Intro

This book will give you a great, hands on introduction to computer programming. It's written for complete beginners, and will lead you on a journey from installing everything you need to start coding, to creating a Top Trumps style card game. Along the way you'll learn the fundamental building blocks and techniques for writing clean, easy to read code.

Scala is a great first language to learn, as it straddles two of the main paradigms in programming: Object Oriented and Functional Programming. You'll learn about both of these approaches, and see that in coding there are often many ways to achieve the same results. We'll also take a test driven approach to writing code right from the beginning. Test Driven Development is a practice widely used by professional coders to help improve the quality of their code, but is rarely covered in beginners' books.

There's a wealth of resources out there teaching you how to make games, mobile apps, websites, machine learning apps, Internet of Things controllers, the list goes on. Most of these assume you've got some level of coding knowledge. This book will give you that fundamental knowledge, and unlock all those resources for you.

Learning to code can be challenging. It will almost certainly be frustrating at times. But it's a great goal to aim for. In what other field is it possible to create so much with so little? All you need is a computer, some coding skills, and your imagination. If you've read this far I urge you to give it a go. It's easier than you think!

Who this book is for

This is a book for beginners. I'm going to assume that you know absolutely nothing about coding. However, you will need to have some level of confidence using computers, mainly to get everything set up. We'll be installing programs from the internet, using the command line, and possibly configuring environment variables on your computer. Because every computer is different, and things change rapidly in the world of software development, you may find that the steps I describe to get set up don't quite work for you, so you may need to search for solutions. I'll try to point you in the right direction wherever I can.

This is not a Scala book. It is a book about programming, that happens to use Scala. Scala is a very powerful and feature-rich programming language, and trying to cover all of the features would distract from the basics that I'm trying to focus on. There are plenty of good books and resources to comprehensively learn Scala, but they tend to assume that the reader has some pre-existing knowledge of programming. Once you finish this book you'll have that knowledge, and I'll point you to some of these Scala resources at the end of the book.

It will give you enough broad knowledge of the fundamentals for you to read and write basic programs. We'll dip into some important concepts in coding, such as Object Oriented Programming, Functional Programming, and Test Driven Development. These are huge subject areas in themselves, so for the sake of keeping this book focussed we'll be leaving out quite a lot. However, if this book whets your appetite for more, then you'll be in a position to pick up more advanced programming books and understand what they're talking about.

This book is not going to teach you how to write a website, or a mobile app, or a 3D game. Most real world programming takes place within an ecosystem of technologies and concepts. Website development requires knowledge of HTML, CSS, JavaScript frameworks, HTTP; mobile apps need understanding of IOS and Android operating systems; and to write a 3D game you'd probably use a game engine such as Unity, along with sprites, animations and sound effects. The beating heart of each of these ecosystems is the same though - coding.

I see this very much as a *gateway to coding* book. The fundamental concepts that you'll learn throughout this book are found in many other programming languages. Once you've finished you'll be all set to continue your coding journey. Perhaps you'll use the Play framework to create dynamic websites using Scala. Maybe you'll look into Swift and create an app for the Apple App Store. How about some C# for a retro 2D platform game? Or Python for data analytics? All you need is a computer, and the possibilities are limited just by your imagination.

How to read this book

This book is designed to be read in order. It starts by getting your computer set up for coding, then leads you through from very basic concepts to putting together a complex program at the end. Later chapters build on earlier ones, so if you jump about through the book you may find things that you don't understand.

There are plenty of code examples throughout the book, and each chapter will tend to build up a large example from several smaller ones. If you just have a general interest in learning what coding is about, feel free to read through the book without trying any of the examples. However, if you want to be able to write code, then I strongly advise you to follow along with the examples, and actually write and run the code. The best way of learning is to experiment, so write the code as it appears in the book, then try making changes to it and see what happens. If you can, think up your own programs using the concepts you've learnt so far and try writing them from scratch.

Let’s get set up

Unfortunately we can’t just open up a text editor and start coding. This is possible with some languages, such as JavaScript, which you could just write in Microsoft Word and run in your internet browser. However, Scala is a compiled language, which means that we’ll need a program called a *compiler* to convert the text we write to instructions the computer can understand. And that’s not the end of the story. Some compiled languages have a compiler for each type of computer you want your program to run on. So if you were to compile a C++ program using a Windows compiler, the program would only run on a Windows computer. If you wanted to have a version that ran on a Mac as well, you’d need to compile another version with a Mac compiler. Scala is part of the Java family of languages, which take the approach of *compile once, run anywhere*.

How this is achieved is by running another program, called a *Virtual Machine* on every computer that you want to run your program on. The Scala compiler turns your text into language that the Virtual Machine can understand, then the Virtual Machine translates this into instructions for the computer.

On top of this, we’re going to want another program called a *build tool*, that will help to organise and build our programs in a manageable way. And finally, although it’s possible to write code in a normal text editor, there are special coding editors that provide a lot of bells and whistles, and will really make your life easier.

Wow, so that’s a lot of stuff just to get started! At least you won’t have to pay for any of it. All of these programs are free to use.

Installing the JDK

To write Scala code you’ll need at least version 1.8 of the *Java Development Kit*. You can see whether you’ve already got it installed by opening up a Command Prompt (for Windows) or Terminal (for Mac/Linux), and typing `*javac -version*`. If the version shown starts with 1.8 or higher then you’re good to go. Otherwise you’ll need to install it.

As of the time of writing, you can download version 1.8 of the JDK from <https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html> . If that link doesn’t work, then just search for “JDK download”. Although there are more recent versions, I’d stick with version 1.8 to make sure you’re on the same version as me.

Accept the licence agreement and download the relevant file for your computer. If you’re on Windows you’ll see two choices: Windows x86 and Windows x64. You’ll need the x86 version if you have a 32 bit version of Windows, and the x64 version if you’re running a 64 bit version of Windows. If you’re not sure which you’ve got, open the Control Panel and go to the System view. The System Type will tell you whether you’re 32 or 64 bit.

Once the file has finished downloading, run it and the JDK will be installed. You can check it’s worked by opening a new Command Prompt/ Terminal and running `*javac -version*` again.

Installing IntelliJ

IntelliJ IDEA is an *Integrated Development Environment*, or IDE. You can think of an IDE as the coding equivalent of Photoshop for editing photos, or a CAD program for architecture. Although you can program without one, it makes your life a lot easier. There are several different IDEs for programming in Scala, including Eclipse and Netbeans, but I’d suggest you install IntelliJ to make it easy to follow along.

A nice thing with IntelliJ is that we can install the Scala compiler and build tool as part of it. At the moment, you can download IntelliJ from <https://www.jetbrains.com/idea/download>. Of course, if that link doesn’t work then just search for “Download IntelliJ”. Choose the Community Edition, which is free. Once the file is downloaded, click it to run the installer.

During the installation process just accept all the default options, other than the Plugins. You should get an option to choose Plugins, in which case select to install the Scala Plugin. If you don’t see this option, or miss it by accident, then don’t worry. Once IntelliJ is installed and you create your first project, you can install Plugins by going to **Settings → Preferences → Plugins**.

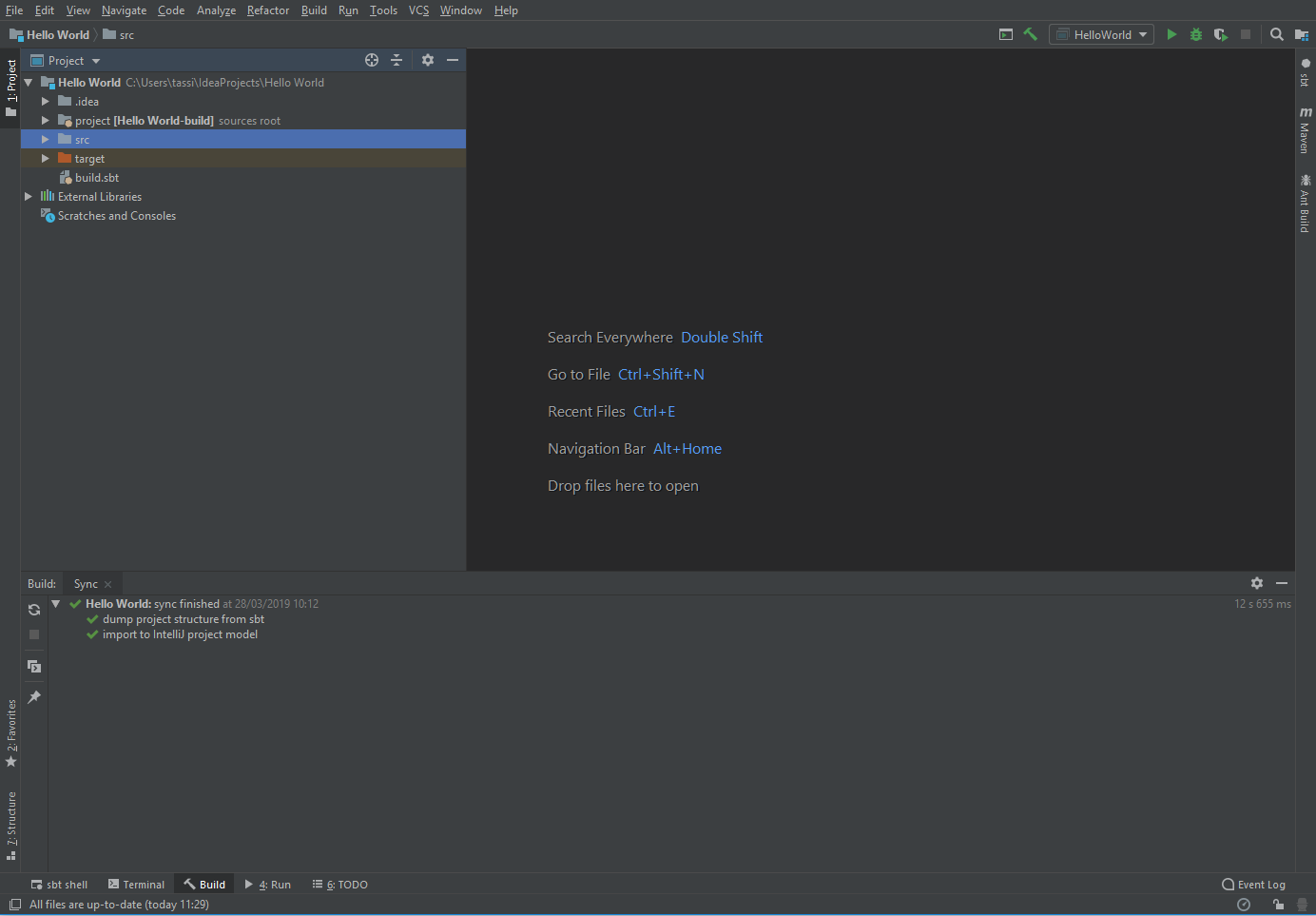
If you get stuck, there are more detailed instructions on the IntelliJ website: <https://docs.scala-lang.org/getting-started-intellij-track/getting-started-with-scala-in-intellij.html>.

Your first program

If all's gone well you should have IntelliJ installed with the Scala Plugin and have a JDK. If so, well done! That's all the boring stuff out of the way, and now we're ready to start coding.

