    id INTEGER,
    name VARCHAR(45) NOT NULL,
    PRIMARY KEY (id)
);

INSERT INTO customer (id, name)
VALUES
    (1, 'Jon Snow'),
    (2, 'Daenerys Targaryen'),
    (3, 'Arya Stark');
```

Let's try to start and stop your PostgreSQL server: your machine will download everything needed the first time. Execute the command `sbt console` from your sbt project's root directory to try out your code interactively:

```
$ sbt console
[...]
[info] Starting scala interpreter...
Welcome to Scala 2.13.4 (OpenJDK 64-Bit Server VM, Java 15.0.1).
Type in expressions for evaluation. Or try :help.

scala>
```

You can initialize and start your PostgreSQL server as following:

```
scala> import org.example.registrations._
import org.example.registrations._

scala> val psqlServer = new PostgreSQL("init.sql")
Starting container...
...
Container postgres:9.6.12 started in PT8.103S
Executing database script from init.sql
Executed database script from init.sql in 150 ms.
HikariPool-1 - Starting...
HikariPool-1 - Start completed.
psqlServer: org.example.registrations.PostgreSQL = org.example.registrations.PostgreSQL@212bd091

scala>
```

Invoking the `stop` function will destroy the container:

```
scala> psqlServer.stop
Stopping container...
```

The setup is now complete, and you are ready to connect to your database and run queries on it.

46.2 Connecting to the PostgreSQL Server

When connecting to a database, you need to provide information about its type, driver, and location. Listing 46.4 shows you how to do this using Quill:

Listing 46.4: Defining a database context

```
// file src/main/scala/org/example/registrations/TestDatabase.scala

package org.example.registrations

import io.getquill.{PostgresAsyncContext, SnakeCase}

object TestDatabase {

    private val psqlServer = new PostgreSQL("init.sql") ①
    val ctx = new PostgresAsyncContext(SnakeCase, psqlServer.config) ②

    def stop(): Unit = psqlServer.stop() ③
}
```

- ① It starts the PostgreSQL test server
- ② It defines how to connect to it
- ③ It stops the PostgreSQL test server

When using Quill, you need to initialize a context with information about your database and its connection details. First, you need to pick a general naming convention: use one amongst `UpperCase`, `LowerCase`, `SnakeCase`, `CamelCase`. If this is not consistent, you can still provide custom settings when matching tables with your Scala code. You then need to give its location and driver by supplying a `Config` instance or a configuration file to parse: this is usually called `application.properties` and lives in your resources directory. The Quill library offers you several modules to support various types of databases. For example, the snippet in Listing 46.5 shows you how to could set the library up to connect to a MySQL database:

Listing 46.5: Example of Context creation from for a MySQL database

```
// in your build.sbt, add the Quill module and the driver for your database

libraryDependencies ++= Seq(
    "mysql" % "mysql-connector-java" % "8.0.15",
    "io.getquill" %% "quill-jdbc" % "3.5.2"
)

// file src/main/resource/application.properties

mydb {
    dataSourceClassName=com.mysql.cj.jdbc.MysqlDataSource
    dataSource.url=jdbc:mysql://host/database
    dataSource.user=root
    dataSource.password=root
}
```

```

dataSource.cachePrepStmts=true
dataSource.prepStmtCacheSize=250
dataSource.prepStmtCacheSqlLimit=2048
connectionTimeout=30000
}

// in your scala file, define a quill context

import io.getquill._
val ctx = new MysqlJdbcContext(CamelCase, "mydb")

```

Have a look at <https://getquill.io/> for more information on defining a database context and the full list of his supported databases.

QUICK CHECK 46.1

Define a Quill context to connect to a PostgreSQL database by parsing a configuration file and using a camel case naming convention. Your `application.properties` contains the following data:

```

db.url=postgresql://host:5432/database?user=root&password=root

app {
  name=my_application
  owner=my_team
  port=8080
}

```

46.3 Executing queries

Imagine that you'd like to run a simple query to ensure that you can connect to your database. Listing 46.6 shows you how to do this:

Listing 46.6: Testing a database connection

```

// file src/main/scala/org/example/registrations/Queries.scala

package org.example.registrations

import io.getquill.{PostgresAsyncContext, SnakeCase}
import scala.concurrent.{ExecutionContext, Future}

class Queries(ctx: PostgresAsyncContext[SnakeCase.type]) { ①
  import ctx._ ②

  def testConnection()(implicit ec: ExecutionContext) ③
  : Future[Boolean] = {
    val q = quote { infix"SELECT 1".as[Int] } ④
    val result: Future[Int] = run(q) ⑤
    result.map(_ == 1) ⑥
  }
}

```

① The Quill context provides information on your database. `SnakeCase.type` returns the type of the object `SnakeCase`

② It enables Quill's Domain Specific Language (DSL)

③ The query returns a Future, so you need an execution context

- ④ It defines a SQL query that returns an integer
- ⑤ It runs the defined query asynchronously. Quill uses an implicit conversion to transform its instance of type `ctx.Result` that `run` produces into a `Future` one.
- ⑥ It checks the result matches expectations

You can now check if you can connect to your database as follows:

```
scala> import org.example.registrations._
import org.example.registrations._

scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> val queries = new Queries(TestDatabase.ctx)
queries: org.example.registrations.Queries = org.example.registrations.Queries@7107c07d

scala> queries.testConnection()
res0: scala.concurrent.Future[Boolean] = Future(<not completed>)

scala> res0
res1: scala.concurrent.Future[Boolean] = Future(Success(true))
```

When connecting to a database, you need to provide a `Quill` context. The class `Queries` requires one for a PostgreSQL database with a snake case naming convention: the compiler will use this information to ensure your queries are syntactically correct. You need to include the instruction `import ctx._` to enable a Domain Specific Language (i.e., `DSL`) that `Quill` offers to define SQL queries, such as the functions `quote` and `run`. The prefix `infix` indicates that the string should be a parsable SQL instruction, while the method `as` defines the expected type of each record of the query. Finally, you invoke the function `run` on your context to execute the query asynchronously.

QUICK CHECK 46.2

The following snippet of code defines a SQL query to retrieve all the customers' names in the database. What happens when you execute it? Use the Scala REPL within your sbt console to validate your hypothesis.

```
import org.example.registrations._
import TestDatabase.ctx._
import io.getquill.Query

import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

val customers: Future[List[String]] = run(quote {
  infix"SELECT name FROM customers".as[List[String]]
})
```

46.4 Running generated queries

Imagine you want to manipulate the customer data stored in the database. In particular, you'd like to

- Retrieve all customers
- Get the customer name by its id

- Insert a new customer
- Update the name of a customer by id
- Delete a customer by id

Listing 46.7 shows you how to achieve this using Quill's query generation:

Listing 46.7: Generated queries for Customers

```
// file src/main/scala/org/example/registrations/CustomerQueries.scala

package org.example.registrations

import io.getquill.{PostgresAsyncContext, SnakeCase}
import scala.concurrent.{ExecutionContext, Future}

case class Customer(id: Int, name: String) ①

class CustomerQueries(ctx: PostgresAsyncContext[SnakeCase.type]) {
  import ctx._

  private val customers = quote { query[Customer] } ②

  def all()(implicit ec: ExecutionContext): Future[List[Customer]] = {
    // Generated SQL: SELECT x.id, x.name FROM customer x
    run(customers)
  }

  def nameById(id: Int)
    (implicit ec: ExecutionContext): Future[List[String]] = {
    // Generated SQL: SELECT x1.name FROM customer x1 WHERE x1.id = ?
    val q = quote {
      customers.filter(_.id == lift(id)) ③
        .map(_.name) ④
    }
    run(q)
  }

  def save(customer: Customer)
    (implicit ec: ExecutionContext): Future[String] = {
    // Generated SQL:
    // INSERT INTO customer (id, name) VALUES (?, ?) RETURNING name
    val q = quote {
      customers.insert(lift(customer)) ⑤
        .returning(_.name) ⑥
    }
    run(q)
  }

  def updateNameById(id: Int, nameToUpdate: String)
    (implicit ec: ExecutionContext): Future[Long] = {
    // Generated SQL: UPDATE customer SET name = ? WHERE id = ?
    val q = quote {
      customers.filter(_.id == lift(id))
        .update(_.name -> lift(nameToUpdate)) ⑦
    }
    run(q)
  }
}
```

```

def deleteById(id: Int)
    (implicit ec: ExecutionContext): Future[Long] = {
  // Generated SQL: DELETE FROM customer WHERE id = ?
  val q = quote {
    customers.filter(_.id == lift(id)).delete ⑧
  }
  run(q)
}
}

```

- ① The case class to represent a customer
- ② `query[Customer]` matches your case class to a table with name `customer` and two columns `id` and `name`.
- ③ `filter` allows you to select only records that respect a given predicate. The function `lift` indicates that the value is not static: your program will provide it at runtime
- ④ `map` selects only specific columns in your query
- ⑤ `insert` allows you to save new records into your database
- ⑥ `returning` defines what information to return after an operation
- ⑦ `update` changes the value of the given fields.
- ⑧ `delete` removes all the selected records.

Writing queries by hand can have disadvantages: different types of databases use different SQL dialects. Ensuring that they match your program's expectations can be challenging to maintain. Quill offers you a DSL to generate queries at compile-time and help you mitigate these issues.

In the example you have seen in listing 46.8, your Quill context uses a snake case naming convention. The library uses this information to match the database tables with your case classes. For example, the expression `query[Customer]` indicates that it should match the table `customer` containing two columns `id` and `name` to your case class `Customer`. You can also provide a custom matching using the function `querySchema`. For example, consider the following instruction:

```
quote { querySchema[Customer]("customers_table",
  _.id -> "customer_id",
  _.name -> "name_column") }
```

It defines a match between your case class `Customer` and a table with name `customers_table` with the columns `customer_id` and `name_column`.

Quill generates queries at compile time: you will see how it translates them into SQL statements in the console when you compile. The function `lift` indicates which values in your queries are dynamic and need adjusting during execution. Dynamic values are values that your program cannot provide at compile-time, but they depend on its runtime execution. Generated queries mark them with a '?' symbol until their value is known.

You can now connect to the database and manipulate your customer data:

```

scala> import org.example.registrations._
import org.example.registrations._

scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> val customers = new CustomerQueries(TestDatabase.ctx)
customers: org.example.registrations.CustomerQueries =
  org.example.registrations.CustomerQueries@2dd8d946

```

```

scala> customers.all
res0: scala.concurrent.Future[List[org.example.registrations.Customer]] =
  Future(Success(List(Customer(1,Jon Snow), Customer(2,Daenerys Targaryen), Customer(3,Arya
  Stark)))))

scala> customers.save(Customer(4, "Martin"))
res1: scala.concurrent.Future[String] = Future(Success(Martin))

scala> customers.updateNameById(4, "Daniela")
res2: scala.concurrent.Future[Long] = Future(Success(1))

scala> customers.nameById(4)
res3: scala.concurrent.Future[List[String]]= Future(Success(List(Daniela)))

scala> customers.deleteById(4)
res4: scala.concurrent.Future[Long] = Future(Success(1))

scala> customers.all
res5: scala.concurrent.Future[List[org.example.registrations.Customer]] =
  Future(Success(List(Customer(1,Jon Snow), Customer(2,Daenerys Targaryen), Customer(3,Arya
  Stark)))))


```

QUICK CHECK 46.3

Add a function `customersByName` to your class `CustomerQueries` to generate and run a query to retrieve customers with a given name.

```

def customersByName(name: String)(implicit ec: ExecutionContext)
  : Future[List[Customer]] = ???
```

46.5 Summary

In this lesson, my objective was to teach you how to query a database using the library Quill.

- You have learned how to connect a database and run queries asynchronously on it.
- You have seen how to match the results to your queries to your case classes.
- You have discovered how to generate SQL queries to select, insert, update, and delete records using Quill's DSL.

Let's see if you got this!

TRY THIS

The `init.sql` file you have downloaded for this lesson also creates another table called `product` with the following structure:

```

CREATE TABLE IF NOT EXISTS product (
  id INTEGER,
  title VARCHAR(45) NOT NULL,
  creation_date DATE NOT NULL,
  PRIMARY KEY (id)
);
```

Define functions to perform the following operations:

- create a product
- select all of those with a given title

- change the title of a specific product
- delete a product by id.

46.6 Answers to Quick Checks

QUICK CHECK 46.1

You can define a Quill context for your database as the following:

```
import io.getquill._
new PostgresAsyncContext(CamelCase, "db")
```

QUICK CHECK 46.2

The code compiles, but it fails at runtime because the table's name is incorrect: the correct table name is customer, not customers.

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

import org.example.registrations._
import TestDatabase.ctx._
import io.getquill.Query._

import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

val customers: Future[List[String]] = run(quote {
  infix"SELECT name FROM customers".as[Query[String]]
})

// Exiting paste mode, now interpreting.

val customers: Future[List[String]] = run(quote {
  ^
<pastie>:7: SELECT x.* FROM (SELECT name FROM customers) AS x
import org.example.registrations._
import TestDatabase.ctx._
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
customers: scala.concurrent.Future[List[String]] = Future(<not completed>
Error with message -> ErrorMessage(fields=HashMap(Position -> 35, Line -> 1160, V -> ERROR,
Message -> relation "customers" does not exist, Severity -> ERROR, File ->
parse_relation.c, SQLSTATE -> 42P01, Routine -> parserOpenTable))
```

If you correct your query, you code will return the name of the three customers in your database:

```
scala> val customers: Future[List[String]] = run(quote{
  | infix"SELECT name FROM customer".as[Query[String]] } )
  ^
  SELECT x.* FROM (SELECT name FROM customer) AS x
res0: Future[List[String]] =
Future(Success(List(Jon Snow, Daenerys Targaryen, Arya Stark)))
```

Note that the above snippet uses `as[Query[String]]` instead of `as[String]` because the defined query can return zero or more records, rather than one.

QUICK CHECK 46.3

A possible implementation for the function `customersByName` is the following:

```
def customersByName(name: String)(implicit ec: ExecutionContext)
    : Future[List[Customer]] = {
  val q = quote { customers.filter(_.name == lift(name)) }
  run(q)
}
```

47

The Quiz Application: Part 1

In this capstone, you will:

- Read and write the quiz data from a PostgreSQL database asynchronously.
- Define SQL queries to create, retrieve, update, and delete records.
- Chain multiple asynchronous computations to safely store a question and its answers

In this capstone, you'll define an application's data access layer to create and answer quizzes: you'll implement its business logic and HTTP API at the end of the next unit. Your quiz application has the following requirements:

- It should read and write categories and a set of questions and answers assigned to them.
- Its users should pick a category, answer randomly selected questions about it, and receive a final score based on their performance.

Your application has access to a PostgreSQL database containing three tables called category, question, and answers: figure 47.1 shows their table structure.



Figure 47.1: Visualization of your database schema: primary keys are in bold, while foreign keys are linked to their reference. The entity-relationship diagram has been drawn using dbdiagram.io.

47.1 Download the base project

First, let's speed things up by downloading a base project for you to use. In an empty directory of your choice, run the following `git` commands to download the remote branch:

```
$ git init
$ git remote add daniela https://github.com/DanielaSfregola/get-programming-with-scala.git
$ git fetch daniela
$ git checkout -b my_lesson47 daniela/baseline_unit7_lesson47
```

You will notice lots of similarities with the baseline project you have used in the previous lesson. It contains the following files:

- The file `project/build.properties` defines the `sbt` version.
- `build.sbt` defines the Scala version and the external dependencies for your project (see listing 47.1).