We're going to get going with a really simple program. It won't do very much, other than print out some text, but it's quite an achievement to actually get something running. I'm going to follow the age old programmer's initiation rite of writing my first program to print out "Hello World!", but feel free to change it to print out whatever you like. In fact, one of the best ways of learning as you follow through these tutorials will be to understand what the code is doing, then play around and try to get it to do something slightly different.

First things first, open up IntelliJ and create a new project. Choose a Scala sbt-based project and give your project a name. I’m going to call mine “Hello World”. You get an option of where to save your project, but I'm just going to leave mine in the default directory. Click **Finish** and IntelliJ will open your new project in a window that looks like this:



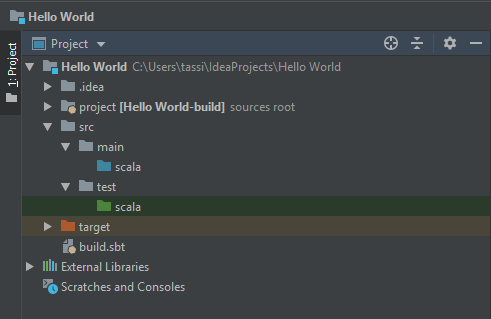
Now I know that looks pretty scary! There’s a lot going on here. IntelliJ is a very powerful and fully featured IDE, and there are even lots of features that many professional programmers don’t use. We’re going to be using just a small subset of the features throughout the course of this book, and I’ll explain each part before we use it, so take a breath… and let’s see what we’ve got.

The **Project** tab on the left hand side is like Windows Explorer, or Finder on the Mac. It shows the structure of your project, with directories and files. Directories have little icons that look a bit like paper folders, and files have different icons depending on the type of file. Directories also have little triangles to the left of them, which you can click to toggle whether the contents of the directory is shown or not.

There are two top level directories in your project: one has your project name (in my example this is *Hello World*), and the other is called *External Libraries*. The External Libraries directory contains code from other sources that is needed for your program to run, and you don’t really need to think about.

Under your project name directory there are four other directories and a file. The directories are **.idea**, **project**, **src**,and **target**. The file is called **build.sbt**. The .idea directory contains stuff that IntelliJ needs to know, but we’re not interested in. The project directory has stuff related to this project, but we won’t need to worry about anything in here for this book. The target directory is where compiled code gets created. And the build.sbt file has instructions for building our project.

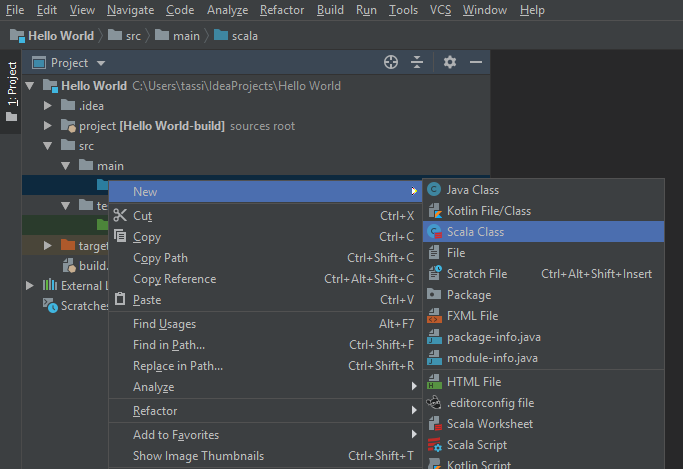
Almost everything we do is going to be inside the **src** directory, so go ahead and open it up by clicking the little triangle to the left of it. You should find a structure that looks like this:



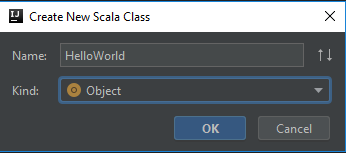
You can see that there are two Scala directories, one in blue and the other in green. When I'm talking about files or directories, if there's ambiguity about which one I mean I'll prefix them with enough of the path for it to be clear. So we've got a *src/main/scala* directory and a *src/test/scala* directory. I could also refer to these as *main/scala* and *test/scala* as that's enough to distinguish them.

All the code for our program is going to live in the *main/scala* directory. We're also going to be writing code to test that our program works properly. Can you guess where that is going to go? We won't be writing any tests for this first program, but don't worry we'll be getting to them very soon!

Right click on the *main/scala* directory, and choose **New -> Scala Class** as shown in the picture below. If the **Scala Class** option isn't there, make sure you've got the Scala framework support turned on by right clicking on the project name directory, selecting **Add framework support**, then choosing Scala.

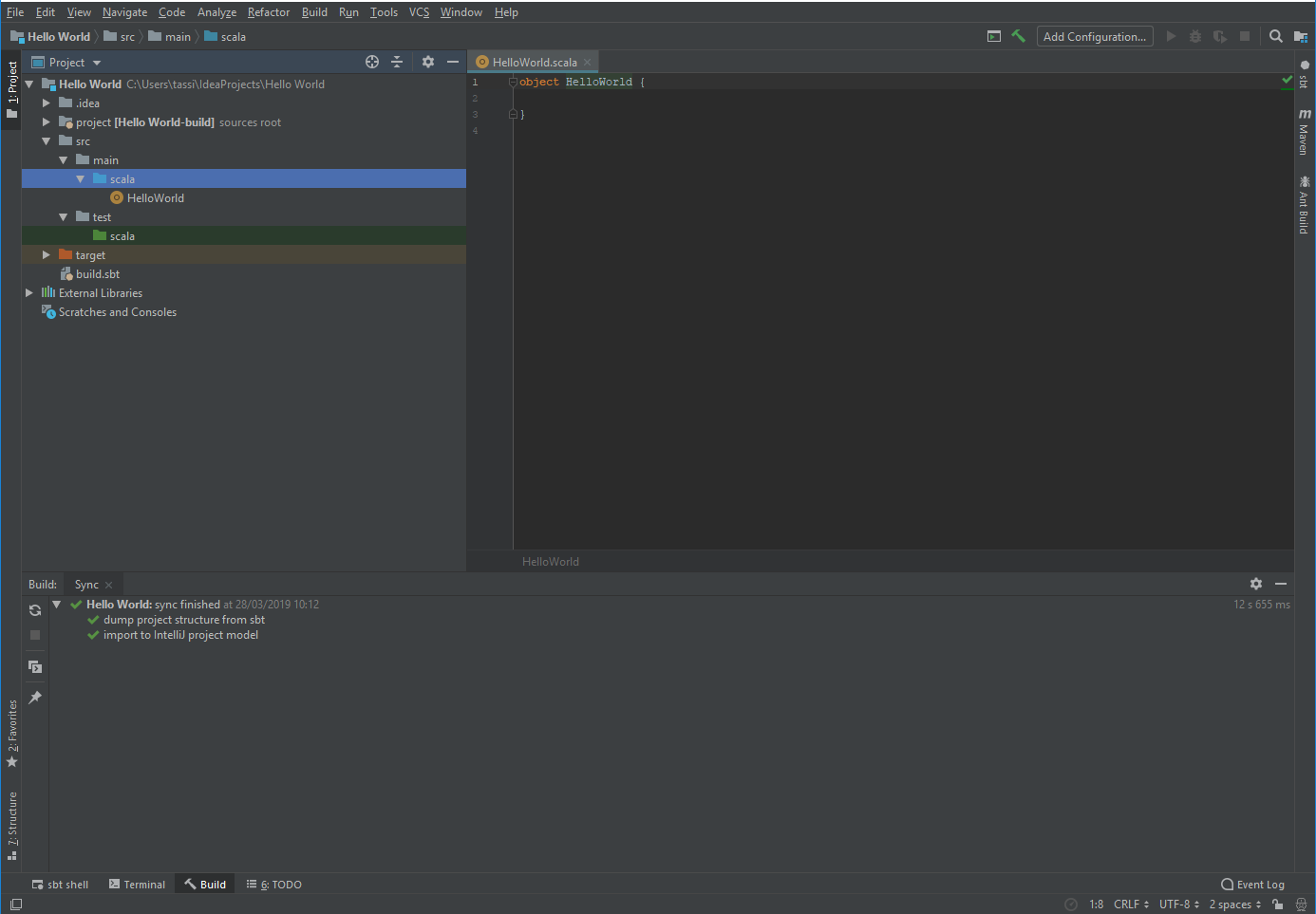


In the **Create New Scala Class** pop up, change the Kind from Class to Object, and give it a name. If you want to follow along with me, call your new object *HelloWorld*.



Note that the names of things in Scala can't have spaces in them, and object names should start with a capital letter. Most programming languages don't let you have spaces in names, and so have naming conventions to make it easier to read multi-word names. The convention in Scala is to use a capital letter for the beginning of each new word, but other languages might have different conventions such as putting an underscore between each word, like Hello\_world.

IntelliJ should now look like this:



Looking in the project view you can now see a file called *HelloWorld* in the *src/main/scala* directory. It’s got a yellow **O** next to it, which means that the file contains an object. IntelliJ can be a little confusing here, in that the file is actually called *HelloWorld.scala*, but it’s displayed as just *HelloWorld* in the project view.

You can see the full file name in the top tab of the **editor view**. This is the window to the right of the project view. The tab at the top of the editor view has the file name, and as you create more files you'll be able to move between them by selecting the different tabs, just as you do when moving between different tabs in your Internet browser.

The editor view shows the contents of the current file, which you can see isn't empty. IntelliJ has helpfully auto-generated some code to get you going. Your *HelloWorld.scala* file should contain the following code:

object HelloWorld {

}

Let's do some typing. Change the code so it looks like this:

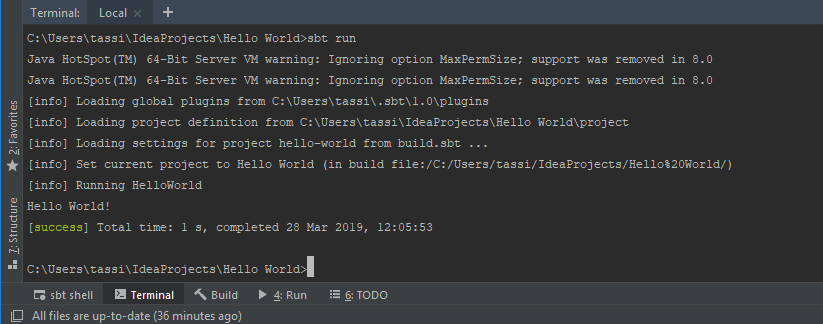
object HelloWorld extends App {

println("Hello World!")

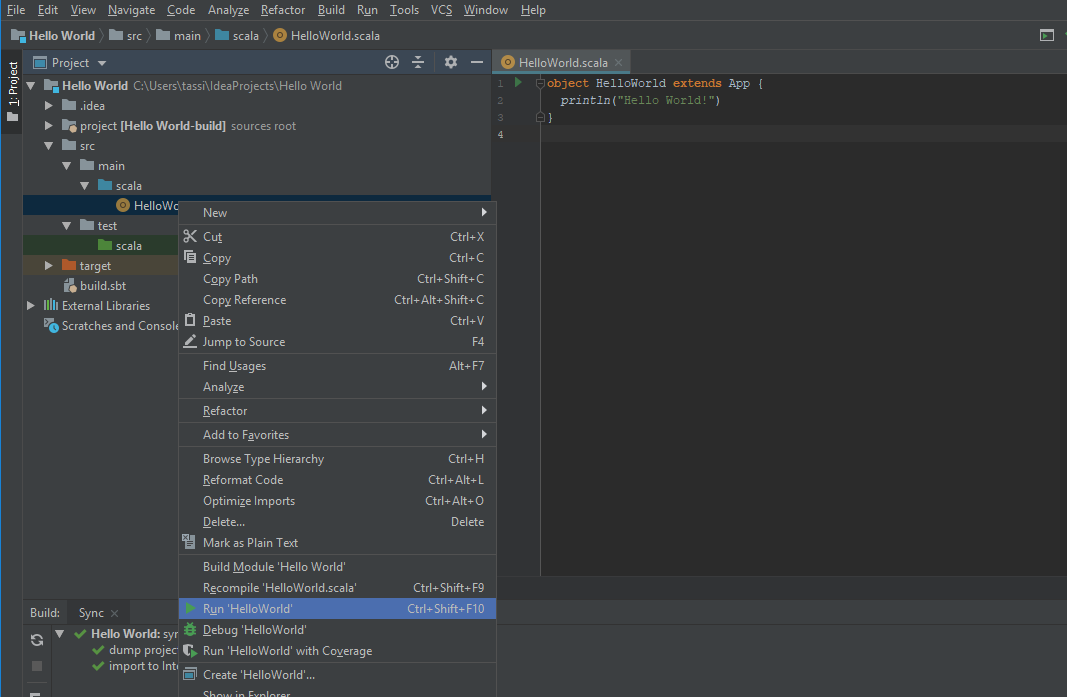
}

And that's it, you've written your first program! It's not going to break the Internet, but we've got to start somewhere. The *println* command tells the computer to print a line of text to the output. In this case the output is the command prompt or terminal window from which we run the program.

IntelliJ auto-saves, so you don’t need to worry about saving your work. The last thing to do is to run it. Click the **Terminal** tab near the bottom of IntelliJ to switch the bottom view to the terminal, then type `*sbt run*`. Hopefully you should see something like below.



Just above the *[success]* line you should see the phrase “Hello World!”. That’s the output of the program. Admittedly it's not very pretty, but congratulations, you've written your first program! Why don't you try making it say something else, or else printing a couple of greetings on separate lines. We'll go into more detail about what the code means in the next chapter, so give yourself a pat on the back and take a breather.

Once you’ve run your program from the terminal using *sbt run*, you should thereafter be able to run it in the following way if you prefer. Right click on the HelloWorld file in the project view, and select **Run 'HelloWorld'**. 

With a bit of luck it should switch the bottom window to the terminal view, and you should see your program greeting the world!

Troubleshooting

Sometimes things don't go quite according to plan. It may be that things have changed since this book was written, or your computer is set up differently to mine. The first thing to do if things aren’t working is to check for any typos. Make sure you’re copying the code exactly. If you’ve got an error in your code, IntelliJ should underline it in red. Don’t worry about matching where I put blank lines though. The computer just ignores blank lines, so feel free to add or remove them to make the code look nicer to you.

If you're still having problems, you might need to resort to searching for the type of error you're getting on Google or post a question on https://stackoverflow.com.

Objects

We're now going to be introduced to some of the fundamental building blocks we use to write programs. We've seen objects already. In fact, we created one called *HelloWorld* in the last section. But what is an object? Under the hood computers don't understand programming languages in the way that we write them. They just see a program as a series of ones and zeros, which tell them to do things at a very low level, such as "Put a 1 in memory register A", or "If memory register A contains a 1 and memory register B contains a 0, then put a 1 in memory register C". In the early days of computing people wrote programs at this level, but as computers and programs got more complex this approach became impossible to manage. Can you imagine having to think at the level of memory registers when writing something like a big budget game?

Programming languages were developed to provide higher levels of abstraction over the nuts and bolts, and one very successful approach to this abstraction is called **Object Oriented Programming**. Scala is one of these Object Oriented Programming languages.

The basic premise of Object Oriented Programming (OOP from now on) is that you should be able to think about your program in the same way as you think about real life. In the real world we're surrounded by objects: a house, a car, a saucepan, my neighbour's cat, etc etc. Objects tend to have attributes about them, and behaviour.

Let's take an example. If I go to the fridge and take out a milk carton, what are some of the attributes that this carton has? Well, it has a colour, it has a logo printed on it, it's made from a certain mix of materials, it has a capacity for how much milk it can hold, and it's currently holding a certain volume of milk. How about behaviours? Admittedly it's not doing much just sitting on the breakfast table, but I can pour some milk out of it, which would mean that the attribute for how much milk it currently contains would change. Other behaviours might be adding milk to it, or changing the logo on it.

OOP encourages you to build your program out of a collection of objects, each with attributes and behaviour, and get them to interact with each other to do something useful.

Let's take a look back at the HelloWorld program we wrote. The code looked like this:

object HelloWorld extends App {

println("Hello World!")

}

There are actually many objects involved in this tiny program. The first is easy to see. It's called *HelloWorld* and you created it by typing the word *object* followed by the name you wanted to give the object, with the body of the object between curly braces. Ignore the *extends App* for a moment. I'll explain it in a second.

The object body can contain attributes (describing the characteristics and current state of the object) and behaviour. Our object doesn't have any attributes, but it does have some behaviour that causes "Hello World!" to be printed to the terminal.

But hang on a second! You said there were many objects? Where are the others? Scala comes with loads of its own objects which you'll take advantage of to write your programs. *println* is actually some behaviour that is defined in one of these objects called *Predef*. If you hold **Ctrl** on Windows or **Cmd** on the Mac, and click on the word *println*, IntelliJ will open up the file containing the definition of this behaviour. You'll be in the body of the *Predef* object, and should be looking at the follow code:

def println(x: Any) = Console.println(x)

This is behaviour written down as a **method**. Objects can have two types of behaviour. The first is behaviour that runs as soon as the object is created, like our *println("Hello World!")*. This is any code within the object body, but not within a method. The second type is behaviour that is only run if it is specifically called from somewhere else in the program, and methods are of this second type. The vast, vast majority of behaviour you'll write will be methods.

In a similar way to how we create objects with the word *object* followed by the name we want to give our object, we create methods with the word *def* followed by the name we want to give our method. After that there's some difference though. Notice the *(x: Any)* straight after the method name. When you call a method you can pass it some data, which it might need to be productive. Going back to our milk carton example, we said there was some behaviour for pouring milk. We might express this as a method in which we can specify how much milk we want to pour.

Everything on the left hand side of the *`=`* is called the **method definition**, and everything on the right hand side is called the **method body**. The method definition specifies the method name, what data it takes in when it is called, and what data it might return to the caller. The method body is where you describe how the method should actually do what it needs to do.

Ignoring the method body for now, our milk carton object with a method definition might look like this:

object MilkCarton {

def pourVolumeOfMilk(millilitres: Int) =

}

You've now seen two examples of defining the data to be passed into a method: *(x: Any)* and *(millilitres: Int)*. These are called **method parameters**, and they are written inside parentheses directly after the name of the method. They have two parts, separated by a colon. The first part is the parameter name, which in our examples are *x* and *millilitres*. The second part is the type of the parameter.

There are some types included with the language, such as *Int* which stands for integer, meaning a whole number, *Float* which represents a floating point number, meaning a number with a decimal point, and *String* which represents text like "Hello World!". In the *pourVolumeOfMilk* method, we've specified that the method parameter is of an *Int* type, as it wouldn’t make sense to call the method with a string for example. What would *pourVolumeOfMilk("HelloWorld!")* mean?

In the *println* method, the *Any* type is a catch all, meaning you're allowed to pass anything to it. In our program we're actually calling it with a String (the text "Hello World!"). You can think of parameters as containers with different shapes. Maybe an Int container is long and thin, and a String container is short and fat. So a long, thin integer will fit nicely in an Int container, but won't squeeze into a String container. And each parameter container has the parameter name stamped on the outside. So in the method body we can refer to the parameter by name, and the computer will go find the container with the right name, and get its contents out for us to use.

Let's have a look at what's happening in the method body for the *println* method in the *Predef* object. The method body is *Console.println(x)*. This is what the computer will run when we call the *println* method on the *Predef* object from our *HelloWorld* object. This method body is actually showing the standard way of calling a method on an object. *Console* is another object, and it also has a method called *println*. The syntax for calling methods on objects is to write the object name, followed by a period (i.e. a full stop), followed by the method name, with method parameters within parentheses.

If you like you can look at the *println* method in the *Console* object, by holding **Ctrl** or **Cmd** and clicking on the method name. You can see that our simple program is actually working by calling a method on an object, which is calling a method on another object, and so on and so on until eventually we'll get to some object that knows how to actually do the writing to the terminal!

So now we know enough to understand what's going on in the *Predef* object. The whole method looks like this:

def println(x: Any) = Console.println(x)

We've got a method definition creating a method called *println* which takes a single parameter called `*x*`, which can be of *Any* type. We're calling this method from our *HelloWorld* object like this:

println("Hello World!")

So we're passing in a String type parameter with the value "Hello World!" to the *println* method, where it gets put in the parameter container with the name "x" stamped on the outside. When the computer sees the `*x*` in the method body, it goes and finds the parameter container called "x", gets the value out of it (i.e. the string "Hello World!"), and passes that value as a method parameter to the *println* method on the *Console* object.

Phew, that was quite complicated! Well done if you got that first time. If not, you might want to re-read that section, but don't worry too much. It will all become a lot clearer when we start writing some code.

There's one final confusing thing we should cover before we move on. Some of you might be thinking, "He told me that the syntax for calling methods was the object name, followed by a period, followed by the method name. But we're not using the Predef object name when we call *println* on it. Huh?" Well, Predef is a special object and doesn't play by the normal rules, so you can call any of the Predef methods without having to use the object name first. In other words, *println("Hello World!")* and *Predef.println("Hello World!")* are exactly equivalent.

Oh, and I almost forgot! I said I'd explain what that *extends App* thing is. When you run a program, the computer needs to know where to start it from. You might have hundreds of files, loads of objects, thousands of methods, so how do you specify where the beginning is? Well, you're allowed to write *extends App* for just one of your objects in the whole program, and that signifies to the computer that it should start running the program from this object. There's actually one other way of specifying the start point of your program, and that's to write a method called `main` in one of your objects. This is a special method that has a specific method definition, and can only appear once in your whole program. If we were to rewrite our *HelloWorld* object to use the *main* method, it would look like this:

object HelloWorld {

def main(args: Array[String]) = {

println("Hello World!")

}

}

When you run the program, the computer looks for this method called *main* which takes a single method parameter. The parameter is traditionally called *args* which is short for "arguments", and its type is *Array[String]*. We'll learn about arrays later, but you can think of them like a list. So in this example, *args* is like a list of Strings. It's a way for your program to access arguments that can be passed in to it when you run it from the command line. You don't need to understand this for now, so if it doesn't make sense don't worry, we'll be covering it later. Note that this method body is inside curly braces, whereas the *println* body wasn't. Braces are optional if you can fit the method body on a single line, but have to be there if your body takes up more than one line.

Test Driven Development

We're going to play around creating some objects and methods, but first I'd like to talk to you a bit about testing. This is something that's not often covered in beginner's programming books, but it's a hugely important skill to have as a professional coder, so I think it's good to get used to it right from the start.