Listing 47.1: The build.sbt file

```
// file build.sbt

name := "get-programming-with-scala-lesson47"

version := "0.1"

scalaVersion := "2.13.1"

libraryDependencies ++= Seq(
  "io.getquill" %% "quill-async-postgres" % "3.5.2",
  "org.testcontainers" % "postgresql" % "1.13.0",
  "org.postgresql" % "postgresql" % "42.2.11",
  "ch.qos.logback" % "logback-classic" % "1.2.3"
)
```

- `src/main/resources/logback.xml` provides the logging format your application will use.
- The files `PostgreSQL.scala` and `TestDatabase.scala` in the directory `src/main/scala/org/example/quiz` have been copied from the previous lesson. They allow your application to start a small temporary instance of a PostgreSQL database initialized by loading an SQL script: in a real-world application, you should use a database that is permanent, monitored, and backed-up regularly.
- `src/main/resources/init.sql` is the SQL script to create the tables in the temporary database. Listing 47.2 shows its content:

Listing 47.2: The init.sql file

```
CREATE TABLE IF NOT EXISTS category (
  id SERIAL, ①
  name VARCHAR(45) NOT NULL UNIQUE,
  PRIMARY KEY (id)
);

CREATE TABLE IF NOT EXISTS question (
  id SERIAL, ①
  category_id INTEGER,
```

```

text VARCHAR(45) NOT NULL,
PRIMARY KEY (id),
FOREIGN KEY (category_id) REFERENCES category(id) ON DELETE CASCADE ②
);

CREATE TABLE IF NOT EXISTS answer (
    id SERIAL, ①
    question_id INTEGER,
    text VARCHAR(45) NOT NULL,
    is_correct BOOLEAN,
    PRIMARY KEY (id),
    FOREIGN KEY (question_id) REFERENCES question(id) ON DELETE CASCADE ②
);

```

- ① The database generates a value for it during the record insertion
 ② It deletes the record if deleting the reference

You can now launch sbt to download the external dependencies and compile the existing code using the command `sbt compile`. You are now ready to start developing your application!

47.2 Health Check Queries

In this capstone, your objective is to implement your application's data access layer: this is a set of classes that allow you to read and write data from a database. First, let's create a package to group these classes: let's call it `dao`. The term *dao* stands for "Data Access Object": this term indicates the abstraction between an application and its persistence layer in software design. Separating how your application exposes data to the public from how it stores them makes them easier to maintain and evolve.

You can now add the class `GenericDao` to perform a simple query to ensure your application can successfully connect to the database:

Listing 47.3: The class GenericDao

```

// file src/main/scala/org/example/quiz/dao/GenericDao.scala

package org.example.quiz.dao

import io.getquill.{PostgresAsyncContext, SnakeCase}

import scala.concurrent.{ExecutionContext, Future}

class GenericDao(ctx: PostgresAsyncContext[SnakeCase.type])
  (implicit ec: ExecutionContext) {
  import ctx._

  def testConnection(): Future[Boolean] = { ②
    val q = quote { infix"SELECT 1".as[Int] }
    val result: Future[Int] = run(q)
    result.map(_ == 1)
  }
}

```

- ① Importing the Quill query domain-specific language
 ② Returning true if it returned the expected value

Your application will use the `testConnection` function to health check the connectivity to its database.

47.3 Category Queries

Let's now define how to interact with the table `category` in your database. First, let's define a case class that matches its structure. This representation is specific to the database structure: let's define a new package called `dao.records`.

Listing 47.4: The Category Record

```
// file src/main/org/example/quiz/dao/records/Category.scala

package org.example.quiz.dao.records

case class Category(id: Long = 0, ①
                    name: String)
```

① The column is a serial primary key, which means the database is responsible for assigning a value for it

Your application can perform the following operations on the table `category`:

- Create a category
- Retrieve all categories
- Delete a category with a given id

The class `CategoryDao.scala` in the package `dao` defines how to execute them:

Listing 47.5: The CategoryDao class

```
// file src/main/org/example/quiz/dao/CategoryDao.scala

package org.example.quiz.dao

import io.getquill.{SnakeCase, PostgresAsyncContext}
import org.example.quiz.dao.records.Category

import scala.concurrent.{ExecutionContext, Future}

class CategoryDao(ctx: PostgresAsyncContext[SnakeCase.type]) ①
  (implicit ec: ExecutionContext) {
  import ctx._ ②

  private val categories = quote { query[Category] }

  def save(category: Category): Future[Long] = { ③
    val q = quote {
      categories.insert(lift(category)) ④
        .returningGenerated(_.id) ⑤
    }
    run(q)
  }

  def all(): Future[List[Category]] = run(categories)

  def deleteById(id: Long): Future[Boolean] = { ⑥
```

```

    val q = quote { categories.filter(_.id == lift(id)).delete } 7
    run(q).map(_ > 0)
}

}

```

- 1** It picks a naming convention for the table
- 2** It imports the Quill query domain-specific language
- 3** It returns the id the database has assigned to the category
- 4** It uses returningGenerated rather than returning because id is a value that the database generated.
- 5** Thanks to the returningGenerated function, Quill excludes the value id from the insertion query as the database selects its value.
- 6** It returns true if it deleted at least one record.
- 7** When deleting a category, the database will also ensure to delete its questions.

Your application will use these operations to define quiz categories and display them to the user.

47.4 Question and Answer Queries

You now need to define a representation for the tables question and answer: define two case classes in the package `dao.records` as the following:

Listing 47.6: The Question and Answer Records

```

// file src/main/scala/org/example/quiz/dao/records/Question.scala

package org.example.quiz.dao.records

case class Question(id: Long = 0, 1
                    text: String,
                    categoryId: Long)

// file src/main/scala/org/example/quiz/dao/records/Answer.scala

package org.example.quiz.dao.records

case class Answer(id: Long = 0, 1
                  questionId: Long = 0, 2
                  text: String,
                  isCorrect: Boolean = false)

```

- 1** The column is a serial primary key, which means the database is responsible for assigning a value for it
- 2** Your application will adjust its value as soon as the database assigns an id to the new question record

The records in the tables question and answer are strictly correlated: having a question without answers or answers without a question is not meaningful to your quiz application. To ensure they will be consistent, let's define a class called `QuestionAnswerDao` that acts on both the tables.

Your application can perform the following operations on the tables question and answer:

- Create a question together with its answers.
- Find all the questions and their answers assigned to an existing category.
- Delete a question and its answers by a given question id.

Listing 47.7 shows you how to achieve this:

Listing 47.7: The QuestionAnswerDao class

```
// file src/main/scala/org/example/quiz/dao/QuestionAnswerDao.scala

package org.example.quiz.dao

import io.getquill.{SnakeCase, PostgresAsyncContext}
import org.example.quiz.dao.records.{Answer, Question}

import scala.concurrent.{ExecutionContext, Future}

class QuestionAnswerDao(ctx: PostgresAsyncContext[SnakeCase.type]) ①
  (implicit ec: ExecutionContext) {

  import ctx._

  private val questions = quote { query[Question] }
  private val answers = quote { query[Answer] }

  def save(newQuestion: Question, newAnswers: List[Answer]): Future[(Long, List[Long])] = ③
    val saveQuestion = quote {
      questions.insert(lift(newQuestion)).returningGenerated(_.id)
    }

  val saveAnswers = { questionId: Long =>
    val newAnswersWithQuestionId =
      newAnswers.map(_.copy(questionId = questionId)) ④
    quote {
      liftQuery(newAnswersWithQuestionId).foreach { a => ⑤
        answers.insert(a).returningGenerated(_.id)
      }
    }
  }

  transaction { implicit ec => ⑥
    for {
      questionId <- run(saveQuestion)
      answerId <- run(saveAnswers(questionId))
    } yield questionId -> answerId
  }
}

def pickById(categoryId: Long, n: Int): Future[Map[Question, List[Answer]]] = {⑦
  val result: Future[List[(Question, Answer)]] = run {
    for {
      question <- questions.filter(_.categoryId == lift(categoryId))
    .sortBy(_ => infix"random()").take(lift(n)) ⑧
      answer <- answers.filter(_.questionId == question.id)
    } yield question -> answer
  }

  result.map { questionsAndAnswers => ⑨
    val questions: List[Question] =
      questionsAndAnswers.map { case (q, _) => q }.distinct
    val answersByQuestionId: Map[Long, List[Answer]] =
      questionsAndAnswers.map { case (_, a) => a }.groupBy(_.questionId)
  }
}
```

```

        questions.map { question =>
          question -> answersByQuestionId.getOrElse(question.id, List.empty)
        }.toMap
      }
    }

def deleteById(id: Long): Future[Boolean] = { ⑩
  val q = quote { questions.filter(_.id == lift(id)).delete } ⑪
  run(q).map(_ > 0)
}

def getCorrectQuestionAnswers(questionIds: List[Long]): Future[List[(Long, Long)]] = { ⑫
  val q = quote {
    for {
      question <- questions.filter(q => lift(questionIds).contains(q.id))
      correctAnswer <- answers.filter(a =>
        a.questionId == question.id && a.isCorrect)
    } yield question.id -> correctAnswer.id
  }
  run(q)
}
}

```

- ① Both tables have a snake case naming convention.
- ② It imports the Quill query domain-specific language
- ③ It returns the ids the database assigned to the question and answers
- ④ It updates the answers with the generated question id before inserting them
- ⑤ The function `liftQuery` allows you to generate a query to perform a bulk insert.
- ⑥ It defines a transaction so that we can rollback all insertions in case of failure.
- ⑦ It returns the questions and their answers in a dictionary-like structure.
- ⑧ It sorts records randomly and selecting up to n
- ⑨ It converts from a flat structure of type `Future[List[(Question, Answer)]]` to one of type `Future[Map[Question, List[Answer]]]`
- ⑩ It returns true if it deleted at least one record.
- ⑪ When deleting a question, the database ensures to delete its answers.
- ⑫ It returns the correct answer ids for the given question ids.

The `transaction` function allows you to define a database transaction when running queries: it rolls back all the applied changes if any of them fails, maintaining the database data consistency. The `transaction` function is a high order function that takes a function of type `TransactionExecutionContext => Future[T]` as its parameter. The class `TransactionExecutionContext` lives in the package `io.getquill.context.async`, and it is a custom implementation of the class `scala.concurrent.ExecutionContext`.

The implementation of the data access layer is now complete and ready to use.

47.5 Let's give it a try!

Even the implementation of your application is not complete yet, let's use the sbt console to load the project inside the Scala REPL and see it in action. Navigate to the root directory of your project and run the command `sbt console`:

```
$ sbt console
...
[info] /Users/danielasfregola/Development/get-programming-with-
```

```

scala/unit7/lesson47/src/main/scala/org/example/quiz/dao/CategoryDao.scala:22:8: SELECT
  x.id, x.name FROM category x
[info]      run(categories)
[info]      ^
...
[info] /Users/danielasfregola/Development/get-programming-with-
  scala/unit7/lesson47/src/main/scala/org/example/quiz/dao/QuestionAnswerDao.scala:64:8:
  DELETE FROM question WHERE id = ?
[info]      run(q).map(_ > 0)
[info]      ^
[info] Starting scala interpreter...
Welcome to Scala 2.13.4 (OpenJDK 64-Bit Server VM, Java 15.0.1).
Type in expressions for evaluation. Or try :help.

scala>

```

When compiling, Quill shows you the SQL queries it generated according to the naming convention you have provided. First, let's start your temporary database instance and define the classes to run queries on it:

```

scala> :paste
// Entering paste mode (ctrl-D to finish)

import scala.concurrent.ExecutionContext.Implicits.global
import org.example.quiz._
import org.example.quiz.dao._
import org.example.quiz.dao.records._

val genericDao = new GenericDao(TestDatabase.ctx)
val categoryDao = new CategoryDao(TestDatabase.ctx)
val qaDao = new QuestionAnswerDao(TestDatabase.ctx)

// Exiting paste mode, now interpreting.

Starting container...
Loaded org.testcontainers.dockerclient.UnixSocketClientProviderStrategy from
  ~/testcontainers.properties, will try it first
Accessing docker with local Unix socket
...
Executing database script from init.sql
Executed database script from init.sql in 174 ms.
import scala.concurrent.ExecutionContext.Implicits.global
import org.example.quiz._
import org.example.quiz.dao._
import org.example.quiz.dao.records._
genericDao: org.example.quiz.dao.GenericDao = org.example.quiz.dao.GenericDao@77531d42
categoryDao: org.example.quiz.dao.CategoryDao = org.example.quiz.dao.CategoryDao@2ae37a6b
qaDao: org.example.quiz.dao.QuestionAnswerDao = org.example.quiz.dao.QuestionAnswerDao@23787cc0

scala>

```

You can now check if you can connect to the database:

```

scala> genericDao.testConnection()
res0: scala.concurrent.Future[Boolean] = Future(Success(true))

scala>

```

Let's try to create a category and then define two questions for it:

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> :paste
// Entering paste mode (ctrl-D to finish)

val questionA = { categoryId: Long =>
  val question = Question(categoryId = categoryId, text = "Is this a test?")
  val answers = List(Answer(text = "True", isCorrect = true),
    Answer(text = "False"))
  qaDao.save(question, answers)
}

val questionB = { categoryId: Long =>
  val question = Question(categoryId = categoryId, text = "Another test?")
  val answers = List(Answer(text = "XXX"),
    Answer(text = "YYY", isCorrect = true),
    Answer(text = "ZZZ"))
  qaDao.save(question, answers)
}

for {
  categoryId <- categoryDao.save(Category(name = "Test"))
  _ <- Future.sequence(List(questionA(categoryId), questionB(categoryId)))
} yield categoryId

// Exiting paste mode, now interpreting.

questionA: Long => scala.concurrent.Future[(Long, List[Long])] =
  $$Lambda$5405/1534932780@383f2392
questionB: Long => scala.concurrent.Future[(Long, List[Long])] =
  $$Lambda$5406/1081244036@6435c63c
res0: scala.concurrent.Future[Long] = Future(<not completed>

scala> res0
res1: scala.concurrent.Future[Long] = Future(Success(1))

scala>
```

You can now retrieve the created category, questions, and answers:

```
scala> categoryDao.all()
res2: scala.concurrent.Future[List[org.example.quiz.dao.records.Category]] =
  Future(Success(List(Category(1,Test)))

scala> qaDao.pickByCategoryId(1, n = 5)
res3:
  scala.concurrent.Future[Map[org.example.quiz.dao.records.Question, List[org.example.quiz.dao.records.Answer]]] = Future(Success(Map(Question(1,Is this a test?,1) ->
  List(Answer(1,1,True,true), Answer(3,1,False,false)), Question(2,Another test?,1) ->
  List(Answer(2,2,XXX,false), Answer(4,2,YYY,true), Answer(5,2,ZZZ,false))))
```

Finally, let's ensure that your application can delete a category, together with its questions and answers:

```
scala> categoryDao.deleteById(1)
```

```
res4: scala.concurrent.Future[Boolean] = Future(Success(true))

scala> qaDao.pickCategoryId(1, n = 5)
res5:
scala.concurrent.Future[Map[org.example.quiz.dao.records.Question, List[org.example.quiz.dao.records.Answer]]] = Future(Success(Map()))
```

The database automatically deletes any questions and answers that reference a deleted category id because of the “DELETE ON CASCADE” selected strategy.

47.6 Summary

In this capstone, you have implemented an application’s data access layer to create and play quizzes.

- You have implemented a query to check the connectivity to the database.
- You have represented each table with a case class.
- You have defined which operations your application can perform in each table.
- You have used your code to read and write records in the database.

Unit 8

JSON (De)Serialization

In unit 7, you have mastered how to represent asynchronous computations in Scala. In this unit, you'll discover how to serialize and deserialize data in JSON format and how lazily represent side effects. In the capstone, you'll complete the implementation of your quiz application by defining its business logic and HTTP API. In particular, you'll learn about the following topics:

- Lesson 48 shows you how to use a popular library called `circe` to convert a class into a JSON string and vice versa by defining JSON encoders and decoders.
- Lesson 49 teaches you about lazy evaluation and how it differs from eager evaluation. You'll see how by name parameters behave differently from by value ones. You'll also discover how to lazily initialize a value using the `lazy` keyword.
- Lesson 50 introduces you to the type `IO` from the library `cats-effect` to lazily evaluate both synchronous and asynchronous side effects.
- Lesson 51 shows you how to apply the `map` and `flatMap` operations on an `IO` instance to manipulate and transform its value. You'll also learn how to coordinate multiple side effects by running them in sequence or parallel.
- Lesson 52 introduces you to the basics of writing tests using a library called `ScalaTest` for both synchronous and asynchronous code.
- Finally, you'll define an HTTP server with an API that exposes and accepts data in JSON format in lesson 53. You'll implement an app that picks ten questions for a given category and calculate a score based on the received answers.

48

JSON (De)serialization with circe

After reading this lesson, you will be able to:

- Convert a data structure or Scala instance into JSON format.
- Parse JSON data as Scala code.

You have learned about asynchronous computations in the previous unit. In this lesson, you are going to learn about working with JSON in Scala. Scala doesn't offer JSON support natively, so you'll use circe, a popular library to work with JSON in Scala. JSON stands for "JavaScript Object Notation", and it is a commonly used lightweight format to exchange data that is straightforward for humans to read and for machines to parse. You'll see how to represent data in JSON format: you can refer to this process as serialization. You'll also discover the inverse operation of defining a Scala instance by parsing a JSON structure: people refer to this operation as deserialization. Figure 48.1 provides a visual comparison between the processes of JSON serialization and JSON deserialization.

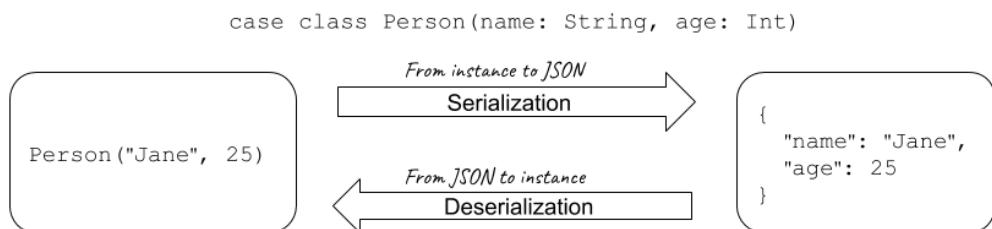


Figure 48.1: The process of serialization allows you to represent a Scala instance in JSON format. Its inverse is deserialization, which allows you to parse data in JSON format into a Scala instance.

In the capstone, your quiz application's HTTP API will send and receive data using the JSON format.

Consider this

Imagine you are developing a weather application to analyze how the temperature changes over time. It uses a third party's HTTP API to receive weather information daily on your areas of interest: it represents its data in JSON format. How would you parse it so that your application can process it?

48.1 Project Setup

Before you can learn how to work with JSON data, you need to include a JSON library as an external dependency in your project. First, create an empty sbt project and include the modules of the circe library you are going to use as external dependencies by adding the following to your build.sbt file:

Listing 48.1: Including the circe modules as external dependencies

```
// file build.sbt

// ...

val CirceVersion = "0.14.0-M4"

libraryDependencies ++= List(
  "io.circe" %% "circe-core" % CirceVersion,
  "io.circe" %% "circe-generic" % CirceVersion,
  "io.circe" %% "circe-parser" % CirceVersion
)
```

You can now launch sbt using the command `sbt console` to download the modules and load them into a Scala REPL session. You are now ready to discover how to use the circe library to serialize and deserialize data in JSON format.:.

48.2 JSON Serialization: from Instance to JSON

Imagine that you are developing a program to keep track of a person's full name and age. Your program exposes this information via an HTTP API to retrieve the person's details with a given id. Listing 48.3 shows you how to define a conversion to JSON for your person representation using the library circe and the type class pattern:

Listing 48.2: Converting Person to JSON

```
import java.time.{LocalDate, Period}

import io.circe._ ①

case class Person(fullName: String, dateOfBirth: LocalDate)

object Person {
```

```

given personEncoder: Encoder[Person] with { ❷
  def apply(p: Person): Json = {
    val age = Period.between(p.dateOfBirth, LocalDate.now()).getYears ❸
    Json.obj(
      "fullName" -> Json.fromString(p.fullName),
      "age" -> Json.fromInt(age)
    )
  }
}

```

❶ Importing the circe library

❷ The trait Encoder defines how to convert an instance into a JSON object

❸ Computing the age based on today's date and their date of birth

Your encoder calculates it on the fly from the person's date of birth when converting an instance into JSON to ensure that the field age is always up to date. After defining an encoder, you can import the package `io.circe.syntax._` and use its `asJson` function to perform the encoding. The method `asJson` has the following (simplified) signature:

```
def asJson[A](using encoder: Encoder[A]): Json
```

You can use it to convert a `Person` instance to JSON in the following way:

```

scala> import io.circe.syntax._
import io.circe.syntax.__
// it includes several helper functions such as asJson

scala> val p = Person("John Doe", LocalDate.of(1987, 11, 22))
val p: Person = Person(John Doe,1987-11-22)

scala> val json = p.asJson
val json: io.circe.Json =
{
  "fullName" : "John Doe",
  "age" : 32
}
// it returns an instance of type io.circe.Json

scala> json.toString
val res1: String =
{
  "fullName" : "John Doe",
  "age" : 32
}
// it returns an instance of type String

```

The library `circe` uses the type class pattern to define a conversion to JSON for any instance. You can convert any type `T` to JSON that has an instance of `Encoder[T]`. It has a predefined set of encoders for basic types, ready for you to use:

```

scala> import io.circe.syntax._
import io.circe.syntax.__

scala> List("Hello", "World").asJson
val res0: io.circe.Json =

```

```
[  
  "Hello",  
  "World"  
]  
  
scala> Map(1 -> "Scala").asJson  
val res1: io.circe.Json =  
{  
  "1" : "Scala"  
}  
  
scala> import java.time.LocalDate  
import java.time.LocalDate  
  
scala> LocalDate.of(1985, 5, 12).asJson  
val res2: io.circe.Json = "1985-05-12"
```

When using case classes, you can request to generate an encoder for them automatically. Listing 48.4 shows you how to achieve this for the case class `Person`:

Listing 48.3: Deriving an encoder for Person

```
import java.time.LocalDate  
  
import io.circe._  
import io.circe.generic.semiauto._ ①  
  
case class Person(fullName: String, dateOfBirth: LocalDate)  
  
object Person {  
  
  given personEncoder: Encoder[Person] = deriveEncoder[Person] ②  
}
```

① Importing the package that contains the encoder generation

② Derivation an encoder based on the structure of the case class

The generated encoder analyzes the structure of a case class to define a new encoder. For example, the derived instance for `Encoder[Person]` produces the following JSON representation: it does not contain the logic to compute the person's age, but it shows the date of birth.

```
scala> import io.circe.syntax._  
import io.circe.syntax._  
  
scala> val p = Person("John Doe", LocalDate.of(1987, 11, 22))  
val p: Person = Person(John Doe,1987-11-22)  
  
scala> p.asJson  
val res0: io.circe.Json =  
{  
  "fullName" : "John Doe",  
  "dateOfBirth" : "1987-11-22"  
}
```

QUICK CHECK 48.1

The library circe uses the ISO-8601 format to convert a local date to its text representation by default:

```
scala> import java.time.LocalDate
import java.time.LocalDate

scala> import io.circe.syntax._
import io.circe.syntax._

scala> LocalDate.of(1981, 7, 25).asJson
val res0: io.circe.Json = "1981-07-25"
```

Define a new encoder for `LocalDate` to convert dates using the following date formatter instead:

```
import java.time.format.DateTimeFormatter
val formatter = DateTimeFormatter.ofPattern("MM/dd/yyyy")
```

HINT: You can use the function `formatter.format(myDate)` to convert an instance `myDate` of type `LocalDate` to `String`.

48.3 JSON deserialization: from JSON to Instance

Let's consider your application to track people's names and ages and imagine it also offers an HTTP API to send new records using the following JSON structure:

```
{ "fullName" : "John Doe", "dateOfBirth" : "1987-11-22" }
```

Listing 48.5 shows you how to define a conversion from a JSON structure to a `Person` instance:

Listing 48.4: Converting JSON to Person

```
import java.time.LocalDate
import io.circe._ ①

case class Person(fullName: String, dateOfBirth: LocalDate)

object Person {

  given personDecoder: Decoder[Person] with { ②

    def apply(c: HCursor): Either[DecodingFailure, Person] = ③
      for {
        fullName <- c.downField("fullName").as[String] ④
        dateOfBirth <- c.downField("dateOfBirth").as[LocalDate] ⑤
      } yield Person(fullName, dateOfBirth)
  }
}
```

① Importing the circe library

② The trait Decoder defines how to parse a JSON object into an instance

③ The parsing that either result in a decoding failure or return a Person instance

④ Finding a field with name `fullName` and parsing it as `String`

⑤ Selecting a field with name `dateOfBirth` and converting it to `LocalDate`

The function `downField` selects a JSON component by a given name. Then, the function `as[T]` uses the decoder for `T` to parse and produce a value of its type. Once you have defined a decoder, you can import the package `io.circe.parser._`, and use its `decode` function to produce an instance representing the JSON structure. The function `decode` has the following (simplified) signature:

```
def decode[A](input: String)(using decoder: Decoder[A]): Either[Error, A]
```

It returns a value of type `Person` if the parsing is successful. Otherwise, it returns an error. A few examples of how to use the `decode` function are the following:

```
scala> import io.circe.parser._
import io.circe.parser._

scala> val goodData: String =
""" { "fullName": "John Doe", "dateOfBirth": "1987-11-22" } """

val goodData: String = " { \"fullName\": \"John Doe\", \"dateOfBirth\": \"1987-11-22\" } "
// Using triple quotes when defining a String allows you
// to automatically escape any special character, such as " or \"

scala> decode[Person](goodData)
val res0: Either[io.circe.Error,Person] = Right(Person(John Doe,1987-11-22))
// The decoding was successful!

scala> val badData: String = """ { "fullName": "John Doe" } """
val badData: String = " { \"fullName\": \"John Doe\" } "

scala> decode[Person](badData)
val res1: Either[io.circe.Error,Person] = Left(DecodingFailure(Attempt to decode value on failed
cursor, List(DownField(dateOfBirth))))
// The decoding fails because the field dateOfBirth is missing
```

The library `circe` offers a set of predefined decoders for commonly used types:

```
scala> import io.circe.parser._
import io.circe.parser._

scala> decode[List[String]](""" ["Hello", "World"] """)
val res0: Either[io.circe.Error,List[String]] = Right(List>Hello, World))

scala> decode[Map[Int, String]](""" { "1": "Scala" } """)
val res1: Either[io.circe.Error,Map[Int,String]] = Right(Map(1 -> Scala))

scala> import java.time.LocalDate
import java.time.LocalDate

scala> decode[LocalDate]("1985-05-12")
val res2: Either[io.circe.Error,java.time.LocalDate] = Left(io.circe.ParsingFailure: expected
whitespace or eof got '1985-05-12' (line 1, column 5))
// the text '1985-05-12' is not a valid JSON!

scala> decode[LocalDate](" 1985-05-12  ")
val res3: Either[io.circe.Error,java.time.LocalDate] = Right(1985-05-12)
// the text ' 1985-05-12  ' is a valid JSON representation for LocalDate
```

When using case classes, circe can automatically infer a decoder by analyzing their structure. Listing 48.6 shows you how to do this:

Listing 48.5: Deriving a decoder for Person

```
import java.time.LocalDate

import io.circe._  
import io.circe.generic.semiauto._ ①

case class Person(fullName: String, dateOfBirth: LocalDate)

object Person {  
  
    given personDecoder: Decoder[Person] = deriveDecoder[Person] ②  
}
```

① Importing the package that contains the decoder generation

② Derivation decoder based on the structure of the case class

The decoder circe has generated in listing 48.6 is equivalent to the one you have implemented in listing 48.5.

QUICK CHECK 48.2

The library circe uses the ISO-8601 format to decode a JSON string into a local date instance by default:

```
scala> import java.time.LocalDate
import java.time.LocalDate

scala> import io.circe.parser._
import io.circe.parser._

scala> decode[LocalDate]( """ "1981-07-25" """)
val res0: Either[io.circe.Error,java.time.LocalDate] = Right(1981-07-25)
```

Define a new decoder for `LocalDate` to convert to dates using the following date formatter instead:

```
import java.time.format.DateTimeFormatter
val formatter = DateTimeFormatter.ofPattern("MM/dd/yyyy")
```

HINT: You can use the method `LocalDate.parse(myDate, formatter)` that converts an instance `myDate` of type `String` to `LocalDate`.

48.4 Summary

In this lesson, my objective was to show you how to use the library circe to serialize and deserialize data into JSON format.

- You have learned how to define an encoder to convert a Scala instance into a JSON structure.
- You have discovered how to define a decoder to generate a Scala instance from a JSON representation.
- You have seen how you can automatically derive encoders and decoders for case classes.

Let's see if you got this!

TRY THIS

Consider the following case class to represent a book:

```
case class Book(title: String, authors: List[String], genres: Set[String])
```

Serialize and deserialize its instances to JSON using the circe library.

48.5 Answers to Quick Checks

QUICK CHECK 48.1

The new encoder for `LocalDate` should look similar to the following:

```
import java.time.LocalDate
import io.circe._

given dateEncoder: Encoder[LocalDate] with {
  def apply(date: LocalDate): Json =
    Json.fromString(formatter.format(date))
}
```

You can now import the new encoder in your local implicit scope, and circe will use it when converting local dates:

```
scala> import io.circe.syntax._
import io.circe.syntax._

scala> LocalDate.of(1981,7,25).asJson
val res0: io.circe.Json = "25/07/1981"
```

QUICK CHECK 48.2

A possible implementation for the local date decoder is the following:

```
import java.time.LocalDate
import io.circe._

given dateEncoder: Encoder[LocalDate] with {
  def apply(date: LocalDate): Json =
    Json.fromString(formatter.format(date))
}
```

After adding the decoder into your implicit scope, you can use it to convert to dates using the new format:

```
scala> import io.circe.parser._
import io.circe.parser._

scala> decode[LocalDate]("11/22/1987")
val res0: Either[io.circe.Error, java.time.LocalDate] = Right(1987-11-22)
```

49

Lazy Evaluation

After reading this lesson, you'll be able to:

- Implement functions that use by name parameters
- Initialize a value lazily by using the keyword `lazy`

After mastering JSON serialization and deserialization, you'll learn about lazy evaluation. You have already encountered examples of lazy evaluations when discussing Scala collections. For example, the function `getOrElse(key: K, default: => V)` acting on an instance of type `Map[K,V]` has two parameters: `key`, which is eagerly evaluated (or called by value), and `default`, which is lazily evaluated (or called by name). In this lesson, you'll see how to evaluate parameters by name (i.e., lazy evaluation) rather than by value (i.e., eager evaluation). You'll also learn how to initialize values lazily using the keyword `lazy`: this can be useful when its initialization is expensive either in terms of time or resources. In the capstone, you'll implement the logic and API layer of your quiz application using a type called `IO`, which allows you to represent computations lazily.

Consider this

Imagine you are developing a program that relies on a third-party HTTP API to validate postcodes. Unfortunately, network issues outside of your control can cause errors in your application, and you'd like to implement an automatic retry system for any of those known-to-be-problematic HTTP calls. How would you implement this?

49.1 By Name Parameters

Suppose you are developing a game in which players roll one 6-sided die to determine the outcome of actions they wish to perform: they succeed if they roll four or higher. Its implementation is the following:

```
import scala.util.Random

def rollDie(): Int = {
    // selecting random number between 1 inclusive and 7 exclusive
    val n = Random.nextInt(6) + 1
    println(s"Rolled $n...")
    if (n < 4) throw new IllegalStateException(s"Failure! Rolled $n")
    else n
}
```

On some occasions, players can re-roll the die up to two more times. Listing 49.1 shows you a possible implementation for it:

Listing 49.1: Re-rolling die: eager evaluation

```
import scala.annotation.tailrec *
import scala.util.Try

@tailrec ❶
def retry[T](n: Int, operation: T): Try[T] = { ❷
    val result = Try(operation) ❸
    if (result.isFailure && n > 0) retry(n - 1, operation)
    else result
}
```

- ❶ Using the `@tailrec` annotation to ensure that the function is tail-recursive
- ❷ Evaluating `operation` by value
- ❸ Wrapping any exception the function may throw

The annotation `@tailrec` ensures that the recursive function tail-recursive, which means it is safe to use as the compiler can optimize it at compile time not to exhaust its function call stack. It will produce a compilation error if it detects that your function is not tail-recursive. Let's call the `retry` function to re-roll the die and see if it behaves as expected:

```
scala> retry(n = 2, rollDie())
Rolled 4...
res0: scala.util.Try[Int] = Success(4)
// Success! The first roll gave you a 4, so no re-roll is needed.