Programs can be enormously complicated, and when you're releasing programs into the wild you want to minimise the chance that they don't work the way you intended them to. The best way to make sure a program is doing what you want it to is to test it. Now you could sit down with the running application and manually walk through every scenario you can think of, and then repeat that every time you make a change to the code to make sure the change hasn't broken anything. Or you could write another program to automatically test your program. A program that never gets bored, doesn't get tired and make mistakes, and runs through all the scenarios in the fraction of the time it would take a person. I know which one I'd rather do.

There are whole books written on Test Driven Development (or TDD), so I'm only going to lightly scratch the surface here. More like scuff it a bit. But the basic concept is that you think about what you want your program to do, you specify how you expect that behaviour to work by writing a test for it, and then you write the code to make the test pass.

Scalatest

We're going to be using a test framework called **Scalatest** which will handle running our tests and telling us whether they've passed or not. We'll need to install it as a dependency in our project. Dependencies are external pieces of code that you can use in your project, but that don't come packaged up with the standard install of Scala, so you need to download them into your project. Fortunately it's very easy to do this using the Scala Build Tool (sbt).

Open up the *build.sbt* file which you can see in the project view. We're going to add a line to the bottom of this file to tell it to include Scalatest as a dependency. You can find the command to install the latest version of Scalatest by Googling for "Install Scalatest". As of the time of writing the command is `*libraryDependencies += "org.scalatest" % "scalatest\_2.12" % "3.0.5" % "test"*`. The “2.12” refers to the version of Scala you are using, so make sure the *scalaVersion* in the file starts with “2.12”. Your *build.sbt* file should look something like:

name := "NameOfMyProject"

version := "0.1"

scalaVersion := "2.12.0"

libraryDependencies += "org.scalatest" % "scalatest\_2.12" % "3.0.5" % "test"

Click out of the *build.sbt* file, and IntelliJ should prompt you to import the dependency. Just click to enable auto-imports. If you don't get the prompt, go to the terminal view and run `**sbt update`**. It may take a couple of minutes to download.

Writing a test

First I'm going to think about what I want my program to do, then I'm going to write a test for it, then write the code to make it work. My program is going to be very simple. I want an object to represent myself, so I'll call it "Ian". And I want Ian to be able to introduce himself, so I'd like a method that will return the string "Hello, I'm Ian" when it's called. And that's it for now. So we can start writing our test.

Feel free to look around the Scalatest website. There are many different styles of writing tests, so you might decide you prefer a particular one later, but for now we're going to use FlatSpec as I think it has a nice syntax.

I'm going to create my first test file, then I'll explain what's going on. Right click on the *src/test/scala* directory in the project view, and select **New -> Scala Class**. Leave the type as Class, and call the file IanSpec (or you can use your own name instead of Ian!) Copy the following into your file:

import org.scalatest.\_

class IanSpec extends FlatSpec with Matchers {

"SayHello" should "return 'Hello, I'm Ian'" in {

Ian.sayHello() shouldBe "Hello, I'm Ian"

}

}

I'm going to walk through what's happening here, but to fully understand it you'll need concepts that we're not going to cover until later, so it doesn't matter if you don't really get it yet. You can treat it like magic for now.

The first line is telling the computer that we'll be using stuff from Scalatest in this file. We're then creating a class called IanSpec. We'll learn a lot more about classes very soon, but at the moment you can think of them as similar to objects. I know I'm going to be creating an object called "Ian", and this file is all about testing that Ian object, so I'm going to follow the FlatSpec convention of naming the class with the name of the thing I want it to test followed by the word "Spec".

The *extends FlatSpec with Matchers* is telling the computer that we're going to be using some specific stuff from Scalatest inside this class. There's an opening curly brace at the end of the class definition line, and a matching closing curly brace at the end of the file. Everything between these is the body of the class, where the tests live.

The tests have their own special language, which isn't the standard Scala syntax you'll use in your actual program. The structure of the test is:

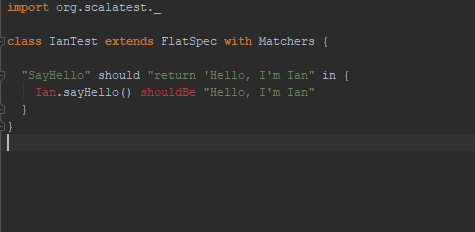
*a string describing what you're testing - the word* ***should*** *– a string describing what the behaviour you're testing should be - the word* ***in*** *- then the test body between curly braces.*

So you should be able to tell that this test is about making sure that the *sayHello* method of the *Ian* object will return the string "Hello, I'm Ian".

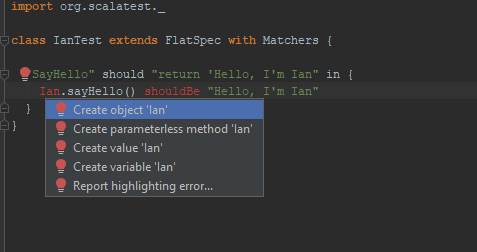
In the test body, we start by calling the method we want to test. Remember the syntax for calling methods on objects? It's object name, followed by a period, followed by the method name with parenthesis straight after. Previously we saw this with *Console.println(x)*, and here we're calling *Ian.sayHello()*. I don't need to pass any parameters into this method, as it wouldn't do anything with them, so the parentheses are empty.

The *shouldBe* compares what's on the left of it after the method call (i.e. the result of calling *Ian.sayHello()*) to what's on the right of it (the string “Hello, I’m Ian”). If they match up then the test passes. If they're different the test will fail.

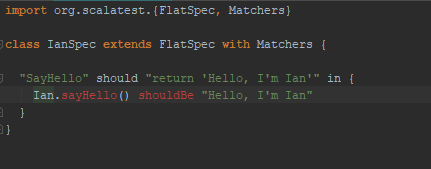
You should be getting some red text in IntelliJ, signifying that something's wrong. It probably looks something like this:



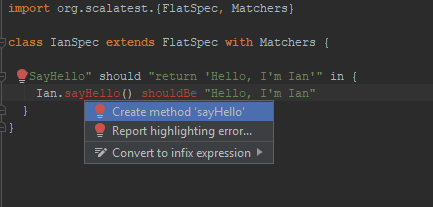
That's because we haven't created an actual object called Ian yet, so IntelliJ sees that we’ve written *Ian*, can't find what it refers to, and flags up a problem. We could fix this by right clicking on the *src/main/scala* directory, selecting **New -> Scala Class**, changing the kind to **Object**, naming it Ian, and clicking **OK**. But there's an easier way. IntelliJ has loads of clever shortcuts to help you work faster. If you click on the red Ian text you can either press **Alt + Enter** or click on the little red lightbulb that appears, then select **Create object 'Ian'** from the drop down list, like so:



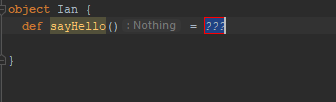
It asks you where you want to put the object, so select **New File**. Unfortunately it's not clever enough to put it in the *main/scala* directory, so look in the project view and you'll see your new file in the *test/scala* folder. Just drag it up into the *main/scala* directory, and hit **Refactor** in the box that pops up. Go back to the test file, and you'll see that *Ian* looks ok now, but *sayHello* is now red.



You've guessed it. That's because there's no *sayHello* method in the Ian object. Let's do the same trick again. Click on the method name, then **Alt + Enter** or click on the red lightbulb, and select **Create method 'sayHello'**.



Your object should now look like this:



Those *???* mean there's no method body yet, so it’s a reminder for you to write something. You might also have spotted something different about the method definition. What's that *:Nothing* after the parentheses? This is the return type of the method. Remember how method parameters have types? Our carton of milk had a *def pourVolumeOfMilk(millilitres: Int)*, where the millilitres parameter had a type of Int. Most useful methods return something to the caller of the method, and the thing that's being returned has a type. For instance, we might have a method that takes an Int, and returns another Int with the value of double the input. That would look something like:

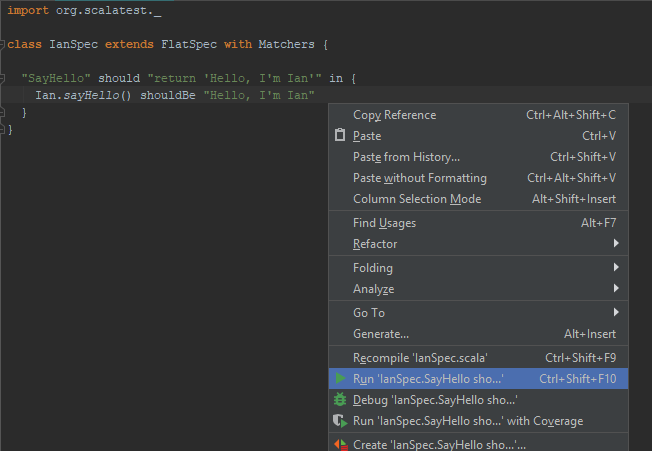
def double(x: Int): Int = x + x

In Scala, most of the time you don't have to type in the return type, as it can work it out for itself. But IntelliJ will helpfully display what it thinks is the return type for you anyway. In this instance we haven’t written a method body yet, so IntelliJ is saying the return type is Nothing.

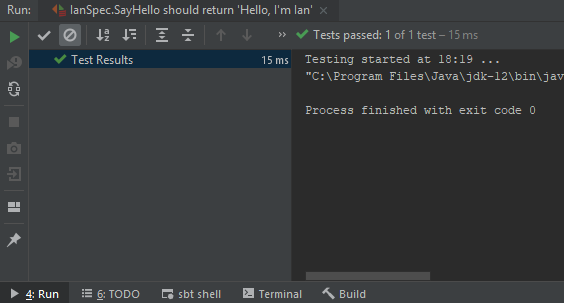
Ok, let's write some code instead of those question marks. I'm actually going to write something that will make the test fail to start with, just so we can see what a failing test looks like. My method will look like this:

def sayHello() = "Hello, I'm Snoopy"

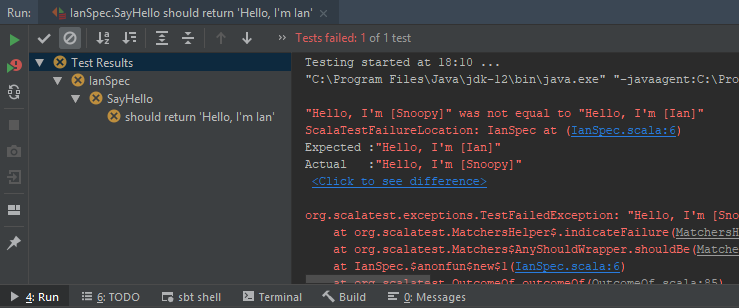
Go back to your test file, and all that red stuff should have gone. Hopefully we're all good to go! Right click somewhere in your test, or on the name of the test file in the project view, and select **Run 'IanSpec.SayHello sho...'**, or press **Ctrl + Shift + F10**.



In the Run view at the bottom of IntelliJ you should see the test output, saying there's a failed test and some info about it. There's a lot of stuff in the output you don't need to worry about, but you can see the explanation for why it failed. It was expecting "Hello, I'm Ian", but got "Hello, I'm Snoopy". You can even click the **Click to see the difference** link for a nice comparison of the expected and actual result.



Let's fix our code so our test can pass. Go back to the Ian object, and change "Snoopy" to "Ian", then rerun the test. It's a bit small to see, but everything's green which is a good sign, and you can see that 1 test passed out of 1 test run, and on my laptop it took 15 ms to complete the test.



Now we can be sure that the `Ian.sayHello` method will always return "Hello, I'm Ian", and if you accidentally change it while you're coding you'll find out about it as soon as you run your tests. Result!

The last thing we're going to do in this chapter is use our new object and method from our actual program rather than from the test. I'll leave it to you as an exercise, but try to get your HelloWorld object to print "Hello, I'm Ian" to the terminal by calling the *sayHello* method on the *Ian* object.

More methods, and variables

Let's get some more practice with writing methods and tests. We'll continue with our project from the last chapter, so we should have an *Ian* object in the *src/main/scala* directory and an *IanSpec* class in the *src/test/scala* directory. I want to expand the behaviour of our *Ian* object so that it can introduce itself to other people by their name. This will be the first dynamic behaviour we code, whereby a method can return different things depending on what's passed into it.

A well written test should be just as good at describing behaviour as written prose, so I'm going to go ahead and write down how I'd like this to work in a new test case. This will go in the class body of the *IanSpec* class. You can put it below the existing test case, so your file should look something like this:

import org.scalatest.\_

class IanSpec extends FlatSpec with Matchers {

"SayHello" should "return 'Hello, I'm Ian'" in {

Ian.sayHello() shouldBe "Hello, I'm Ian"

}

"SayHelloTo(name)" should "return 'Hello (name), I'm Ian'" in {

Ian.sayHelloTo("Snoopy") shouldBe "Hello Snoopy, I'm Ian"

Ian.sayHelloTo("Linus") shouldBe "Hello Linus, I'm Ian"

}

}

You should have almost all the tools you need to write the code to make this test pass on your own now. Remember, you can use IntelliJ to generate method definitions for you. If you do this, it will generate a name for your method parameter (it calls mine *str*), which will be highlighted within a red box. You can either press **Enter** to keep the name it chose for you, or you can type a new name. *str* isn't very descriptive, so I've changed mine to *name*.

You might need a bit of a hand with the method body. We're going to be referring to the method parameter when we're inside the body, inserting its value into the return string. There are multiple ways of manipulating strings. If you want to create a new string by joining two strings together you can use the *+* operator between them.

def joinStrings() = "Hello " + "Snoopy"

If we call *joinStrings()* it will return a single string with the value "Hello Snoopy".

def joinStringsWithParameter(name: String) = "Hello " + name

If we call *joinStringsWithParameter("Snoopy")* it will return a single string with the value "Hello Snoopy". But this method is now dynamic, and can change what it returns depending on what is given to it. We can call *joinStringsWithParameter("James Bond")* and we'll get back a single string with the value "Hello James Bond".

The method we're writing needs to insert a parameter value between the strings "Hello " and ", I'm Ian". We can use the *+* operator to join as many strings as we like, so something like *"Hello " + name + ", I'm Ian"* works perfectly well, but isn't that readable.

A nicer way of doing this is called *string interpolation*, which means writing out the finished string as you'd like it to look, but referring to any parameters directly inside the string. The computer will see the string interpolation pattern and replace the parameter names with their values. The syntax for this is to write a small *s* immediately before the opening quotation mark for the string, and then prefix any parameter names within the string with a *$* sign. An example should help.

def interpolateStrings(name: String) = s"Hello $name, I'm Ian"

Not the most beautiful thing ever, but definitely more readable than using lots of *+*. (Joining strings with *+* is called *concatenation*, by the way).

Now you definitely have all the tools to make that test pass! Have a go at doing this yourself. My solution will be below.

What d'you know?

Let's have a little recap of what you've learnt so far. You know how to set up new projects, how to install a test framework, how to run a program and where the computer looks for the entry point to the program, how to print stuff to the terminal, how to create objects, how to specify object behaviour using methods, what *types* are, what method parameters are, how to call methods on objects, how to concatenate and interpolate strings, how to write and run tests, how to interpret test failures, what TDD stands for and how to do it, how to use IntelliJ to generate methods for you, and that your code is built on top of a ton of pre-existing classes, objects and methods that you can look at with a **Ctrl + Click**.

Not bad!

Back to methods

There are two more things I'd like to talk about before we leave learning about methods: multi-parameter methods and method overloading.

So far we've only seen methods that take no parameters (like *def sayHello()*), or take a single parameter (*def println(x: Any)* or *def interpolateStrings(name: String)*). But what if we need to add up two numbers? Or take in integers representing red, green and blue values in order to render the right colour pixel for a game? Well fortunately methods can take more than one parameter, and it's really simple to do. Just separate the parameters with a comma. Let's see an example to get some practice.

I'd like to make a calculator object, that has methods that add and subtract numbers. They will both take two integer parameters. The add method will return a new integer that is the sum of both the parameters, and the subtract method will subtract the second number from the first and return the result. Maths operations work the way you would expect. If you want to add two numbers, use the`*+*` operator. Use *`-`* if you want to subtract.

You should have the knowledge to do this on your own now. Remember to start with a test. We're going to be creating a Calculator class, so we'll want a new test file called CalculatorSpec. The test case for the *add* method should have a description along the lines of *"Add" should "return the sum of the two parameters" in { }*. You can use **Alt + Enter** to get IntelliJ to generate objects and methods for you (but remember it puts new objects in the test directory rather than the main directory, so you'll need to move them).

My solution is a bit further down, but try and write this yourself before moving on.

Method overloading

When you call a method from somewhere in your code, the computer looks up the object the method is defined in, then looks up the specific method to call. As you've seen, an object can have more than one method in it. However, it doesn't just go by the method name to identify the right method. It also looks at the method parameter types. So you could have an object with lots of methods, all with the same name, but with differences in their parameter types, and the computer would still be able to tell them apart.

For instance, it would be alright to have all of the following methods in the same object, and the computer could find the right one depending on the parameter types it is being called with:

object StuffWithMethods {

def add(x: Int, y: Int) = x + y

def add(x: String, y: String) = x + y

def add(x: Int, y: String) = ???

def add(x: Int) = ???

}

So if we called *StuffWithMethods.add(1, 2)* it would see we are passing in two *Ints*, so would choose the top method and return 3. If we called *StuffWithMethods.add(“Hi”, “there”)* it would choose the method that takes two *Strings* and return “Hithere”.

Just a word of caution though. Method overloading works when you've got different parameter types, but it doesn't work when only the parameter names or the method return types are different. You'd get errors if you tried to put these methods in the same object:

object WontWork {

def add(x: Int, y: Int): Int = ???

def add(a: Int, b: Int): Int = ???

def add(x: Int, y: Int): String = ???

}

Try adding an overloaded method to your calculator object that will take two strings, and return them concatenated. Don't forget to write the test first!

Ok, let's just make sure we're still on the same page. You should have an *Ian* object looking like this:

object Ian {

def sayHello() = "Hello, I'm Ian"

def sayHelloTo(name: String) = s"Hello $name, I'm Ian"

}

A *CalculatorSpec* test class like this:

import org.scalatest.\_

class CalculatorSpec extends FlatSpec with Matchers

"Add" should "return the sum of two integers" in {

Calculator.add(1, 2) shouldBe 3

}

"Subtract" should "return the difference between two integers" in {

Calculator.subtract(4, 3) shouldBe 1

}

"Add passing in strings" should "return the concatenated strings" in {

Calculator.add("Hello", "World") should be "HelloWorld"

}

}

And finally the implementation of the Calculator object should look like:

object Calculator {

def add(x: Int, y: Int) = x + y

def subtract(x: Int, y: Int) = x - y

def add(x: String, y: String) = x + y

}

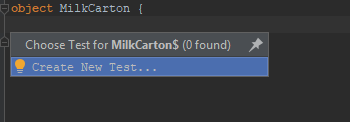
Let's talk about state, baby

Sorry for the cheesy title! You might remember when we were talking about Object Oriented Programming that I said that objects can have attributes. For our example of a milk carton, these were things like its colour, the logo printed on it, and how much milk it currently contains. These are all things that programmers would say describe the state of an object.

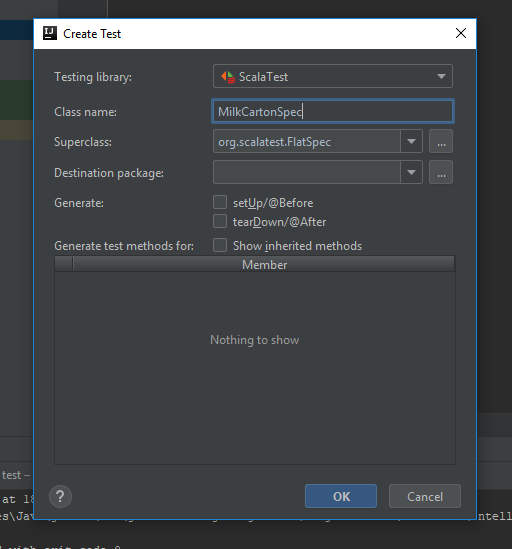
When we're modelling these objects in code, we might decide that some of these attributes will never change, such as the colour. Other attributes are likely to vary over time, such as the amount of milk the carton currently contains. In programming-speak, attributes that can vary are called **mutable**, and are created using the keyword *var*. Those values that are fixed are called **immutable**, and are created with the keyword *val*.

We're going to demonstrate the use of mutable and immutable attributes by modelling our milk carton in code. It will have an immutable attribute to represent its colour, and a mutable attribute representing how much milk it contains. We'll then code the behaviour for pouring and adding milk in methods, and see how we can link these methods to change the state of the object. You can keep using the existing project, or create a new one if you like.

I'm going to show you a slightly different way of creating objects and tests, meaning you won't have to move the auto-generated object from the test directory to the main directory. Right click on the *src/main/scala* directory, and select **New -> Scala Class**. Change the kind to Object, name the object MilkCarton, and click **OK**. IntelliJ should open up your new MilkCarton object in the editor view. Put the cursor somewhere in the object body and press **Ctrl + Shift + T**. You should get a drop down box with the option to create a new test:



Select this option, and you'll get a box pop up with some options for your new test. Make sure the class name is *MilkCartonSpec* and the Superclass says *org.scalatest.FlatSpec* and click **OK**:



And voila, IntelliJ has created a new test file for you in the right place!

It's still not quite perfect for us though. We've got *class MilkCartonSpec extends FlatSpec* but you need to add *with Matchers*. If you type *Matchers* and it's in red, make sure it's being imported into the file. In other words, the first line of the file should be *import org.scalatest.{FlatSpec, Matchers}* or *import org.scalatest.\_*.

Right, we're ready to start writing some tests. First we're going to check that we can get the colour attribute out of the object. We reference attributes on objects in the same way that we reference methods on objects. The only difference being that attributes don't have parentheses after their name. It doesn't make much sense to pass a parameter into an attribute. Follow along with me as we write this first test.

import org.scalatest.\_

class MilkCartonSpec extends FlatSpec with Matchers {

"The MilkCarton's colour" should "be white" in {

MilkCarton.colour shouldBe "white"

}

}

Hopefully that makes sense. Because we haven't defined the *colour* attribute on the *MilkCarton* object, IntelliJ is highlighting the word `colour` in red, as it can't work out what it's referring to. We know how to fix that! Put your cursor in the word `colour` and either **Alt + Enter** or click on the red lightbulb to get the options drop down. Select **Create value 'colour'**, and IntelliJ will insert some code into your MilkCarton object for you:

val colour = ???