scala> retry(n = 2, rollDie())
Rolled 1...
java.lang.IllegalStateException: Failure! Rolled 1
  at .rollDie(<console>:16)
  ... 28 elided
// what? Something doesn't look right...
```

Something unexpected happens: your program doesn't re-roll the die for numbers lower than 4, and the exception that `rollDie` throws leaks outside your function rather than be contained inside the `Try` class.

By default, Scala evaluates a function parameter by value (i.e., eager evaluation): it computes its value and then passes it to the function. However, there are cases in which you may want to evaluate a parameter when the function invokes it, not before: you can refer to this as parameter evaluated by name (i.e., lazy evaluation). Use the symbol `=>` to indicate that your program should evaluate a parameter by name. When declaring that your code should evaluate a parameter `T` by

name, the compiler transforms it to a function parameter that takes no parameters, and it returns a value of type T (i.e., with shape $() \Rightarrow T$). Listing 49.2 shows you the correct implementation for the `retry` function:

Listing 49.2: Re-rolling die: lazy evaluation

```
import scala.annotation.tailrec
import scala.util.Try

@tailrec
def retry[T](n: Int, operation: () => T): Try[T] = { ①
  val result = Try(operation)
  if (result.isFailure && n > 0) retry(n -1, operation)
  else result
}
```

① Evaluating operation by name

The `retry` function now behaves as expected:

```
scala> retry(n = 2, rollDie())
Rolled 1...
Rolled 5...
res0: scala.util.Try[Int] = Success(5)
// Success on the second try

scala> retry(n = 2, rollDie())
Rolled 6...
res1: scala.util.Try[Int] = Success(6)
// Success on the first try

scala> retry(n = 2, rollDie())
Rolled 3...
Rolled 1...
Rolled 2...
res2: scala.util.Try[Int] = Failure(java.lang.IllegalStateException: Failure! Rolled 2)
// A bit of bad luck: failure even after two re-rolls
```

Have a look at figure 49.1 for the execution flow comparison between eager and lazy evaluations of parameters.

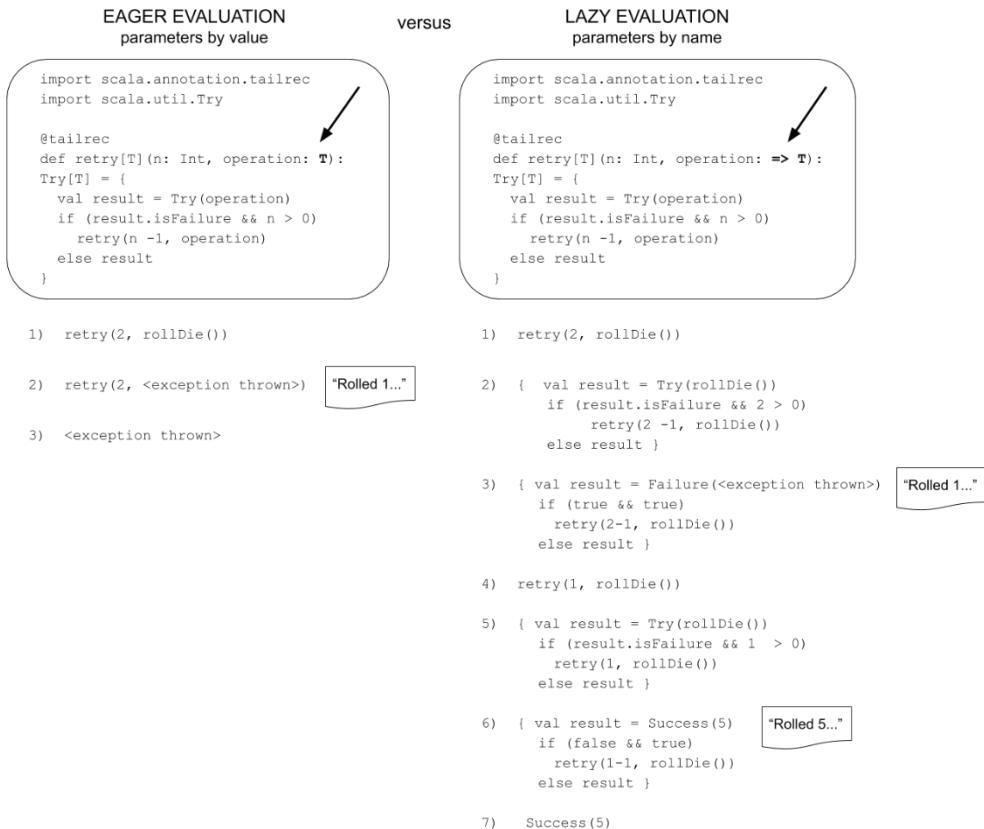


Figure 49.1: Execution flow comparison between eager and lazy evaluation. When passing the parameter by value (i.e., eagerly evaluating it), your program will first evaluate it, then use it in the function. When using a parameter by name (i.e., lazily evaluating it), your program will first pass it to the function, then evaluate it every time it is invoked.

QUICK CHECK 49.1

Consider the following functions: what is the behavior of their function calls? Use the REPL to validate your hypotheses.

```
def fooByValue(n: Int): Int = n + n
fooByValue { println("Scala"); 21 }

def fooByName(n: => Int): Int = n + n
fooByName { println("Scala"); 21 }
```

49.2 Lazy Values

Imagine that the game you are developing gives players access to statistics about their past games. As your game becomes more and more popular, regular users start complaining about long

loading times. After some investigation, you discover that the player statistics' initialization takes a long time. Your code looks similar to the following:

Listing 49.3: Eagerly loading player stats

```
class Stats(playerId: Long) {
    /* some meaningful stats loaded here */

    // Sleeping here to simulate a slow operation
    Thread.sleep(10000) // 10 seconds
}

class Player(id: Long, name: String) {
    val stats: Stats = new Stats(id)
}
```

The initialization of each player now takes at least 10 seconds:

```
scala> import java.time._
|   val start = Instant.now()
|   val daniela = new Player(1, "Daniela")
|   val duration = Duration.between(start, Instant.now())
|   println(s"Took ${duration.getSeconds} seconds!")

Took 10 seconds!
start: java.time.Instant = 2020-10-03T13:30:48.250Z
daniela: Player = Player@f53ebe2
duration: java.time.Duration = PT10.01S
```

Scala eagerly evaluates values: your program initializes the field `stats` as part of the `Player` instance's initialization. A possible solution is to run `stats` as an asynchronous computation by wrapping it into a `Future` type. Another solution is to request the lazy evaluation for the field `stats` using the keyword `lazy`:

Listing 49.4: Lazily loading player stats

```
class Stats(playerId: Long) {
    /* some meaningful stats loaded here */

    // Sleeping here to simulate a slow operation
    Thread.sleep(10000) // 10 seconds
}

class Player(id: Long, name: String) {
    lazy val stats: Stats = new Stats(id) ①
}
```

① It is evaluated the first time it is invoked, not during initialization

Your program delays the evaluation of lazy values until their first invocation. The initialization of each player is no longer slow:

```
scala> import java.time._
|   val startA = Instant.now()
|   val daniela = new Player(1, "Daniela")
```

```

| val durationA = Duration.between(startA, Instant.now())
| println(s"Took ${durationA.getSeconds} seconds!")

Took 0 seconds!
import java.time._
startA: java.time.Instant = 2020-10-03T13:32:10.343Z
daniela: Player = Player@3cfc3dd0
durationA: java.time.Duration = PT0.001S

```

Loading the player statistics still takes 10 seconds, but your program does this the first time it invokes that field:

```

scala> import java.time.__
| val startB = Instant.now()
| daniela.stats
| val durationB = Duration.between(startB, Instant.now())
| println(s"Took ${durationB.getSeconds} seconds!")

Took 10 seconds!
import java.time.__
startB: java.time.Instant = 2020-10-03T13:32:56.068Z
durationB: java.time.Duration = PT10.005S

```

Figure 49.2 provides a syntax diagram of how to declare lazy values.

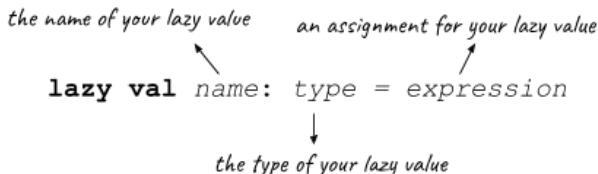


Figure 49.2: Syntax Diagram of lazy values in Scala. Your program delays the initialization of a lazy value until its first invocation.

QUICK CHECK 49.2

Consider the following snippet of code. What does it return? Why? What would happen if it wasn't lazy? Use the REPL to validate your hypothesis. Note: make sure to evaluate all the instructions at the same time by copying and pasting the whole snippet of code in the REPL.

```

lazy val a = b
val b = "hello"
a.length

```

49.3 Summary

In this lesson, my objective was to compare eager and lazy evaluation in Scala.

- You have learned how to pass parameters to function by value rather than by name.
- You have seen how to delay the initialization of a value to its first invocation using the keyword `lazy`.

Let's see if you got this!

TRY THIS

In this lesson, you have seen how to measure duration in seconds of two specific operations: initializing a player and retrieving its game statistics. Reuse the same logic to define a function, called `timed`, to measure the execution time in seconds of any operation.

49.4 Answers to quick checks

QUICK CHECK 49.1:

Both function calls return the value 42. However, the function `fooByName` prints the text "Scala" to the console twice, while `fooWithValue` does it only once.

```
scala> def fooWithValue(n: Int): Int = n + n
fooWithValue: (n: Int)Int

scala> fooWithValue { println("Scala"); 21 }
Scala
res0: Int = 42

scala> def fooByName(n: => Int): Int = n + n
fooByName: (n: => Int)Int

scala> fooByName { println("Scala"); 21 }
Scala
Scala
res1: Int = 42
```

QUICK CHECK 49.2:

The snippet of code returns the number 5:

```
scala> lazy val a = b
|   val b = "hello"
|   a.length

a: String = <lazy>
b: String = hello
res0: Int = 5
```

The REPL initially does not assign a value to `a`, but it displays the message `a: String = <lazy>`. Then, it assigns `b` to the text "hello". It eventually initializes the value `a` when executing the instruction `a.length`, which returns the integer 5. If you do not mark the value `a` as `lazy`, the snippet of code throws a `NullPointerException`:

```
scala> val a = b
|   val b = "hello"
|   a.length

<pastie>:11: warning: Reference to uninitialized value b
val a = b
^
java.lang.NullPointerException
... 36 elided
```

50

The IO Type

After reading this lesson, you will be able to:

- Represent synchronous and asynchronous side effects lazily.
- Execute and process their result

After mastering the difference between eager and lazy evaluation, you are going to learn about cats.effect.IO. Impure functions produce side effects. The type IO, which is part of the cats-effect library, allows you to represent synchronous and asynchronous side effects lazily: you'll be able to separate the definition of what to execute from its actual execution, making it easier to maintain and test. You can consider the type IO as the lazy alternative to the eagerly evaluated Future. After discussing why lazily evaluating side effects can be advantageous, you'll see how to represent and execute side effects that are either synchronous or asynchronous. In the capstone, you'll use the type IO to define side effects in your quiz application.

Consider this

Imagine you are developing an application that accesses the list of transactions of a bank account. You need to log the date, the name of the user executing the operation, and the account number for security reasons. You'd like to write tests to ensure this happens as expected. How would you implement this?

50.1 Why IO?

In the previous unit, you have learned that the type `Future` can improve your application's scalability and runtime performance by running expensive computations in the background. With time and experience, you will notice some of its stylistic disadvantages.

Most of your functions will return a value of type `Future` for good reasons. You do not want to block your program waiting for the result of your asynchronous computation unless strictly

necessary to gain the maximum performance benefit: functions that invoke functions returning `Future` values often return `Future` instances themselves. Most of them will also have an implicit `ExecutionContext` parameter. Your program eagerly evaluates `Future` value (i.e., it starts their execution as soon as it initializes them), so it must know how to run them, such as how many threads and retry policy to use.

`Futures` can also be challenging to understand. For example, consider the following two functions:

Listing 50.1: Future is not referentially transparent

```
import scala.concurrent.{ExecutionContext, Future}

def fooA(using ec: ExecutionContext): Future[Int] = {
  def future = Future { println("Scala"); 5 } ①
  for {
    a <- future
    b <- future
  } yield a + b
}

def fooB(using ec: ExecutionContext): Future[Int] = {
  val future = Future { println("Scala"); 5 } ②
  for {
    a <- future
    b <- future
  } yield a + b
}
```

① `future` is a function: you re-initialize it at every invocation

② `future` is value: you initialize it only once

The two functions are almost identical: `future` is a function in `fooA`, while it is a value in `fooB`. This minor detail drastically changes the two functions' behavior: `fooA` prints the text "Scala" twice, while `fooB` does it only once.

```
scala> fooA
Scala
Scala
res0: scala.concurrent.Future[Int] = Future(<not completed>)

scala> fooB
Scala
res1: scala.concurrent.Future[Int] = Future(<not completed>)
```

The type `Future` is not referentially transparent: you cannot replace its invocation with its returned value and be confident you will obtain the same result.

The type `IO` provides a solution to these problems by providing an alternative way to represent side effects lazily, rather than eagerly. You evaluate its instances on-demand, rather than at initialization: you'll create a description of your computation (i.e., *what* to do), and then you are going to execute it by invoking a function in which you provide information on what resources to use (i.e., *how* to do it). The type `IO` is also referentially transparent, thanks to its clear separation between description and execution. Consider the following functions:

Listing 50.2: IO is referentially transparent

```
import cats.effect.IO

def fooA: IO[Int] = {
  def io = IO { println("Scala"); 5 } ①
  for {
    a <- io
    b <- io
  } yield a + b
}

def fooB: IO[Int] = {
  val io = IO { println("Scala"); 5 } ②
  for {
    a <- io
    b <- io
  } yield a + b
}
```

- ① `io` is a function
 ② `io` is a value

One is declaring `io` as a function, the other as a value, but this does not affect their behavior because the type `IO` is referentially transparent:

```
scala> val resultA = fooA
resultA: cats.effect.IO[Int] = IO$976545953
// This is just the description of the task

scala> resultA.unsafeRunSync()
Scala
Scala
res0: Int = 10
// Actually running the task

scala> val resultB = fooB
resultB: cats.effect.IO[Int] = IO$976545953
scala> resultB.unsafeRunSync()
Scala
Scala
res1: Int = 10
// fooA and fooB are equivalent!
```

The type `IO` is extremely powerful, and it has several other advantages. For example, the type `IO` is cancellable while `Future` is not. In this book, you'll learn only the basics about the `IO` type: have a look at their documentation at <https://typelevel.org/cats-effect> to learn more about what you can achieve with it.

50.2 Project Setup

The type `IO` is not a standard type of the Scala language: you need to add the cats-effect library as your dependency before using it. Let's create an empty sbt project by defining the `project/build.properties` and `build.sbt` files. They should look similar to the following:

Listing 50.3: An empty sbt project

```
// file project/build.properties

sbt.version = 1.4.4


// file build.sbt

name := "get-programming-wil-scala-lesson50"

version := "0.1"

scalaVersion := "2.13.4"
```

You can add the cats-effect library by adding the following configuration to your build.sbt file:

Listing 50.4: Adding cats-effect as an external dependency

```
// file build.sbt

libraryDependencies += "org.typelevel" %% "cats-effect" % "2.3.1"
```

Run the command `sbt console` in your project's root directory to launch sbt, download all its external dependencies, and load them into a Scala REPL session.

50.3 Synchronous Side Effect

Let's consider the function you have written to roll one 6-sided die for your game application in the previous lesson:

```
import scala.util.Random

def rollDie(): Int = {
    val n = Random.nextInt(6) + 1
    println(s"Rolled $n...")
    if (n < 4) throw new IllegalStateException(s"Failure! Rolled $n")
    else n
}
```

The function `rollDie` is impure because it has several side effects: it selects a random number, it prints a message to the console, and it throws an exception when the rolled number is not high enough. Listing 50.5 shows you how you can re-implement it using IO:

Listing 50.5: Describing how to roll die using IO.apply

```
import cats.effect.IO ①
import scala.util.Random

def rollDie(): IO[Int] = IO { ②
    val n = Random.nextInt(6) + 1
    println(s"Rolled $n...")
    if (n < 4) throw new IllegalStateException(s"Failure! Rolled $n")
    else n
}
```

① Adding the type `IO` to the scope

② Invoking the method `IO.apply` apply to represent the synchronous side effect: you can omit the `apply` function name.

When calling the `rollDie` method, you will describe the side effect without evaluating it. When ready to do so, you can call its `unsafeRunSync` method:

```
scala> val myRoll = rollDie()
myRoll: cats.effect.IO[Int] = IO$81277280
// The representation of the side effect

scala> myRoll.unsafeRunSync()
Rolled 4...
res0: Int = 4
// Rolling the die: the execution returns a 4

scala> myRoll.unsafeRunSync()
Rolled 2...
java.lang.IllegalStateException: Failure! Rolled 2
  at .$.anonfun$rollDie$1(<pastie>:7)
// Another roll: the execution throws an exception
```

When working with side effects, you can use the type `IO` to represent them by separating their description from their execution. The function `IO.apply` allows you to define a synchronous side effect. You can then use the method `unsafeRunSync` to run it synchronously. Listing 50.6 shows you their (simplified) signatures:

Listing 50.6: IO.apply and unsafeRunSync

```
package cats.effect

class IO[A] {
    def unsafeRunSync(): A = ???
}

object IO {
    def apply[A](body: => A): IO[A] = ???
}
```

All the methods that evaluate a side effect use the word “unsafe”: this reminds you that they are impure functions that produce side effects. Be mindful when invoking them: calling them one or multiple times will change your program’s behavior. Table 50.1 provides a summary of the methods you can use when representing synchronous side effects lazily.

Table 50.1: Summary of the main methods you can use the type IO when coding with synchronous side effects. The function apply initializes a synchronous side effect without executing it. The method unsafeRunSync will then allow you to evaluate it and produce a value as its result.

	Acts on	Signature	Usage
apply	IO	apply[A] (body: => A) : IO[A]	It describes a synchronous side effect lazily.
unsafeRunSync	IO[A]	unsafeRunSync() : A	It executes a side effect synchronously, and it returns its value.

QUICK CHECK 50.1

The function `scala.io.StdIn.readLine()` allows you to read text from the terminal. Implement a function called `read` to describe this side effect using the type `IO`. Use the Scala REPL to execute it and type your name.

50.4 Asynchronous Side Effect

Let's consider your game application again and its functionality of loading statics of the player's previous games you have seen in the previous lesson. Imagine you have the following function to compute the player's statistics:

```
class Stats(playerId: Long) {
    /* some meaningful stats loaded here */
    println(s"Loading statistics for player $playerId...")

    // Sleeping here to simulate a slow operation
    Thread.sleep(10000) // 10 seconds
}

object Stats {
    def load(playerId: Long): Stats = new Stats(playerId)
}
```

This operation is a side effect because its outcome depends on its historical data. It is also known to be slow at times, so you'd like to make it asynchronous. Listing 50.7 shows you how to achieve this:

Listing 50.7: Describing how to load the player's statistics

```
import cats.effect.IO ①

class Stats(val playerId: Long) {
    /* some meaningful stats loaded here */
    println(s"Loading statistics for player $playerId...")

    // Sleeping here to simulate a slow operation
    Thread.sleep(10000) // 10 seconds
}
```

```
object Stats {

    def load(playerId: Long): IO[Stats] = IO.async { callback => ②
        val either: Either[Throwable, Stats] = Right(new Stats(playerId))
        callback(either)
    }

}
```

① Adding the type `IO` to the scope

② Calling the function `IO.async` to represent the asynchronous side effect

Invoking the function `Stats.load` will give you a description of the side effect. You need to invoke one of the available functions to execute it. For example, you could use its method `unsafeRunAsync` in the following way:

```
scala> val myStats = Stats.load(1)
myStats: cats.effect.IO[Stats] = IO$287324394

scala> myStats.unsafeRunAsync {
    |   case Left(ex) => println(s"Error: $ex")
    |   case Right(_) => println(s"Success!")
    |
}
Loading statistics for player 1...
Success!
```

You can also use its function `unsafeRunAsyncAndForget` not to process its result further.

```
scala> myStats.unsafeRunAsyncAndForget()
Loading statistics for player 1...
```

You typically use this operation when computing side effects in the background that you are not interested in processing further, for example when saving data into a database.

You can use the function `IO.async` to describe a side effect that your program should execute asynchronously. You can then invoke one of its methods `unsafeRunAsync` and `unsafeRunAsyncAndForget` to evaluate it. Have a look at listing 50.8 for their (simplified) signature:

Listing 50.8: `IO.async` and its `unsafe` functions

```
package cats.effect

class IO[A] {

    def unsafeRunAsync(cb: Either[Throwable, A] => Unit): Unit = ???

    def unsafeRunAsyncAndForget(): Unit = ???

}

object IO {
    def async[A](k: (Either[Throwable, A] => Unit) => Unit): IO[A] = ???
}
```

Their signatures are relatively complex and will be confusing at first: let's try to clarify their meaning.

When invoking the method `IO.async`, you need to implement a function that receives a callback as its parameter: define the operation to run asynchronously by producing a value of type `Either`, and finally, feed it back to the callback.

```
val myIO = IO.async { callback =>
    // result is an instance of type Either!
    val result: Either[Throwable, A] = ??? // do your operation here
    callback(result)
}
```

When evaluating the side effect, you will receive the result that the last callback produced. The function `unsafeRunAsync` allows you process it on completion of its execution:

```
myIO.unsafeRunAsync {
    case Left(ex) => // do something with the error
    case Right(value) => // do something with the value
}
```

You can use its method `unsafeRunAsyncAndForget` if you do not wish to process its result:

```
myIO.unsafeRunAsyncAndForget()
```

Both the functions `unsafeRunAsync` and `unsafeRunAsyncAndForget` return `Unit`: their behavior is comparable to `Future.onComplete`. Have a look at table 50.2 for a recap of the methods you have seen to describe and evaluate asynchronous side effects lazily.

Table 50.2: Recap of the functions you can use to represent and evaluate asynchronous side effects using the type IO.

	Acts on	Signature	Usage
<code>async</code>	<code>IO</code>	<code>async[A](k: (Either[Throwable, A] => Unit) => Unit): IO[A]</code>	It describes an asynchronous side effect lazily.
<code>unsafeRunAsync</code>	<code>IO[A]</code>	<code>unsafeRunAsync[A](cb: Either[Throwable, A] => Unit): Unit</code>	It evaluates a side effect asynchronously and it uses a given callback to process its result before discarding its value.
<code>unsafeRunAsyncAndForget</code>	<code>IO[A]</code>	<code>unsafeRunAsyncAndForget(): Unit</code>	It runs a side effect asynchronously and it discards its value.

You can also use `unsafeRunSync` to execute an asynchronous side effect: this will run synchronously and give you a value in return. This approach can be useful when running tests but extremely inefficient in your production code because it impacts your application's runtime performance.

QUICK CHECK 50.2

Implement a function `convertToIO` to convert a `Future` instance into one of type `IO` using the `IO.async` function.

```
import cats.effect.IO
import scala.concurrent.{ExecutionContext, Future}

def convertToIO[T](future: => Future[T])
                  (using ec: ExecutionContext): IO[T]
```

50.5 Summary

In this lesson, my objective was to introduce you to the type `IO` from the `cats-effect` library.

- You have learned that representing side effects with a lazy approach can make your code less prone to errors.
- You have seen how to describe and execute synchronous impure computations using the methods `IO.apply` and `unsafeRunSync`.
- You have discovered how to represent and run asynchronous side effects using `IO.async`, `unsafeRunAsync`, and `unsafeRunAsyncAndForget`.

Let's see if you got this!

TRY THIS

Let's consider the following snippet of code that prints the list of files in the current directory:

```
import java.io.File
new File(".").listFiles().foreach(println)
```

In lesson 43, you have used `Future` to execute this operation asynchronously: implement it using `IO` instead.

50.6 Answers to Quick Checks

QUICK CHECK 50.1

The implementation for your `read` function should look like the following:

```
import cats.effect.IO

def read: IO[String] = IO(scala.io.StdIn.readLine())
```

You can execute it using its method `unsafeRunSync`:

```
scala> read.unsafeRunSync()
// the cursor waits for you to type,
// and it returns it after you press the enter key.
res0: String = Daniela
```

QUICK CHECK 50.2

A possible implementation for your function `convertToIO` is the following:

```
import cats.effect.IO

import scala.concurrent.{ExecutionContext, Future}
import scala.util.{Failure, Success}

def convertToIO[T](future: => Future[T])
                  (using ec: ExecutionContext): IO[T] =
  IO.async { callback =>
    future.onComplete {
      case Success(t) => callback(Right(t))
      case Failure(ex) => callback(Left(ex))
    }
  }
```

Note that the function passes the parameter `future` by name as you need to ensure your program executes it after receiving the callback, not before.

51

Working with the IO type

After reading this lesson, you will be able to:

- Manipulate values produced by side effects using the `map` and `flatMap` operations.
- Compute multiple `IO` instances in sequence using for-comprehension.
- Execute side effects in parallel to maximize the resource of your program.

In the previous lesson, you have learned the basics of the type `IO`. You'll now see how to compose smaller side effects to create programs that you can lazily describe and run. You'll master how to use the `map` and `flatMap` operations to manipulate values that impure functions produce. You'll discover how to combine them in sequence using for-comprehension. Also, you'll see how to run them in parallel using the `parSequence` method. In the capstone, you'll combine multiple `IO` instances to define the operations of your quiz application.

Consider this

Imagine your program has two functions using `IO` to print a message to the console and read from it. How would you compose these functions to create a program that asks for the names of your users and greet them accordingly?

51.1 The `map` and `flatMap` operations

The `IO` type offers implementations for the `map` and `flatMap` methods. They are consistent with those for `Future` and the other types you have encountered so far. Let's recap their usage in the following sections. You are going to code using the `IO` type: do not forget to add `cats-effect` as an external dependency for your sbt project by adding the following instruction to your `build.sbt` file:

```
libraryDependencies += "org.typelevel" %% "cats-effect" % "2.3.3"
```

51.1.1 The map function

Suppose you are developing a game that rolls one six-sided die using the following function:

```
import scala.util.Random
import cats.effect.IO

def rollDie: IO[Int] = IO(Random.nextInt(6) + 1)
```

You'd like to define a new function that instead returns a message to inform the user about the rolled die. Listing 51.1 should show how to use its `map` method to manipulate its value:

Listing 51.1: The outcome of a rolled die

```
import scala.util.Random
import cats.effect.IO

def rollDie: IO[Int] = IO(Random.nextInt(6) + 1)

def rollOutcome: IO[String] = rollDie.map(n => s"Rolled $n!")
```

The method `map` for the type `IO` allows you to transform its value by applying a given function. For an instance of `IO[A]`, the function `map` takes one parameter `f` of type `A => B`, and it produces a value of type `IO[B]`. It has the following signature:

```
def map[B](f: A => B): IO[B]
```

If your `IO[A]` produces a value of type `A` during its evaluation, it will apply the parameter `f` to it to produce a value of type `IO[B]`. Nothing happens if your instance of `IO[A]` errors during execution. A few examples of how to use it are the following:

```
scala> import cats.effect.IO
import cats.effect.IO

scala> val ioA = IO("Hello World!").map(_.length)
ioA: cats.effect.IO[Int] = <function1>

scala> ioA.unsafeRunSync()
res0: Int = 12

scala> val ioB = IO(5/0).map(_ + 1)
ioB: cats.effect.IO[Int] = <function1>

scala> ioB.unsafeRunSync()
java.lang.ArithmetricException: / by zero
  at $anonfun$ioB$1(<console>:1)
  ... 1 elided
```

QUICK CHECK 51.1

Define a function called `parseToInt` to parse a numeric text of type `IO[String]` and produce an instance of type `IO[Int]`.

```
import cats.effect.IO

def parseToInt(text: IO[String]): IO[Int] = ???
```

51.1.2 The flatMap function

Consider your game application again and consider that you'd like to define a function to roll a die twice and sum their results. Listing 51.2 shows you a possible implementation using the `flatMap` operation:

Listing 51.2: Rolling a die twice using flatMap

```
import cats.effect.IO
import scala.util.Random

def rollDie: IO[Int] = IO(Random.nextInt(6) + 1)

def rollDieTwice: IO[Int] = rollDie.flatMap { n1 =>
  rollDie.map(n2 => n1 + n2)
}
```

The operation `flatMap` combines the `map` and `flatten` methods to chain instances in a sequence. For an instance of `IO[A]`, the function `flatMap` takes one parameter `f` of type `A => IO[B]`, and it produces a value of type `IO[B]`. It has the following signature:

```
def flatMap[B](f: A => IO[B]): IO[B]
```

If your `IO[A]` produces a value of type `A` during its evaluation, it will apply the parameter `f` to it to produce a value of type `IO[B]`. Nothing happens if your instance of `IO[A]` errors during execution. A few examples of its usage are the following:

```
scala> import cats.effect.IO
import cats.effect.IO

scala> val ioA = IO("Hello World!).flatMap(n => IO(println(n.length)))
ioA: cats.effect.IO[Unit] = IO$1731931686

scala> ioA.unsafeRunSync()
12

scala> val ioB = IO(5/0).flatMap(n => IO(println(n + 1)))
ioB: cats.effect.IO[Unit] = IO$1948471530

scala> ioB.unsafeRunSync()
java.lang.ArithmetricException: / by zero
  at $anonfun$ioB$1(<console>:1)
... 1 elided
```

Does the type `IO` have a `flatten` operation?

The class `IO` does not have an implementation of the `flatten` function, but you can easily add it by adding the following import to your code:

```
import cats.syntax.flatMap._
```

For example, you can do the following:

```
import cats.effect.IO
import cats.syntax.flatMap._
```

```
def rollDieTwice: IO[Int] = rollDie.map { n1 =>
    rollDie.map(n2 => n1 + n2)
}.flatten
```

There is no need to explicitly implement a function `flatten` because you can re-implement it using `flatMap`:

```
def myFlatten[T](nestedIO: IO[IO[T]]): IO[T] =
```

```
    nestedIO.flatMap(x => x)
```

This concept of defining `flatten` as a special case of `flatMap` is applicable to any type with such operation, not just the class `IO`.

Table 51.1 provides a technical summary of the `map` and `flatMap` operation for the type `IO`.

Table 51.1: Summary of the main methods you can use the type `IO` when coding with synchronous side effects. The function `apply` initializes a synchronous side effect without executing it. The method `unsafeRunSync` will then allow you to evaluate it and produce a value as its result.

	Acts on	Signature	Usage
<code>map</code>	<code>IO[A]</code>	<code>map[B](f: A => B): IO[B]</code>	It applies a function to the value that the side effect produce.
<code>flatMap</code>	<code>IO[A]</code>	<code>flatMap[B](f: A => IO[B]): IO[B]</code>	It combines two side effects sequentially.

QUICK CHECK 51.2

Consider the following function to print a message to the console:

```
import cats.effect.IO
def printToConsole(msg: String): IO[Unit] = IO(println(msg))
```

Implement a function called `printRollOutcome` to print the outcome of a roll to the console by composing the `printToConsole` method with `rollOutcome` from listing 51.1.

51.2 For-comprehension

Consider the function you have implemented in listing 51.2 to roll a die twice. Listing 51.3 shows you an alternative implementation for it using for-comprehension:

Listing 51.3: Rolling a die twice using for-comprehension

```
import cats.effect.IO
import scala.util.Random

def rollDie: IO[Int] = IO(Random.nextInt(6) + 1)

def rollDieTwice: IO[Int] =
  for {
    n1 <- rollDie
    n2 <- rollDie
  } yield n1 + n2
```

```
n2 <- rollDie ②
} yield n1 + n2 ③
```

- ① Rolling the die the first time
- ② Rolling the die the second time
- ③ Combining the results

The type `IO` offers you implementations for the `map` and `flatMap` operations, so you can use for-comprehension as an alternative to express their combination in an ordered sequence. A few more examples are the following:

```
scala> import cats.effect.IO
import cats.effect.IO

scala> val myIo = for {
|   _ <- IO(println("Hello"))
|   _ <- IO(println("World!"))
| } yield ()
val myIo: cats.effect.IO[Unit] = IO$23096066

scala> myIo.unsafeRunSync()
Hello
World!
// First we print "Hello", then "World!"
```

QUICK CHECK 51.3

In Quick Check 51.2, you have implemented a function called `printRollOutcome`: refactor it to use for-comprehension.

51.3 Parallel Execution

Imagine that in the game you are developing, players need to throw four different dice types simultaneously. Listing 50.4 shows you how to represent this using the `parSequence` function for the `IO` type:

Listing 50.4: Rolling several dice at the same time

```
import cats.effect.{ContextShift, IO} ①
import cats.syntax.parallel._ ②

import scala.util.Random

def rollDie(n: Int): IO[Int] = IO { ③
  println(s"Rolling $n-side die...")
  Random.nextInt(n) + 1
}

def rollDice(using cs: ContextShift[IO]): IO[List[Int]] = { ④
  List(rollDie(6), rollDie(8), rollDie(12), rollDie(20)).parSequence ⑤
}
```

- ① Including the `IO` and `ContextShift` classes
- ② Adding the method `parSequence` into the scope
- ③ The `rollDie` function can now handle any sided dice.

- 4 The function `parSequence` requires a `ContextShift[IO]` implicit parameter.
- 5 The `parSequence` lazily executes a sequence of `IO` instances in parallel.

You can now define a `ContextShift[IO]` instance from your global execution context and execute the dice roll in parallel as the following:

```
scala> import scala.concurrent.ExecutionContext
import scala.concurrent.ExecutionContext

scala> given cs: ContextShift[IO] = IO.contextShift(ExecutionContext.global)
val cs: cats.effect.ContextShift[cats.effect.IO] = cats.effect.internals.IOContextShift@7766ce6b

scala> rollDice.unsafeRunSync()
Rolling 8-side die...
Rolling 6-side die...
Rolling 12-side die...
Rolling 20-side die...
val res0: List[Int] = List(5, 6, 9, 14)

scala> rollDice.unsafeRunSync()
Rolling 20-side die...
Rolling 6-side die...
Rolling 8-side die...
Rolling 12-side die...
val res1: List[Int] = List(3, 7, 11, 12)
```

The function `parSequence` allows you to execute a sequence of `IO` instances in parallel. For an instance of `List[IO[A]]`, the method `parSequence` requires an implicit context shift, and it returns a value of type `IO[List[A]]`. It will run the `IO` instances in parallel and collect its results in a list during its evaluation. You can see its behavior as analogous to `Future.sequence`. The class `ContextShift[IO]` is equivalent to `ExecutionContext`: it defines which resources are available to your program. You can define a context shift from an execution context instance by calling the `IO.contextShift` function:

```
import scala.concurrent.ExecutionContext
given cs: ContextShift[IO] = IO.contextShift(ExecutionContext.global)
```

A few more examples of how to use the `parSequence` methods are the following:

```
scala> import cats.effect.{ContextShift, IO}
| import cats.syntax.parallel._
| import scala.concurrent.ExecutionContext
| given cs: ContextShift[IO] =
IO.contextShift(ExecutionContext.global)
import cats.effect.{ContextShift, IO}
import cats.syntax.parallel._
import scala.concurrent.ExecutionContext
val cs: cats.effect.ContextShift[cats.effect.IO] = cats.effect.internals.IOContextShift@460d3361

scala> val ios = List(
|   IO(println("Hello")), IO(println("World!"))).parSequence
val ios: cats.effect.IO[List[Unit]] = <function1>

scala> ios.unsafeRunSync()
World!
Hello
```

```
val res0: List[Unit] = List(() , ())
scala> ios.unsafeRunSync()
Hello
World!
val res1: List[Unit] = List(() , ())
```

QUICK CHECK 51.4

Implement a function called `selectionAverage` to pick 100 random integers and compute their average:

```
import cats.effect.{ContextShift, IO}
def selectionAverage(using cs: ContextShift[IO]): IO[Double] = ???
```

Use the given value `randomNumber` to pick one random number between one and ten, and perform the number selection in parallel.

```
import cats.effect.IO
import scala.util.Random

val randomNumber: IO[Int] = IO(Random.nextInt(10) + 1)
```

HINT: The expression `(0 to 99).toList` returns a sequence of size 100 containing all the numbers from 0 to 99 inclusive.

51.4 Summary

In this lesson, my objective was to teach you some of the operations you can perform on lazily evaluated side effects.

- You have seen how to manipulate and transform values produced by side effects using the `map` and `flatMap` operations.
- You have learned how to evaluate `IO` instances in an ordered sequence using for-comprehension.
- You have discovered how you can run several independent side effects in parallel using the `parSequence` function.

Let's see if you got this!

TRY THIS

Using the type `IO`, define a small program that prints a message asking for the user's name, reads it from the console, and displays a personalized greeting message in return.

51.5 Answers to Quick Checks

QUICK CHECK 51.1

A possible implementation for your function `parseToInt` is the following:

```
import cats.effect.IO
def parseToInt(text: IO[String]): IO[Int] = text.map(_.toInt)
```

QUICK CHECK 51.2

Your `printRollOutcome` function should look similar to the following:

```
import cats.effect.IO

def printRollOutcome: IO[Unit] = rollOutcome.flatMap(printToConsole)
```

QUICK CHECK 51.3

You can refactor your `printRollOutcome` function as the following:

```
import cats.effect.IO

def printRollOutcome: IO[Unit] =
  for {
    n <- rollOutcome
    res <- printToConsole(n)
  } yield res
```

QUICK CHECK 51.4

You should implement the function `selectionAverage` as the following:

```
import cats.effect.{ContextShift, IO}
import cats.syntax.parallel._

import scala.util.Random

val randomNumber: IO[Int] = IO(Random.nextInt(10) + 1)

def selectionAverage(using cs: ContextShift[IO]): IO[Double] =
  (0 to 49).toList.map(_ => randomNumber).parSequence.map { numbers =>
  1.0 * numbers.sum / numbers.size
}
```

You need to provide a `ContextShift[IO]` instance to run it:

```
import scala.concurrent.ExecutionContext
given cs: ContextShift[IO] = IO.contextShift(ExecutionContext.global)

selectionAverage.unsafeRunSync()
// val res0: Double = 5.92
```

52

Testing with ScalaTest

After reading this lesson, you'll be able to:

- Write and run tests using ScalaTest.
- Test synchronous and asynchronous code.

Throughout this book, you have learned how to create a fully-working Scala application that connects to a database, processes data asynchronously, and exposes an HTTP API. Your application must include tests to check that it behaves as expected, implements its business requirements correctly, and prevents the introduction of bugs with future code changes. This lesson completes the picture by introducing you to the basics of writing tests in Scala. You'll write tests using a popular library called ScalaTest, and you'll run them using sbt. After writing and running your first test, you'll learn how to test code that runs asynchronously. In the capstone, you'll write tests for your quiz application to check the correctness of its business logic.

Consider this

Suppose you have written an application that analyzes weather data exposed by a third-party API. You have sample data and expected results to use during development. How would you write tests to check that your implementation behaves as expected and prevent new bugs with future code changes?

52.1 Project Setup

The first step in writing tests in Scala is to pick a testing library to use. In this lesson, you'll use ScalaTest, which is among the most popular ones. Define an empty sbt project, and specify it as one of your library dependencies in your build.sbt file by adding the following configuration:

Listing 52.1: Adding ScalaTest as an external testing dependency

```
// file build.sbt

libraryDependencies +=  
"org.scalatest" %% "scalatest" % "3.2.5" % "test" ①
```

① Declaring the dependency restricted to test only

The extra configuration `% "test"` indicates to use the dependency only in test code: this prevents using the library in your application's source code.

When executing the command `sbt test` in the root folder of your project, sbt will download your external dependencies, compile both the source and test code, and search for tests to run:

```
$ sbt test  
// ...  
[info] Run completed in 56 milliseconds.  
[info] Total number of tests run: 0  
[info] Suites: completed 0, aborted 0  
[info] Tests: succeeded 0, failed 0, canceled 0, ignored 0, pending 0  
[info] No tests were executed.  
[success] Total time: 1 s, completed 4 Feb 2021, 12:57:19
```

You haven't written any tests yet, so it executes none.

52.2 Your first test

Suppose that you have written a function that computes the frequency of the characters in a text. For example, for the text "test", it should count the character 't' twice and the others once. The function is called `count` and lives in a class called `org.example.frequency.Frequency`:

Listing 52.2: The count function

```
// file src/main/scala/org/example/frequency/Frequency.scala ①

package org.example.frequency

class Frequency {

    def count(text: String): Set[(Char, Int)] =  
        text.groupBy(char => char).map { case (char, occurrences) =>  
            char -> occurrences.length  
        }.toSet
}
```

① It is a source file because inside the directory src/main

Let's write a test for it. First, let's create a test file under `src/test` using the same package name and define which scenarios you want to verify:

Listing 52.3: The skeleton for FrequencyTest

```
// file src/test/scala/org/example/frequency/FrequencyTest.scala ①
package org.example.frequency

import org.scalatest.flatspec.AnyFlatSpec ②

class FrequencyTest extends AnyFlatSpec {

  "Frequency" should "count no frequency for the empty string" in fail() ③
  it should "count the frequency of the characters in a text" in fail() ③
}
```

- ① It is a test file because inside the directory `src/test`
- ② It enables the test specific Domain Specific Language (DSL)
- ③ The code elements `it`, `should`, `in`, `fail()` are part of the test DSL

The command `sbt test` detects the two tests as failed because they throw a `NoImplementationError` exception:

```
$ sbt test
// ...
[info] FrequencyTest:
[info] Frequency
[info] - should count no frequency for the empty string *** FAILED ***
[info]   org.scalatest.exceptions.TestFailedException was thrown. (FrequencyTest.scala:7)
[info] - should count the frequency of the characters in a text *** FAILED ***
[info]   org.scalatest.exceptions.TestFailedException was thrown. (FrequencyTest.scala:9)
[info] Run completed in 535 milliseconds.
[info] Total number of tests run: 2
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 0, failed 2, canceled 0, ignored 0, pending 0
[info] *** 2 TESTS FAILED ***
[error] Failed tests:
[error] org.example.frequency.FrequencyTest
[error] (Test / test) sbt.TestsFailedException: Tests unsuccessful
[error] Total time: 2 s, completed 5 Feb 2021, 11:18:38
```

You can now implement the tests and check if they pass:

Listing 52.4: The class FrequencyTest

```
// file src/test/scala/org/example/frequency/FrequencyTest.scala

package org.example.frequency

import org.scalatest.flatspec.AnyFlatSpec ①
import org.scalatest.matchers.should.Matchers ②

class FrequencyTest extends AnyFlatSpec with Matchers { ②

  val frequency = new Frequency ③

  "Frequency" should "count no frequency for the empty string" in {
    val result = frequency.count("")
    result.shouldEqual(Set.empty) ④
  }
}
```

```

it should "count the frequency of the characters in a text" in {
  val result = frequency.count("test")
  val expected = Set('t' -> 2, 'e' -> 1, 's' -> 1)
  result.shouldEqual(expected) ④
}
}

```

- ① It enables the test specific Domain Specific Language (DSL)
- ② It enables a DSL to express assertions using the verb should
- ③ The instance of type Frequency to test
- ④ The method shouldEqual is added to the scope by the trait Matchers

You can now use the command `sbt test` to check that the tests are now passing:

```

$ sbt test
// ...
[info] FrequencyTest:
[info] Frequency
[info] - should count no frequency for the empty string
[info] - should count the frequency of the characters in a text
[info] Run completed in 401 milliseconds.
[info] Total number of tests run: 2
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 6 s, completed 5 Feb 2021, 11:37:53

```

The ScalaTest library allows you to write tests for your Scala code. First, you need to define a class that extends one of its specification classes (e.g., `org.scalatest.flatspec.AnyFlatSpec`). Then, you need to select one of its matchers (e.g., `org.scalatest.Matchers`) to define the style of your assertions. ScalaTest is a rich and flexible library: its website www.scalatest.org offers a comprehensive list and guide on the several supported testing styles.

QUICK CHECK 52.1

You have received two bug reports on your implementation of `count`.

1. The function is incorrectly counting characters that are not letters nor digits (e.g., white space, punctuation, etc.): you should ignore them.
2. Also, it should consider each character as case insensitive. For example, it should add the occurrences of the character 'A' to those of the character 'a'.

For example, the text "Test test!" should return the following count (and nothing else!): the character 't' four times, the characters 'e' and 's' twice. Write new test scenarios to replicate the bugs and fix them.

52.3 Asynchronous Testing

Suppose your program needs to compute the frequency of the characters in a file. Your implementation reuses the function `count` that you have defined previously:

Listing 52.5: The function countFromFile

```
package org.example.frequency

import scala.concurrent.{ExecutionContext, Future}
import scala.io.Source

class Frequency {

    def countFromFile(filename: String)
        (using ec: ExecutionContext): Future[Set[(Char, Int)]] =
    Future[Set[(Char, Int)]] =
        readFromFile(filename).map(count)

    private def readFromFile(filename: String)
        (using ec: ExecutionContext): Future[String] =
    Future { ❶
        val source = Source.fromFile(filename)
        try {
            source.getLines().mkString
        } finally source.close() ❷
    }

    def count(text: String): Set[(Char, Int)] =
        text.groupBy(char => char).map { case (char, occurrences) =>
            char -> occurrences.length
        }.toSet
}

}
```

❶ Executing the function asynchronously because it reads from a file, which is a slow operation.

❷ Closing the file resource to avoid memory leaks

You now want to write a test for this new asynchronous function. First, you need to create a test resource for your test: create a new file containing the word "test" called myFile.txt in the directory src/test/resources. You can now use the specification class `AsyncFlatSpec` to define assertions on both synchronous and asynchronous values:

Listing 52.6: Adding an asynchronous test

```
package org.example.frequency

import org.scalatest.flatspec.AsyncFlatSpec ❶
import org.scalatest.matchers.should

class FrequencyTest extends AsyncFlatSpec with should.Matchers {

    val frequency = new Frequency

    // ...

    it should "asynchronously count characters from a file" in {
        val filename = getClass.getResource("/myFile.txt").getPath ❷
        val result = frequency.countFromFile(filename)
        val expected = Set('t' -> 2, 'e' -> 1, 's' -> 1)
        result.map(_.shouldEqual(expected)) ❸
    }
}
```

- 1 Allows you to assert on synchronous and asynchronous values
- 2 Retrieving the path of your test resource
- 3 Defining an assertion for your asynchronous value

The specification class `AsyncFlatSpec` offers the same testing style as `AnyFlatSpec` with dedicated support for values of type `Future`. The assertion is valid only if the `Future` completes successfully and its value respects the given assumption.

QUICK CHECK 52.2

The function `countFromFile` currently returns an asynchronous failure if the given file does not exist: modify its implementation to return an empty set instead. Write a test for this use case. HINT: The `recover` function of a `Future` instance allows you to define recovery values to use when throwing specific exceptions. It was the following signature:

```
def recover[T](pf: PartialFunction[Throwable, T])
  (using executor: ExecutionContext): Future[T]
```

Testing with IO: use unsafeRunSync

If your application uses `cats.effect.IO` to represent asynchronous computation, you do not need dedicated support when writing tests for it. Thanks to its clear separation between the definition and the execution of some computation, you can use the function `unsafeRunSync` for its evaluation, which allows you to evaluate your side effects synchronously, making your tests easier to write.

52.4 Summary

In this lesson, my objective was to introduce you to the basics of writing tests in Scala.

- You have seen how to write a test using the specification class `AnyFlatSpec` from the library `ScalaTest`, and how to run it in `sbt`.
- You have also discovered how to test asynchronous code thanks to its specification class `AsyncFlatSpec`.

Let's see if you got this!

TRY THIS:

You have written a test that reads from a file in listing 52.6. This approach can be inconvenient because reading from an actual file is a slow operation, and you may not want to create a file for each of your test cases. Write new test cases that assert the behavior of `countFromFile` without relying on actual files but rather simulating (or stubbing) their content. HINT: Modify the access modifier of `readFromFile` from `private` to `protected`, and define a new subclass of `Frequency` that overrides it to rely on a dictionary to link a filename to its content instead.

52.5 Answers to quick checks

QUICK CHECK 52.1:

First, you should replicate the bug by adding new test assertions and scenarios. For example, you could do something similar to the following:

```
class FrequencyTest extends AnyFlatSpec with should.Matchers {

  // ...

  it should "count the frequency of the characters in a text" in {
    val resultA = frequency.count("test")
    val expectedA = Set('t' -> 2, 'e' -> 1, 's' -> 1)
    resultA.shouldEqual(expectedA)

    val resultB = frequency.count("Test test")
    val expectedB = Set('t' -> 4, 'e' -> 2, 's' -> 2)
    resultB.shouldEqual(expectedB)
  }

  it should "consider characters as case insensitive" in {
    val result = frequency.count("AaA")
    result.shouldEqual(Set('a' -> 3))
  }

  it should "consider only letters and digits" in {
    val result = frequency.count(" ,.?!")
    result.shouldEqual(Set.empty)
  }
}
```

You should check that the new test assertions replicate the bugs:

```
$ sbt test
// ...
[info] FrequencyTest:
[info] Frequency
[info] - should count no frequency for the empty string
[info] - should count the frequency of the characters in a text *** FAILED ***
[info]   HashSet((T,1), ( ,1), (e,2), (t,3), (s,2)) did not equal Set((t,4), (e,2), (s,2))
  (FrequencyTest.scala:20)
[info]   Analysis:
[info]     HashSet(missingInLeft: [(t,4)], missingInRight: [(T,1), ( ,1), (t,3)])
[info] - should consider characters as case insensitive *** FAILED ***
[info]   Set((A,2), (a,1)) did not equal Set((a,3)) (FrequencyTest.scala:25)
[info]   Analysis:
[info]     Set(missingInLeft: [(a,3)], missingInRight: [(A,2), (a,1)])
[info] - should consider only letters and digits *** FAILED ***
[info]   HashSet((?,1), ( ,2), (!,1), (.,1), (,,1)) did not equal Set() (FrequencyTest.scala:30)
[info]   Analysis:
[info]     HashSet(missingInRight: [(?,1), ( ,2), (!,1), (.,1), (,,1)])
[info] Run completed in 433 milliseconds.
[info] Total number of tests run: 4
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 3, canceled 0, ignored 0, pending 0
[info] *** 3 TESTS FAILED ***
```

Finally, you change your implementation and verify that all tests are passing:

```
class Frequency {

    def count(text: String): Set[(Char, Int)] =
        text.toLowerCase.filter(_.isLetterOrDigit)
            .groupBy(char => char).map { case (char, occurrences) =>
                char -> occurrences.length
            }.toSet

}
```

```
$ sbt test
// ...
[info] FrequencyTest:
[info] Frequency
[info] - should count no frequency for the empty string
[info] - should count the frequency of the characters in a text
[info] - should consider characters as case insensitive
[info] - should consider only letters and digits
[info] Run completed in 399 milliseconds.
[info] Total number of tests run: 4
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 4, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 5 s, completed 5 Feb 2021, 12:34:43
```

QUICK CHECK 52.2:

First, you add a new test case for your function `countFromFile` and verify that it fails:

```
it should "return the empty set if the file does not exist" in {
    val result = frequency.countFromFile("I-do-not-exist.txt")
    result.map(_.shouldEqual(Set.empty))
}
```

You can now change your implementation of `countFromFile` and make the tests pass:

```
def countFromFile(filename: String)
    (using ec: ExecutionContext): Future[Set[(Char, Int)]] =
    readFromFile(filename).map(count).recover {
        case _: FileNotFoundException => Set.empty
    }
```

53

The Quiz Application: Part 2

In this capstone, you will:

- Lazily evaluate side effects using the type `IO`.
- Define an HTTP API to send and receive requests using `http4s`.
- Serialize and deserialize HTTP responses in JSON format using `circe`.
- Write tests to assert that your business logic is correct.

In this capstone, you'll complete the quiz application you have started in unit 7 by defining its service and API layers. In particular, you'll implement a server that offers its users an HTTP API to perform the following operations:

- Inform about the status of your application.
- Display a list of categories a user can choose.
- Generate a quiz by randomly pick ten questions for a given category.
- Calculate the quiz score based on the received answers.

In your application, you'll focus on processing existing data rather than creating a new one: your database will contain preloaded data. However, you can easily add endpoints to allow users to add new categories, questions, and answers following the same patterns if you wish to do so.

You will structure your application in layers to have a clear separation of concerns (see figure 53.1):

- The DAO Layer defines how to communicate with the database. It exposes records, which are a close representation of the structure of the database's tables.
- The Service Layer defines the business logic of your application. A service class accesses a specific DAO instance, and it returns entities, not records. Your application can expose entities through its API, so they should have JSON encoders and decoders defined for them.
- The API layer defines the structure of each endpoint. An API class does not contain any business logic since this is the responsibility of the service layer.

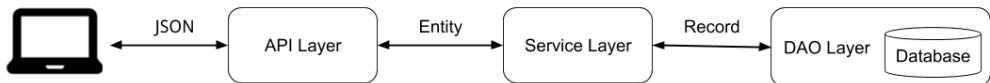


Figure 53.1: Each layer of your application has specific concerns. The DAO layer defines how to communicate with the database, and it produces records, which are a close representation of the database's tables. The service layer creates entities that the API layer can expose in JSON format. The API layer defines each endpoint's structure.

This lesson uses Scala 2

The code examples in this lesson use Scala 2 because the libraries http4s and Quill don't support Scala 3 yet. The lesson will be updated to Scala 3 before publication as soon as http4s and Quill versions for Scala 3 are available.

53.1 Download the base project

The first task you need to perform is downloading the base project you are going to use. Navigate to an empty directory and execute the following `git` commands to check out the remote branch:

```
$ git init
$ git remote add daniela https://github.com/DanielaSfregola/get-programming-with-scala.git
$ git fetch daniela
$ git checkout -b my_lesson53 daniela/baseline_unit8_lesson53
```

The code contains the code you have implemented in the previous unit's capstone with a few minor additions. First, your database will now have preloaded categories, questions, and answers: have a look at its `src/main/resource/init.sql` and `TestDatabase` object to learn how to do this. Its data has been extracted from Open Trivia DB (see opentdb.com), a free and user-contributed collection of trivia questions. Finally, its `build.sbt` file has a few new dependencies: it now lists `cats-effect`, `circe`, `http4s` and `scalatest` among its external libraries.

Listing 53.1: The build.sbt file

```
// file build.sbt

// ...

val CirceVersion  = "0.14.0-M4"
val Http4sVersion = "0.21.8"

libraryDependencies ++= List(
  // ...
  "org.typelevel" %% "cats-effect" % "2.3.3",
  "io.circe" %% "circe-core" % CirceVersion,
  "io.circe" %% "circe-generic" % CirceVersion,
  "io.circe" %% "circe-parser" % CirceVersion,
  "org.http4s" %% "http4s-blaze-server" % Http4sVersion,
  "org.http4s" %% "http4s-dsl" % Http4sVersion,
  "org.http4s" %% "http4s-circe" % Http4sVersion,
  "org.scalatest" %% "scalatest" % "3.2.5" % "test"
)
```

You can now run the `sbt compile` from your project's root directory to download all its external dependencies and compile its code.

53.2 Generic Endpoints

The first set of endpoints you are going to implement is not specific to your application domain. Offering a few endpoints to ensure your system is responsive and healthy is good practice. The `healthCheck` endpoint is useful for monitoring systems, while `ping` allows its users to ensure they can successfully communicate with it by performing a concise and lightweight request.

Listing 53.2 shows you how to create a `GenericService` class inside the package `org.example.quiz.service` containing the logic to produce an informative message about the connectivity to its internal dependencies.

Listing 53.2: The `GenericService` class

```
// file src/main/scala/org/example/quiz/service/GenericService.scala
package org.example.quiz.service

import cats.effect.{ContextShift, IO}
import org.example.quiz.dao.GenericDao

class GenericService(dao: GenericDao)(implicit cs: ContextShift[IO]) { ①

    def healthCheck: IO[String] =
        checkDbConnectivity().map { success =>
            s"Database Connectivity: ${if (success) "OK" else "FAILURE"}"
        }

    private def checkDbConnectivity(): IO[Boolean] =
        IO.fromFuture(IO(dao.testConnection())) ②
            .handleErrorWith(_ => IO(false)) ③

}
```

① The function `IO.fromFuture` needs an implicit `ContextShift` parameter.

② Converting a `Future` instance into `IO`.

③ During its evaluation, if it throws an exception, it returns `IO(false)` instead.

The function `IO.fromFuture` allows you to convert an instance of `Future` into one of type `IO`. It has the following signature:

```
def fromFuture[A](iof: IO[Future[A]])(implicit cs: ContextShift[IO]): IO[A]
```

Rather than refactoring the DAO layer to use `IO` rather than `Future`, you can invoke the `IO.fromFuture` function to perform the conversion in the service layer. You can now define the generic routes to reply to `ping` messages and inform about the status of your application's internal dependencies.

Listing 53.3: The GenericApi class

```
// file src/main/scala/org/example/quiz/api/GenericApi.scala
package org.example.quiz.api

import cats.effect.IO
import org.example.quiz.service.GenericService
import org.http4s.HttpRoutes
import org.http4s.dsl.Http4sDsl

class GenericApi(genericService: GenericService) extends Http4sDsl[IO] {

    val routes = HttpRoutes.of[IO] {
        case GET -> Root / "ping" => Ok("pong")
        case GET -> Root / "healthCheck" => Ok(genericService.healthCheck)
    }
}

}
```

You'll integrate these endpoints into your server and expose them as part of your HTTP API.

53.3 Displaying the Available Categories

The next endpoint allows users to see all the available quiz categories. First of all, you need to decide what information you are willing to expose in your HTTP API by implementing the `CategoryEntity` class:

Listing 53.4: The CategoryEntity class

```
// file src/main/scala/org/example/quiz/entities/CategoryEntity.scala
package org.example.quiz.entities

import org.example.quiz.dao.records.Category
import io.circe.generic.semiauto.-
import io.circe.-

case class CategoryEntity(id: Long, name: String)

object CategoryEntity {

    implicit val encoder: Encoder[CategoryEntity] =
        deriveEncoder[CategoryEntity] ①
    implicit val decoder: Decoder[CategoryEntity] =
        deriveDecoder[CategoryEntity] ①

    def fromRecord(record: Category): CategoryEntity = ②
        apply(id = record.id, name = record.name)
}
```

① Defining JSON encoder and decoder for `CategoryEntity`

② Converting a record (i.e., data read from the database) into an entity (i.e., data that the API exposes)

Its service needs to read all the categories from the database and convert them into entities so that the API can expose them in JSON format. It also needs to retrieve a category by its id: you'll use this when processing a quiz generation request. Listing 535 shows you how to implement the `CategoryService` class:

Listing 53.5: The CategoryService class

```
// file src/main/scala/org/example/quiz/service/CategoryService.scala
package org.example.quiz.service

import cats.effect.{ContextShift, IO}
import org.example.quiz.dao.CategoryDao
import org.example.quiz.dao.records.Category
import org.example.quiz.entities.CategoryEntity

class CategoryService(dao: CategoryDao)(implicit cs: ContextShift[IO]) {❶

    def get(id: Long): IO[Option[CategoryEntity]] =
        IO.fromFuture(IO(dao.findById(id))).map { maybeRecord => ❷
            maybeRecord.map(CategoryEntity.fromRecord)
        }

    def all(): IO[List[CategoryEntity]] =
        IO.fromFuture(IO(dao.all())).map { records => ❷
            records.map(CategoryEntity.fromRecord)
        }
}
```

❶ The function `IO.fromFuture` needs an implicit `ContextShift` parameter.

❷ Converting a Future instance into IO.

You can now define the route to display the categories: listing 53.6 shows you how to achieve this.

Listing 53.6: The CategoryApi class

```
package org.example.quiz.api

import cats.effect.IO
import org.example.quiz.entities.CategoryEntity
import org.example.quiz.service.CategoryService
import org.http4s.HttpRoutes
import org.http4s.circe._ ❶
import org.http4s.dsl.Http4sDsl

class CategoryApi(categoryService: CategoryService) extends Http4sDsl[IO] {

    private implicit val categoriesEncoder =
        jsonEncoderOf[IO, List[CategoryEntity]] ❷

    val routes = HttpRoutes.of[IO] {
        case GET -> Root => Ok(categoryService.all())
    }
}
```

❶ Importing http4s functionalities to generate a JSON entity encoder.

❷ It defines a JSON encoder for an HTTP entity containing a list of categories. It implicitly requires an instance of `Encoder[CategoryEntity]`.

When defining your server, you'll map the category routes to the prefix "categories" so that a user can receive all the categories by performing a GET request to the endpoint `/categories`.

53.4 Creating a Quiz

Let's implement the endpoint that will generate a quiz containing ten random questions for a given category. Implement a class `QuizEntity` in the package `org.example.quiz.entities` to define how to display the quiz questions. You want to make sure not to reveal which ones are the correct answers at this stage.

Listing 53.7: The QuizEntity class

```
// file src/main/scala/org/example/quiz/entities/QuizEntity.scala
package org.example.quiz.entities

import io.circe._
import io.circe.generic.semiauto._
import org.example.quiz.dao.records.{Answer, Question}

case class QuizEntity(questions: List[QuestionEntity])

object QuizEntity {
    implicit val encoder: Encoder[QuizEntity] = ❶
        deriveEncoder[QuizEntity]
    implicit val decoder: Decoder[QuizEntity] = ❶
        deriveDecoder[QuizEntity]
}

case class QuestionEntity(id: Long,
                         text: String,
                         possibleAnswers: Set[PossibleAnswerEntity])

object QuestionEntity {
    implicit val encoder: Encoder[QuestionEntity] = ❶
        deriveEncoder[QuestionEntity]
    implicit val decoder: Decoder[QuestionEntity] = ❶
        deriveDecoder[QuestionEntity]

    def fromRecord(question: Question,
                  answers: List[Answer]): QuestionEntity = { ❷
        val possibleAnswers = answers.map(PossibleAnswerEntity.fromRecord).toSet
        apply(question.id, question.text, possibleAnswers)
    }
}

case class PossibleAnswerEntity(id: Long, text: String)

object PossibleAnswerEntity {
    implicit val encoder: Encoder[PossibleAnswerEntity] = ❶
        deriveEncoder[PossibleAnswerEntity]
    implicit val decoder: Decoder[PossibleAnswerEntity] = ❶
        deriveDecoder[PossibleAnswerEntity]

    def fromRecord(answer: Answer): PossibleAnswerEntity = { ❷
        apply(answer.id, answer.text)
    }
}
```

- ① Defining JSON encoder and decoder for the entity
- ② Converting a record (i.e., data read from the database) into an entity (i.e., data that the API exposes)

You can now implement the `QuizService` class that defines how to generate a quiz from a given category id. Listing 53.8 shows you how to achieve this:

Listing 53.8: The QuizService: generating a quiz

```
// file src/main/scala/org/example/quiz/service/QuizService.scala
package org.example.quiz.service

import cats.effect.{ContextShift, IO}
import org.example.quiz.dao.QuestionAnswerDao
import org.example.quiz.entities._

class QuizService(dao: QuestionAnswerDao,
                 categoryService: CategoryService)
  (implicit cs: ContextShift[IO]) { ①

  private val numberOfQuestions = 10

  def generate(categoryId: Long): IO[Option[QuizEntity]] =
    categoryService.get(categoryId).flatMap {
      case Some(category) =>
        pickQuestions(category, numberOfQuestions).map(qs =>
          Some(QuizEntity(qs)))
      case None => IO(None)
    }

  private def pickQuestions(category: CategoryEntity,
                           n: Int): IO[List[QuestionEntity]] = {
    val randomQAs = IO.fromFuture(IO(
      dao.pickByCategoryId(category.id, n = n))) ②
    randomQAs.map { qas =>
      qas.map { case (q, as) => QuestionEntity.fromRecord(q, as) }.toList
    }
  }
}
```

- ① The function `IO.fromFuture` needs an implicit `ContextShift` parameter.
- ② Converting a Future instance into IO.

Finally, let's define an endpoint that responds to a GET request passing a query parameter `category_id` with either a `QuizEntity` instance or a 404 Not Found error code. Listing 53.9 shows you how to do this:

Listing 53.9: The QuizApi: generating a quiz