You can see the keyword *val*, which means we're creating an immutable attribute. Replace the *???`*with the value we want our attribute to have, which in this instance is the string "white".

val colour = "white"

Go back to your test file and run the test, either by right clicking on the file name and selecting the option to run the test, or clicking somewhere in the test body and pressing **Ctrl + Shift + F10**. With a bit of luck you've got a passing test!

Try changing either the expected value in the test, or the actual value in the object to make the test fail. Cool, now put it back so it's passing again. You've just created your first attribute, and tested that you can get its value out of the object!

Now we're going to do something more complicated, but satisfying. I'd say this is going to be our first *real* coding that does something interesting! We're going to create a mutable attribute to represent the state of how much milk the carton currently holds. Then we're going to create methods that can either add or remove specified amounts of milk, and check that the state of the carton gets updated properly to reflect the new amount of milk it contains. How exciting!

We need to make a decision about how much milk the carton will contain when it is created. It makes sense to me that it will initially be empty. So let's write a test to make sure that a newly created carton contains no milk. The test case will be something like:

"A newly created MilkCarton" should "contain no milk" in {

MilkCarton.millilitresOfMilk shouldBe 0

}

Use IntelliJ to generate the attribute for you, but this time instead of selecting **Create value**, select **Create variable**. This is because we want a *var* instead of a *val*, which will let us change its value over time. Set your new variable to equal 0, like this:

var millilitresOfMilk = 0

And make sure your test passes. Now we need a way of adding milk, which is going to be a method. The way we're going to test that this method works is to call it to add milk, then check that the value of the millilitresOfMilk is equal to the amount of milk we added. So I'm going to add a new test case, like this:

"Adding milk" should "update the state of the MilkCarton" in {

MilkCarton.millilitresOfMilk shouldBe 0

MilkCarton.addMillilitresOfMilk(500)

MilkCarton.millilitresOfMilk shouldBe 500

}

We start by making sure that the MilkCarton is empty. Then we call the method to add 500 mls of milk. And finally we check that the MilkCarton contains 500 mls of milk. Go ahead and get IntelliJ to generate the *addMillilitresOfMilk* method for you. This is how we are going to use the method to update the state of the carton:

object MilkCarton {

var millilitresOfMilk = 0

def addMillilitresOfMilk(mls: Int) = millilitresOfMilk = millilitresOfMilk + mls

}

Run your test and make sure that it passes, then we'll come back to this.

It looks a little bit confusing, doesn't it. The first thing to say is that *=* doesn't mean equals in the way you learnt it at school. In maths we'd say 1 + 1 = 2, which is testing that the left side is the same as the right side. In Scala (and many other programming languages) the *=* sign is used for assignment. It means *assign the value on the right hand side to the variable or method on the left hand side*. So *var millilitresOfMilk = 0* means assign the value 0 to the variable called *millilitresOfMilk*. In our *addMillilitresOfMilk* method, the first *=* means assign everything on the right hand side to the body of the method. It might look a little clearer if I rewrite it with the optional braces around the method body:

def addMillilitresOfMilk(mls: Int) = {

millilitresOfMilk = millilitresOfMilk + mls

}

In the method body, the first *millilitresOfMilk* on the left hand side of the *=* is referencing the *var millilitresOfMilk*. The *=* means assign the result of the right hand side to the *millilitresOfMilk* variable. It's a bit confusing because *millilitresOfMilk* is on both sides. My fingers are beginning to regret calling that variable *millilitresOfMilk*.

Let's change it to be a bit simpler. If we wrote:

millilitresOfMilk = 1 + 2

you could see that we're assigning the value 3 to the millilitresOfMilk attribute. If we wrote:

def addMillilitresOfMilk(mls: Int) = {

millilitresOfMilk = 1 + mls

}

and we called the method passing in a parameter of 10, the computer would see that we're referencing the *mls* parameter on the right hand side, go get the value of that parameter (10), add it to 1 to get 11, and assign that to the *millilitresOfMilk* attribute. So going back to our original method:

def addMillilitresOfMilk(mls: Int) = {

millilitresOfMilk = millilitresOfMilk + mls

}

and say we call the method with a parameter of 10, the computer will evaluate the expression on the right hand side of the *=* by finding the current value of the *millilitresOfMilk* variable (0), finding the value of the *mls* parameter (10), adding them together to get 10, and assigning that result back to the *millilitresOfMilk* attribute. If we then called the method again with a value of 10, the current value of the *millilitresOfMilk* would get updated to 20. To illustrate this, let's update our test to show it in action:

"Adding milk" should "update the state of the MilkCarton" in {

MilkCarton.millilitresOfMilk shouldBe 0

MilkCarton.addMillilitresOfMilk(10)

MilkCarton.millilitresOfMilk shouldBe 10

MilkCarton.addMillilitresOfMilk(10)

MilkCarton.millilitresOfMilk shouldBe 20

MilkCarton.addMillilitresOfMilk(50)

MilkCarton.millilitresOfMilk shouldBe 70

}

You can see that the carton is keeping track of how much milk it contains, and each time we add some it updates itself with the new value. Updating a variable by adding something to its existing value is such a common thing that there's actually a shortcut way of writing it, using the *+=* operator. So instead of:

millilitresOfMilk = millilitresOfMilk + mls

we can write:

millilitresOfMilk += mls

You can also use this shortcut for subtraction, multiplication and division. Here's a little self-contained program that you can run to demonstrate this.

object MyProgram extends App {

var x = 10

x += 100

// x now equals 110

x -= 10

// x now equals 100

x /= 2

// x now equals 50

x \*= 4

// x now equals 200

println(x)

}

This will only work with mutable variables (*var*), not with immutable values (*val*), as once a *val* has been assigned a value it can't be changed.

A comment on comments

You probably noticed something new in the little program above. There are lines of text beginning with a double forward slash (*//*). These are *comments*, which is a way of writing text for humans to read in your program, but which the computer will ignore. Anything after a *//*, but on the same line, is treated as a comment. So I could have written:

x += 100 // x now equals 110

If you want to write comments that go over multiple lines, you start the comment with */\**, and finish it with *\*/*, like so:

object MyProgram extends App {

/\*

This comment

goes over

several lines

\*/

}

Back to milk

Just to make sure we're in the same place, we should now have a *MilkCartonSpec* test file that looks like this:

import org.scalatest.\_

class MilkCartonSpec extends FlatSpec with Matchers {

"The MilkCarton's colour" should "be white" in {

MilkCarton.colour shouldBe "white"

}

"Adding milk" should "update the state of the MilkCarton" in {

MilkCarton.millilitresOfMilk shouldBe 0

MilkCarton.addMillilitresOfMilk(10)

MilkCarton.millilitresOfMilk shouldBe 10

MilkCarton.addMillilitresOfMilk(10)

MilkCarton.millilitresOfMilk shouldBe 20

MilkCarton.addMillilitresOfMilk(50)

MilkCarton.millilitresOfMilk shouldBe 70

}

}

And a *MilkCarton* file that looks like this:

object MilkCarton {

val colour = "white"

var millilitresOfMilk = 0

def addMillilitresOfMilk(mls: Int) = millilitresOfMilk += mls

}

Try adding a method to remove milk from the carton. You might want to call it *pourMillilitresOfMilk*, and it should take an integer as a method parameter specifying how much milk to remove.

Remember to write a test for it first.

And that's it for this chapter! Well done, you've made great progress in learning the building blocks of Object Oriented Programming. You know how to create objects, how to give them attributes that may or may not be allowed to change, and how to use methods to describe the behaviour of and change the state of objects.

This is exactly how a computer game might track and update the state of characters. You'd probably have an object representing the player's character. Some of the player's attributes might not change, such as its walking speed, or the image used to display it, so these could be modelled using *vals*. Others will need to change throughout the course of the game, such as the amount of health it has, or the number of coins it has picked up, so these would be *vars*. And the state of the player would be updated by calling methods on the player object. For instance, you can imagine a *def takeDamage(x: Int)* method that would update the player's health. There are a few more concepts to learn before you can make that big budget role playing game, but you're well on your way!

Try writing a *Player* object with some health that you can take from and add to using methods.

Classes

I'd like to continue our example from the last section in which I asked you to create a *Player* object, in order to cover another crucial aspect of Object Oriented Programming. You can use your own *Player*, or copy mine. Just make sure it has a *var* representing its health as an integer, a *val* representing the amount of damage it can deal, and methods to take damage and to heal. Mine looks like this:

object Player {

var health = 100

val damage = 10

def takeDamage(x: Int) = health -= x

def heal(x: Int) = health += x

}

Now let's say we want to create a fantastic game, where the player goes around fighting monsters. I'm going to write a *Monster* object. It will be a bit less powerful than the player, so I'll give it less health and damage. Also, we don't want it to be able to heal. So it looks something like this:

object Monster {

var health = 50

val damage = 5

def takeDamage(x: Int) = health -= x

}

There's obviously a lot missing if this was going to be a real game, such as being able to tell when the health has got down to zero and killing either the player or the monster, but I'm going to keep things simple.

A fight between the player and the monster could look like this:

object Game extends App {

// Player hits first

Monster.takeDamage(Player.damage)

println(s"Monster now has ${Monster.health} health"

// Monster's turn

Player.takeDamage(Monster.damage)

println(s"Player now has ${Player.health} health")

}

There's something slightly new here. Notice the curly braces around *{Monster.health}* and *{Player.health}*. This is still string interpolation, like we did before (*s"Hello $name"*), but if you're interpolating anything more than a simple variable you need to wrap it in curly braces. Try removing them and see what happens.

So that's great. We could keep getting the player and the monster to take turns hitting each other until the monster is dead. Hooray! But then the player is going to go exploring some more, and guess what's in the next room in the dungeon? That's right, another monster. Hmm, how do we handle this?

With the tools we've currently got I can think of two ways of doing this. We've got a *Monster* object with zero health, and we have access to its health attribute, so we could just reset the health to 50:

Monster.health = 50

But what if we want the player to fight multiple monsters at once? Or if the player ran away from the first monster before it was dead, fought the second monster, then went back to fight the first monster? We'd be in a whole world of pain trying to keep track of how much health the *Monster* object should have at any given time.

The second option would be to write a separate object for each monster. Then each object will hold its own state, and we don't need to worry about tracking that ourselves. This might seem reasonable at first, but I'm intending to write a huge, open world game, with thousands of monsters to fight. I don't really want to have to write a thousand object files for these monsters.

Fortunately you're about to learn the magic trick that makes this really easy! Going back to our trusty milk carton analogy, we've said that a carton is an object. But that carton is getting made in a factory according to a set of instructions. Given those instructions and the factory, we can create as many carton objects as we like, each of which can have their own state (for instance holding different amounts of milk). Sounding similar to the problem we had with our monsters? Who would ever have thought that monsters and milk cartons had something in common?

Most OOP languages, including Scala, provide this factory mechanism for you to create new objects from instructions. And those instructions are called **classes**. Classes look very similar to objects. Let's see by rewriting our monster object as a class:

class Monster {

var health = 50

val damage = 5

def takeDamage(x: Int) = health -= x

}

Spot the difference? We've replaced the word *object* with the word *class*. Make sure you've replaced your monster object with the class, not just created a class as well. It's ok to have an object and a class called the same thing, but I won't be able to demonstrate my point if you've got both at the moment.

Look back in your *Game* object, and IntelliJ should be complaining about not being able to resolve the symbol *Monster*. Remember that a class is *instructions* for how to create an object, not an object itself. It wouldn't make sense to ask the instructions for creating a milk carton how much milk it currently holds, or to pour some milk. So in our program we can't ask the instructions for creating a monster to take some damage. We need to use the instructions to create a *Monster* object. And the way we ask Scala to create a new object for us is to use the word *new*. I'm going to rewrite the *Game* object to show you.

object Game extends App {

// Create a new monster and assign it to a value

val firstMonster = new Monster

// Create another new monster and assign it to a different value

val secondMonster = new Monster

// Fight!!!

firstMonster.takeDamage(Player.damage)

secondMonster.takeDamage(Player.damage)

firstMonster.takeDamage(Player.damage)

println(s"firstMonster has ${firstMonster.health} health remaining") // 30 health left

println(s"secondMonster has ${secondMonster.health} health remaining") // 40 health left

}

Brilliant! We're now able to create monsters willy-nilly, and they'll each maintain their own state of how much health they've got.

Setting attributes when newing up

The technical term for creating a new object from a class is *instantiating*. "I'm going to instantiate a new monster". When I worked at Sky my team there would have said, "I'm going to new up a monster" instead. It's kind of cute. Use whichever you like!

It's useful to be able to set attributes on objects when you create them. There might be a standard set of instructions for creating a milk carton, but the same factory should be able to print different "Best before" labels on it. For my monsters I'd like to have some variation in the amount of damage they deal, so as I create each one I'm going to specify its damage. To do that we need to remove the *val damage = 5* attribute from the class, and pass it in as a **constructor parameter** instead.

Constructor parameters are pretty similar to method parameters. They have names, types, and are found in parentheses after the class name. Here's how the *Monster* looks with a constructor parameter for its damage:

class Monster(damage: Int) {

var health = 50

// Methods omitted

}

And to create a monster object with a damage of 10, we would write:

new Monster(10)

Constructor parameters by default are *val*. In other words, once the object has been created the value can't be changed. We can create mutable constructor parameters as well, by specifying them as *var*. So to enable us to vary the starting health of the monsters as well, we'd remove the *var health = 50* attribute from the class and add a constructor parameter like so:

class Monster(damage: Int, var health: Int) {

// Methods omitted

}

And to create a monster object with a damage of 10 and a health of 20, we would write:

new Monster(10, 20)

So with our new constructor parameters, let's update our game:

object Game extends App {

// Create a weakling monster

val ghost = new Monster(2, 10)

// Create a powerful monster

val dreadLord = new Monster(100, 1000)

}

Named and default constructor parameters

Sometimes it makes sense to have default values for some of your constructor parameters, which means you don't have to specify them when you new up the object. For instance, I might decide that most of my monsters will have 5 damage and 10 health, and only a few special monsters will have different values. By specifying defaults, I'll be able to create most of my monsters by just typing *new Monster*, rather than having to type *new Monster(5, 10)* all the time.

You can specify defaults just by assigning a value to the parameter in the constructor. So the constructor for my new monster class would look like:

class Monster(damage: Int = 5, var health: Int = 10)

I can now create new monsters with both default parameters, and also override one or both of the values:

new Monster // Has damage 5, health 10

new Monster(20) // Only overriding the first parameter, so has damage 20, health 10

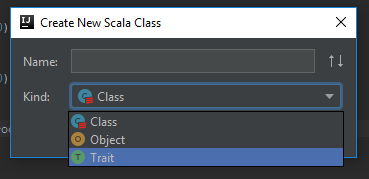
new Monster(20, 50) // Overriding both parameters, so has damage 20, health 50

If I just want to override the second parameter, how do I specify that the first should stay as the default? I can name the parameters when I'm calling them:

new Monster(health = 30) // Only overrides the health parameter, so has damage 5, health 30

Traits

We've learnt about objects, which have attributes and behaviour, and about classes, which allow us to create objects. The final part of the OOP triumvirate is **traits**. You might recognise the term. In fact, you've seen it several times already. When you create a new Scala file you have to choose the kind from the drop down list. The available options are: Class, Object and Trait.



Traits allow us to specify that classes (and hence objects) will have certain attributes and behaviours, but not necessarily have to say what those attributes and behaviours are. That probably sounds a little confusing, so let's have an example. I love examples!

I want to have another type of enemy for our player to fight - a wizard. Now because the wizard is magical it doesn't have any health, but it has magic instead. It can still be attacked, but the attacks will reduce its magic rather than its health. So it will look something like this:

class Wizard(damage: Int = 10, var magic: Int = 50) {

def takeDamage(x: Int) = magic -= x

}

And just for reference, our monster class looks like:

class Monster(damage: Int = 5, var health: Int = 10) {

def takeDamage(x: Int) = health -= x

}

And just for the fun of it, let's create something else that we can damage:

class WoodenCrate(isBreakable: Boolean = false) {

def takeDamage(x: Int) = {

if (isBreakable) println("Bam! I've been smashed!")

else println("Hah! Your puny blows mean nothing to me!")

}

}

Don't worry about the implementation of the *WoodenCrate*. There's some stuff in there we'll learn about shortly. The thing to notice is that all three classes have a *takeDamage(x: Int)* method, but the method bodies do different things.

One other change I'd like to make is that the *Game* object currently has to know about how the *Player* attacks the *Monster*. I think the *Player* should know how to do that itself. So I want the *Game* to look like:

object Game extends App {

val ghost = new Monster

Player.attack(ghost)

}

*Player* doesn't have an *attack* method yet, so we'll have to write one. Try and do this yourself, and write a test in the *PlayerSpec* class first. I'll give you some hints below, and then my solution below that.

Hints

- The *attack* method will have a method parameter of type *Monster*.

- You'll need to call the monster's *takeDamage* method from the body of the *attack* method.

- You can reference the player's *damage* attribute from the method body just by calling it by name.

- To test that it's working create a *Monster* with a known amount of health, attack it, then check that its health has been reduced by the *Player*'s attack value.

My implementation

New test in PlayerSpec:

"Attacking a monster" should "reduce the monster's health by the value of the player's damage" in {

val x = new Monster(health = 50)

Player.attack(x)

x.health shouldBe 40

}

New method in Player:

object Player {

var health = 100

val damage = 10

def attack(monster: Monster) = monster.takeDamage(damage)

// Omitted other methods

}

Hopefully that makes sense. You're passing an object of type *Monster* into the *Player*'s *attack* method. In the method body you're calling the *takeDamage* method on the *Monster* object, and passing in the value of the *Player*'s *damage* attribute. The body of the *Monster*'s *takeDamage* method subtracts the value passed into it from the *Monster*'s *health* attribute.

Back to the game

Great, now that the player can attack a monster, I'd like them to be able to attack Wizards and WoodenCrates as well. I can't just write:

val wiz = new Wizard

Player.attack(wiz)

Give it a try. It's not working because we've written the *attack* method to expect a method parameter of type *Monster*, and we're trying to pass it an object of type *Wizard*. You know about method overloading, so one solution would be to write separate *attack* methods for Wizards and WoodenCrates. That would look like this:

object Player {

var health = 100

val damage = 10

def attack(x: Monster) = x.takeDamage(damage)

def attack(x: Wizard) = x.takeDamage(damage)

def attack(x: WoodenCrate) = x.takeDamage(damage)

// Omitted other methods

}

There's some very similar looking code here. I've renamed the method parameters to *x*, just to highlight how similar the three methods are. In fact, their method bodies are identical. This is manageable when we've got three attackable object types, but not so great if we've got a dozen, or a hundred. Here's where traits come in. As I've said,

*"Traits allow us to specify that classes (and hence objects) will have certain attributes and behaviours, but not necessarily have to say what those attributes and behaviours are."*

We're going to use a trait to specify that our three classes all have the *takeDamage* behaviour, but let the classes specify what those behaviours are themselves. This is really easy to do. Right click on the *src/main/scala* directory and select **New -> Scala class**. In the pop up box, select **Trait** as the kind, and name it *Attackable*. Add the following to the trait body, so the file looks like this:

trait Attackable {

def takeDamage(x: Int)

}

Here we've just got the method definition, and not the method body. In other words we've described a behaviour, but not how that behaviour should work. There are two things we need to update to make this work: the *Player* object to use the trait type instead of the object types, and the attackable objects so the computer can associate them with the trait. I'll update the player first. You can remove the three *attack* methods, and replace them with this:

def attack(x: Attackable) = x.takeDamage(damage)

Here we're saying that the method should take a method parameter of type *Attackable*. And we know that *Attackable* types will have a *takeDamage* method, so it's safe to call *takeDamage* on the parameter in the method body.

Now let's update our three classes so that the computer knows they are *Attackable* types. We do this by **extending the trait**, which just means writing the word *extends* followed by the trait name at the end of the class definition, like so:

class Monster(damage: Int = 5, var health: Int = 10) extends Attackable {

// class body

}

class Wizard(damage: Int = 10, var magic: Int = 50) extends Attackable {

// class body

}

class WoodenCrate(isBreakable: Boolean = false) extends Attackable {

// class body

}

A common source of confusion for people learning this is knowing when they can access methods and attributes that are specific to the class, and when they are restricted to accessing those defined on the trait. Basically, if you create an object from a class that extends a trait, you can treat it as a plain old object the way we've done up to now. You can access all the attributes and methods on it. So the following is all fine:

val m = new Monster

println(m.health)

println(m.damage)

m.takeDamage(20)

However, as soon as you pass it to something that expects the trait, you lose access to any attributes and methods that aren't defined in the trait.

object Player {

def attack(x: Attackable) = {

println(x.health) // Not ok!

println(x.damage) // Not ok!

}

}

If you think about it, it makes sense. We're allowed to pass both Monsters and Wizards to the *attack* method. Monsters have a health attribute, but not a magic attribute. Wizards have a magic attribute, but not a health attribute. If we tried to access the health attribute when we've got a Wizard, the program wouldn't know what to do and would crash. You can think of a trait as a contract guaranteeing that any object extending it will have certain attributes available, or have an implementation of certain methods.

Default methods

Similar to how we learned to provide default values for constructor parameters, we can also provide default method bodies in traits. If we provide a default implementation then classes can extend the trait without providing their own implementation, and the default will be used. If they decide to implement the method themselves then the default will be ignored.