```
// file src/main/scala/org/example/quiz/api/QuizApi.scala
package org.example.quiz.api

import cats.effect.IO
import org.example.quiz.entities._
import org.example.quiz.service.QuizService
import org.http4s.HttpRoutes
import org.http4s.dsl._
import org.http4s.circe._ ①
```

```

class QuizApi(quizService: QuizService) extends Http4sDsl[IO] {

    private implicit val quizEncoder = jsonEncoderOf[IO, QuizEntity] ②

    private object CategoryParam extends
QueryParamDecoderMatcher[Long]("category_id") ③

    val routes = HttpRoutes.of[IO] {
        case GET -> Root :? CategoryParam(categoryId) =>
            quizService.generate(categoryId).flatMap {
                case Some(quiz) => Ok(quiz)
                case None => NotFound(s"Category $categoryId does not exist")
            }
    }
}

```

① Importing http4s functionalities to generate a JSON entity encoder.

② It defines a JSON encoder for an HTTP entity containing a quiz entity. It implicitly requires an instance of Encoder[QuizEntity].

③ It matches a query parameter with name category_id and a value of type Long.

In the server, you'll map the quiz routes to a "quiz" prefix. For example, a user can request a quiz for the category with id 10 by sending a GET request to the /quiz?category_id=10 endpoint.

53.5 Answering a Quiz

The last endpoint that you'll implement allows users to send their quiz answers and receive their quiz score in return containing the percentage of given correct answers, together with a summary of which ones are correct and which ones are not.

The class GivenAnswerEntity defines the format users must follow for each of their answers, while the class ScoreEntity specifies what information to display when replying with their score.

Listing 53.10: The GivenAnswerEntity and ScoreEntity classes

```

// file src/main/scala/org/example/quiz/entities/GivenAnswerEntity.scala
package org.example.quiz.entities

import io.circe._
import io.circe.generic.semiauto._

case class GivenAnswerEntity(questionId: Long, answerId: Long)

object GivenAnswerEntity {

    implicit val encoder: Encoder[GivenAnswerEntity] = ①
        deriveEncoder[GivenAnswerEntity]
    implicit val decoder: Decoder[GivenAnswerEntity] = ①
        deriveDecoder[GivenAnswerEntity]
}

// file src/main/scala/org/example/quiz/entities/ScoreEntity.scala
package org.example.quiz.entities

import io.circe._
import io.circe.generic.semiauto._


```

```

case class ScoreEntity(score: Double,
                      correct: List[GivenAnswerEntity],
                      wrong: List[GivenAnswerEntity])

object ScoreEntity {

    implicit val encoder: Encoder[ScoreEntity] = ①
        deriveEncoder[ScoreEntity]
    implicit val decoder: Decoder[ScoreEntity] = ②
        deriveDecoder[ScoreEntity]
}

```

① Defining JSON encoder and decoder for the entity

You can now add the logic to create a `ScoreEntity` instance from an instance of type `List[GivenAnswerEntity]` in the `QuizService` class:

Listing 53.11: The QuizService class: computing the score

```

// file src/main/scala/org/example/quiz/services/QuizService.scala
package org.example.quiz.service

import cats.effect.{ContextShift, IO}
import org.example.quiz.dao.QuestionAnswerDao
import org.example.quiz.entities._

class QuizService(dao: QuestionAnswerDao,
                  categoryService: CategoryService)
  (implicit cs: ContextShift[IO]) {

    // ...

    def score(givenAnswers: List[GivenAnswerEntity]): IO[ScoreEntity] = {
        val questionIds = givenAnswers.map(_.questionId)
        IO.fromFuture(IO(dao.getCorrectQuestionAnswers(questionIds))) ①
        .map { correctAnswers =>
            val goodAnswers = givenAnswers.filter { answer =>
                correctAnswers.exists { case (q, a) =>
                    q == answer.questionId && a == answer.answerId
                }
            }
            val badAnswers = givenAnswers.diff(goodAnswers)
            val score = 1.0 * goodAnswers.size / givenAnswers.size
            ScoreEntity(score, correct = goodAnswers, wrong = badAnswers)
        }
    }
}

```

① Converting a Future instance into IO.

Finally, let's define an endpoint for it in the `QuizApi` class for the users to post their responses:

Listing 53.12: The QuizApi class: computing the score

```
// file src/main/scala/org/example/quiz/api/QuizApi.scala
package org.example.quiz.api

import cats.effect.IO
import org.example.quiz.entities._
import org.example.quiz.service.QuizService
import org.http4s.HttpRoutes
import org.http4s.dsl._
import org.http4s.circe._ ①

class QuizApi(quizService: QuizService) extends Http4sDsl[IO] {

    // ...

    private implicit val givenAnswersDecoder =
        jsonOf[IO, List[GivenAnswerEntity]] ②
    private implicit val quizScoreEncoder =
        jsonEncoderOf[IO, ScoreEntity] ③

    val routes = HttpRoutes.of[IO] {
        //...
        case request @ POST -> Root =>
            val response = for {
                answers <- request.as[List[GivenAnswerEntity]]
                score <- quizService.score(answers)
            } yield score
            Ok(response)
    }
}
```

- ① Importing http4s functionalities to generate a JSON entity encoder.
- ② It creates a JSON decoder for an HTTP entity containing a list of given answers. It implicitly requires an instance of Decoder[GivenAnswerEntity].
- ③ It defines a JSON encoder for an HTTP entity containing a score entity. It implicitly requires an instance of Encoder[ScoreEntity].

You will map the quiz routes to the prefix “quiz”: users can send a POST request to the endpoint /quiz with their responses in JSON format as its body and receive their score in response.

53.6 The HTTP Server

The core implementation is now over: let’s see how to connect all the components and define an HTTP server for your quiz application.

First, let’s define classes representing each layer of your application:

Listing 53.13: Defining each layer

```
// file src/main/scala/org/example/quiz/dao/Daos.scala
package org.example.quiz.dao

import io.getquill.{PostgresAsyncContext, SnakeCase}
import scala.concurrent.ExecutionContext
```

```

class Dao(ctx: PostgresAsyncContext[SnakeCase.type]) ①
  (implicit ec: ExecutionContext) {

  val category = new CategoryDao(ctx)
  val generic = new GenericDao(ctx)
  val questionAnswer = new QuestionAnswerDao(ctx)

}

// file src/main/scala/org/example/quiz/services/Services.scala
package org.example.quiz.service

import cats.effect.{ContextShift, IO}
import org.example.quiz.dao.Dao

class Services(dao: Dao)(implicit cs: ContextShift[IO]) { ②

  val generic = new GenericService(dao.generic)
  val category = new CategoryService(dao.category)
  val quiz = new QuizService(dao.questionAnswer, category)

}

// file src/main/scala/org/example/quiz/api/Api.scala
package org.example.quiz.api

import org.example.quiz.service.Services

class Api(services: Services) { ③

  val category = new CategoryApi(services.category)
  val generic = new GenericApi(services.generic)
  val quiz = new QuizApi(services.quiz)

}

```

① The DAO layer can only access the database

② The service can only access the DAO layer

③ The API layer can only access the service layer

The final step of your implementation is to define an HTTP server: you need to bind it to a port and host and mount the routes that you want to expose in your HTTP API. Listing 53.14 shows you how to do this:

Listing 53.14: The QuizApp executable object

```

// file src/main/scala/org/example/quiz/QuizApp.scala
package org.example.quiz

import cats.effect.{ExitCode, IO, IOApp}
import org.example.quiz.api.Api
import org.example.quiz.dao.Dao
import org.example.quiz.service.Services
import org.http4s.server.Router

```

```

import org.http4s.implicits._
import org.http4s.server.blaze.BlazeServerBuilder
import scala.concurrent.ExecutionContext

object QuizApp extends IOApp { ①

    private val dao = new Dao(TestDatabase.ctx)(ExecutionContext.global)
    private val services = new Services(dao)
    private val api = new Api(services)

    private val httpApp = Router(
        "/" -> api.generic.routes,
        "categories" -> api.category.routes,
        "quiz" -> api.quiz.routes
    ).orNotFound ②

    override def run(args: List[String]): IO[ExitCode] =
        stream(args).compile.drain.as(ExitCode.Success)

    private def stream(args: List[String]) =
        BlazeServerBuilder[IO](ExecutionContext.global)
            .bindHttp(8000, "0.0.0.0")
            .withHttpApp(httpApp)
            .serve
}

```

- ① An executable app using IO that provides an implicit context shift instance.
 ② Return a 404 error if the server does not find a matching endpoint for the received request.

The implementation of your quiz application is now complete.

53.7 Writing tests

This section will show you a possible strategy to use when writing tests for your application. You could follow one or many testing strategies. You could write tests that start your application using a test database, send HTTP requests to it and assert its HTTP responses. You could also write tests specific to its business logic and independent from the other components of your application, such as its API and DAO layers.

Let's write a test for the class `CategoryService`: the tests for the other services follow the same structure. A `CategoryService` has one parameter of type `CategoryDao`: you need to define a class that simulates its behavior by reading some pre-defined test data, called fixtures, rather than querying a database.

Listing 53.15: Simulating CategoryDao

```

// file src/test/scala/org/example/quiz/stubs/dao/Fixtures.scala

package org.example.quiz.stubs.dao

import org.example.quiz.dao.records._

object Fixtures {

    val catA = Category(id = 1, name = "General")
    val catB = Category(id = 2, name = "History")
}

```

```

    val categories: List[Category] = List(catA, catB)
}

// file src/test/scala/org/example/quiz/stubs/dao/FakeCategoryDao.scala

package org.example.quiz.stubs.dao

import org.example.quiz.dao.CategoryDao
import org.example.quiz.dao.records.Category

import scala.concurrent.{ExecutionContext, Future}

class FakeCategoryDao(implicit ec: ExecutionContext) extends CategoryDao(ctx = null) { ①

    private var fakeCategories = Fixtures.categories

    private def safelyModify(f: List[Category] => List[Category]): Unit = ②
        synchronized { fakeCategories = f(fakeCategories) }

    override def save(category: Category): Future[Long] = { ③
        safelyModify(_ :+ category)
        Future(category.id)
    }

    override def all(): Future[List[Category]] = Future(fakeCategories)

    override def findById(id: Long): Future[Option[Category]] = ③
        Future(fakeCategories.find(_.id == id))

    override def deleteById(id: Long): Future[Boolean] = { ③
        val isPresent = fakeCategories.exists(_.id == id)
        safelyModify(_.filterNot(_.id == id))
        Future(isPresent)
    }
}

```

① It sets the database context to null because unused

② Ensuring only one thread at the time can modify the list of categories

③ Overriding the behavior of the function to rely on the list of categories rather than querying a database

You can now use your `FakeCategoryDao` implementation to test the business logic defined in `CategoryService`:

Listing 53.16: Writing a test for CategoryService

```

// file src/test/scala/org/example/quiz/service/CategoryServiceTest.scala
package org.example.quiz.service

import cats.effect.{ContextShift, IO}
import org.example.quiz.entities.CategoryEntity
import org.example.quiz.stubs.dao.{FakeCategoryDao, Fixtures}
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers

```

```
import scala.concurrent.ExecutionContext

class CategoryServiceTest extends AnyFlatSpec with Matchers {

    private def mkService() = {
        implicit val ec: ExecutionContext = ExecutionContext.global
        implicit val cs: ContextShift[IO] = IO.contextShift(ec)

        new CategoryService(new FakeCategoryDao)
    }

    "CategoryService" should "return all the categories" in {
        val service = mkService()
        val entities = service.all().unsafeRunSync()
        val expectedEntities =
            Fixtures.categories.map(CategoryEntity.fromRecord)
        entities.shouldEqual(expectedEntities)
    }

    it should "return a category if the id exists" in {
        val service = mkService()
        val entity = CategoryEntity.fromRecord(Fixtures.catA)
        val optRes = service.get(entity.id).unsafeRunSync()
        optRes.shouldEqual(Some(entity))
    }

    it should "return no category if the id is invalid" in {
        val service = mkService()
        val optRes = service.get(-1).unsafeRunSync()
        optRes.shouldEqual(None)
    }
}
```

You can use this pattern to write tests for all the remaining services. Once your tests pass using the command `sbt test`, you are ready to run your application.

53.8 Let's give it a try!

It's time to see your quiz application in action. Navigate the root directory of your project and execute the command `sbt run` to start your server. Once ready, you will see the following message in your console:

First, let's ensure that you can connect to the service by performing a GET request to <http://localhost:8000/ping>:

```
$ curl -i http://localhost:8000/ping
HTTP/1.1 200 OK
Content-Type: text/plain; charset=UTF-8
Date: Sun, 08 Nov 2020 11:34:56 GMT
Content-Length: 4

pong
```

You can also check that your application is healthy and that it can talk to the database by sending a GET request to <http://localhost:8000/healthCheck>:

```
$ curl -i http://localhost:8000/healthCheck
HTTP/1.1 200 OK
Content-Type: text/plain; charset=UTF-8
Date: Sun, 08 Nov 2020 11:36:12 GMT
Content-Length: 25

Database Connectivity: OK
```

Sending a request to an endpoint that does not exist should return a 404 – Not found error:

```
$ curl -i http://localhost:8000/invalid-endpoint
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=UTF-8
Date: Sun, 08 Nov 2020 11:40:32 GMT
Content-Length: 9

Not found
```

You can see the list of available categories at <http://localhost:8000/categories>:

```
$ curl -i http://localhost:8000/categories
HTTP/1.1 200 OK
Content-Type: application/json
Date: Sun, 08 Nov 2020 11:43:30 GMT
Content-Length: 898

[{"id":9,"name":"General Knowledge"}, {"id":10,"name":"Entertainment: Books"}, {"id":11,"name":"Entertainment: Film"}, {"id":12,"name":"Entertainment: Music"}, {"id":13,"name":"Entertainment: Musicals & Theatres"},  
// ...truncating output...
{"id":32,"name":"Entertainment: Cartoon & Animations"}]
```

First, let's request a quiz for a category that does not exist and ensure that it fails with a meaningful message:

```
$ curl -i http://localhost:8000/quiz?category_id=42
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=UTF-8
Date: Sun, 08 Nov 2020 11:47:46 GMT
Content-Length: 26

Category 42 does not exist
```

Let's try again, this time by requesting a category, for example "General Knowledge" with id 9:

```
$ curl -i http://localhost:8000/quiz?category_id=9
HTTP/1.1 200 OK
```

```
Content-Type: application/json
Date: Sun, 08 Nov 2020 11:50:37 GMT
Content-Length: 2299

[{"questions": [{"id": 17, "text": "What is the right way to spell the capital of Hungary?", "possibleAnswers": [{"id": 62, "text": "Bhudapest"}, {"id": 63, "text": "Budapast"}, {"id": 64, "text": "Budapest"}, {"id": 65, "text": "Boodapest"}]}, {"id": 9, "text": "Complete the following analogy: Audi is to Volkswagen as Infiniti is to ?", "possibleAnswers": [{"id": 30, "text": "Subaru"}, {"id": 31, "text": "Nissan"}, {"id": 32, "text": "Honda"}, {"id": 33, "text": "Hyundai"}]}]}
```

The application will calculate your score when sending your answers as a POST request to `http://localhost:8000/quiz`:

```
$ curl -i -X POST -d '[{"questionId": 22, "answerId": 80}, {"questionId": 17, "answerId": 62}]' http://localhost:8000/quiz
// sending only two answers rather than ten for breifty

HTTP/1.1 200 OK
Content-Type: application/json
Date: Sun, 08 Nov 2020 12:00:25 GMT
Content-Length: 99

{"score": 0.5, "correct": [{"questionId": 22, "answerId": 80}], "wrong": [{"questionId": 17, "answerId": 62}]}
```

You sent two answers: only one of them was correct, so you receive a score of 50%.

53.9 Summary

In this capstone, you have implemented an HTTP server that exposes an HTTP API to generate and answer quizzes.

- You have implemented endpoints to monitor the status of your application.
- You have defined its business logic in its service layer and wrote tests for it.
- You have designed its endpoints in its API layer.
- You have implemented a server that exposes an HTTP API, and you have consumed its data to monitor its status and play a quiz.