We might want some objects in our game to be attackable, but for nothing much to happen if they are attacked. For instance, the player might attack a wall, or a rock. Let's say if any of these type of objects are attacked, it will just print out, "Well, that didn't do much". We'll decide that this is the default behaviour for attackable objects, unless they specify something else should happen. We also don't want to have to write *def takeDamage(x: Int) = println("Well, that didn't do much")* in every one of these classes. So this is a great chance to use a default method in a trait. I'm going to write my *Rock* class first:

class Rock extends Attackable

See that IntelliJ is underlining the name of the class in red. Something's not right. If you cover your mouse pointer over the class name, you should see a tooltip with a message like, *"Class 'Rock' must either be declared abstract or implement abstract member 'takeDamage(x: Int): Unit' in 'Attackable'"*. This is basically saying that the class *Rock* is breaking the *Attackable* contract by not specifying the behaviour for *takeDamage*. If we were to create a new *Rock* object and pass it into the *Player*'s *attack* method, the computer would try to call *takeDamage* on it and wouldn't find any method to run. We're going to fix this by providing the method body in the trait.

trait Attackable {

def takeDamage(x: Int) = println("Well, that didn't do much")

}

Now look back at your *Rock* class, and everything should be fine.

We're almost there, but take a look at your *Monster* class. Oh dear, we've now got some problem with our *takeDamage* method. Hover over the method and you should get a tooltip explaining the problem: *"Method 'takeDamage' needs override modifier"*. Because you've got a default implementation in the trait, Scala wants you to explicitly say that you want the method in the class to override the method in the trait. It's very easy to fix. Just put the word **override** before the word **def**.

override def takeDamage(x: Int) = health -= x

You'll have to do this for the Wizard and the WoodenCrate classes as well. Now you can add the following lines to your Game object:

val wall = new Wall

Player.attack(wall) // prints out "Well, that didn't do much"

A recap

Let's check we're still kind of in sync with each other. I've got the following files:

Attackable.scala

trait Attackable {

def takeDamage(x: Int) = println("Well, that didn't do much") // Default implementation

}

Monster.scala

class Monster(damage: Int = 5, var health: Int = 10) extends Attackable {

override def takeDamage(x: Int) = health -= x

}

Wizard.scala

class Wizard(damage: Int = 10, var magic: Int = 80) extends Attackable {

override def takeDamage(x: Int) = magic -= x

}

WoodenCrate.scala

class WoodenCrate(isBreakable: Boolean = false) extends Attackable {

override def takeDamage(x: Int) = {

if (isBreakable) println("Bam! I've been smashed!")

else println("Hah! Your puny blows mean nothing to me!")

}

}

Wall.scala

class Wall extends Attackable

Player.scala

object Player {

val damage = 10

var health = 100

def attack(enemy: Attackable) = enemy.takeDamage(damage)

def takeDamage(x: Int) = health -= x

}

Game.scala

object Game extends App {

val ghost = new Monster(health = 5, damage = 5)

val beast = new Monster(health = 50, damage = 10)

val wiz = new Wizard

val crate = new WoodenCrate

val wall = new Wall

Player.attack(ghost)

println(s"Ghost now has ${ghost.health} health") // Should print -5 health

Player.attack(beast)

println(s"Beast now has ${beast.health} health") // Should print 40 health

Player.attack(wiz)

println(s"Wiz now has ${wiz.magic} magic") // Should print 70 magic

Player.attack(crate) // Should print "Hah! Your puny blows mean nothing to me!"

Player.attack(wall) // Should print "Well, that didn't do much"

}

object Game extends App

We first wrote this object definition back when we created our first "Hello World" program, and I promised I'd explain it later. Well, the time is now! Go to your *Game.scala* file, and **Ctrl + Click** or **Cmd + Click** on the word *App*. IntelliJ should take you to a new file where *App* is defined, and surprise surprise, it's a trait! Have a look through the file, but don't worry if you don't understand everything. The main thing I want you to notice is a default method defined in the *App* trait:

def main(args: Array[String]) = {

// Some code in the method body

}

When I told you previously that the computer looks for an object extending *App* to work out where to start running your program, I wasn't quite telling you the whole truth. What it actually looks for is a method called *main* taking one method parameter of type *Array[String]*. So because our *Game* object is extending a trait with the *main* method, the computer can find it and know where to start.

You don't have to create an object extending *App* to start your program if you don't want to. You can create a class and write a *main* method yourself. If I wanted to change my *Game* object to a class, I'd write it like this:

class Game {

def main(args: Array[String]) = {

val ghost = new Monster(health = 5, damage = 5)

val beast = new Monster(health = 50, damage = 10)

// etc

Player.attack(ghost)

println(s"Ghost now has ${ghost.health} health") // Should print -5 health

Player.attack(beast)

println(s"Beast now has ${beast.health} health") // Should print 40 health

// etc

}

}

It's really a matter of personal preference which way you want to start your program.

Dependency Injection, Stubs and Mocks

When you start to write programs of any complexity, it's likely that you'll start having dependencies on objects that other people have written. In fact, we're already depending on the *Console* object to enable us to print things out using its *def println(x: Any)* method.

Writing good code takes skill, and the question of "what is good code" has different answers to different people. However, I'd say that most professional programmers would agree that testing the core business logic of the code you're writing is important, as is writing code in a way that is maintainable and not too difficult to change in the future.

You can't really test absolutely *everything* though, and we generally have a certain amount of trust in the dependencies we use to do what they're supposed to. We also want to be able to swap out dependencies when our business needs change. An example of this might be initially using an object to save things to the file system, then deciding to swap this out for an object that saves things to a database. This chapter is about the art of structuring your code so that you can test the important bits, not have to test certain dependencies, and making it easy to swap dependencies.

What the Bool?

The examples that follow need you to understand what **Booleans** are. So now's as good a time as any to cover them. A *Boolean* is one of the types built into Scala, in much the same way that *String* and *Int* are types. Whereas an *Int* can represent any whole number, and a *String* can represent any piece of text, a *Boolean* can only be one of two values: **true** or **false**. Here are some examples of plain English statements that are either true or false:

- One plus one equals two - true

- Apple starts with the letter 'A' - true

- 10 is less than 5 - false

- Leicester City will win the Premier League - false (hmm)

Where *Boolean*s really shine are when we use them to change how our programs behave based on whether they are true or false. So far our programs have had a single path of execution. They start, then they run sequentially through the instructions we've given, then they finish. *Boolean*s let us introduce branch points into our programs, using something called *conditional expressions*. This basically tells the computer to "do this thing if this condition is true, otherwise do this other thing". *Conditional expression* is a bit of a mouthful, so most people just call them *if statements*.

The syntax of an if statement is the word *if*, followed by some expressions that the computer can evaluate to either true or false in parentheses, followed by the code to execute if the expression evaluates to true. It's probably easier to see this by example:

object Program extends App {

val x = 20

if (x > 10) println(s"$x is greater than 10")

}

Here the computer evaluates the expression *x > 10* by looking at the value of *x* and seeing whether it is greater than 10. So the expression the computer sees is, "20 is greater than 10". This is true, so it runs the code printing the message to the terminal. Try changing the value of *x* to 5 and running the program again. Because the expression, "5 is greater than 10" is false, the code isn't run.

We've got the code to say, "do this thing if this expression is true", but we also need the bit to say "otherwise do this other thing". The way we do that is by using the keyword *else*. I'll extend the example to demonstrate.

object Program extends App {

val x = 20

if (x > 10) println(s"$x is greater than 10")

else println(s"$x is less than 10)

}

Hopefully that's pretty straightforward. You can see how this would be useful in something like a game:

if (attack button is pressed) use sword

else raise shield

Sometimes you need to use curly braces around the bits of code to execute, and sometimes you don't. The rule is the same as for method bodies. If the body is going to take up more than one line, you need to wrap it in curly braces. You can still use curly braces if the body is only one line, but they're optional. I'll rewrite the example to show you:

object Program extends App {

val x = 20

if (x > 10) {

println(s"$x is greater than 10")

println("And here's another line, so we have to be wrapped in braces")

}

else {

println(s"$x is less than 10)

}

}

You can also extend your conditional expressions to have more than a single true and false branch. Use *else if* to add more branches:

object Program extends App {

val x = 20

if (x > 10) {

println(s"$x is greater than 10 but less than 15")

}

else if (x > 15) {

println(s"$x is greater than 15")

}

else {

println(s"$x is less than 10)

}

}

Hmm, that doesn't seem quite right. Hopefully you've been following along, and if you ran that program it would have printed, "20 is greater than 10 but less than 15". That's not what we wanted. It should have said, "20 is greater than 15". What's happening is that the statements are being evaluated in order, and as soon as one of them evaluates to true the rest of them will be ignored. 20 is indeed greater than 10, so the first statement *x > 10* evaluates to true.

We can fix this by swapping the two statements around, but this shows that you sometimes need to pay attention to the order in which things are written.

object Program extends App {

val x = 20

if (x > 15) {

println(s"$x is greater than 15")

}

else if (x > 10) {

println(s"$x is greater than 10 but less than 15")

}

else {

println(s"$x is less than 10)

}

}

When it comes to the types of statement we can use in conditionals, anything that evaluates to a *Boolean* is fine, but we commonly write statements that compare one thing to another. For instance, we've been comparing two numbers, and evaluating whether one is greater than the other. The types of comparisons we can do are as follows:

x == y // True if x is equal to y

x > y // True if x is greater than y

x >= y // True if x is greater than or equal to y

x < y // True if x is less than y

x <= y // True if x is less than or equal to y

x != y // True if x does not equal y

That double equals (==) catches a lot of people out. Why do we have to write two *=*? Well we know that a single *=* is already used for assignment, so we need a different way of expressing that we want to do a comparison. If you're getting errors in your comparisons, make sure you're not doing an assignment instead!

Have a bit of a play around using if statements with each of these types of comparison. When you're ready, move on to the next part.

The next part

Right, after that diversion into the world of true and false we're ready to tackle dependency injection, stubs and mocks. Here's a problem:

class MilkCarton(var amountOfMilk: Int) {

def pourMilk(x: Int) = {

// Oooh, an if statement!

if (x > amountOfMilk) println("Error, can't pour that much milk")

else amountOfMilk -= x

}

}

object Program extends App {

val carton = new MilkCarton(amountOfMilk = 10)

carton.pourMilk(20) // Should print "Error, can't pour that much milk"

}

How can we test this logic in the *pourMilk* method? It's actually pretty difficult to test that something is being printed to the terminal. And I'm not sure that printing to the terminal is actually the important logic we want to test. We actually want to make sure that some kind of alert is being generated if we try to pour too much milk. For now the alert is represented as a message being printed. But in the spirit of making things easy to change, it could be that we'd want the alert to be represented by an email being sent to someone, or something being logged in a database somewhere. We'd like to be able to change this end functionality without having to change the test of the core alerting logic. Let me express this in a test:

import org.scalatest.\_

class MilkCartonSpec extends FlatSpec with Matchers {

"Trying to pour more milk than is available" should "result in an alert being triggered" in {

val carton = new MilkCarton(10)

carton.pourMilk(20)

// How do we test the alert has been triggered here?

}

}

I'm going to show you two ways of testing this, both involving a similar trick with slight of hand. Firstly I'm going to wrap the *println* call inside a method in another class, which I'll call *ConsolePrinter*. It does what it says on the tin!

class ConsolePrinter {

def alert() = println("Error, can't pour that much milk")

}

Now we'll change our *MilkCarton* class to use our new *ConsolePrinter*. We're going to go through several steps to get to the final result, in order to demonstrate how things work, but once you've got the hang of it you can jump straight to using the whole pattern.

class MilkCarton(var amountOfMilk: Int) {

val alerter = new ConsolePrinter

def pourMilk(x: Int) = {

if (x > amountOfMilk) alerter.alert()

else amountOfMilk -= x

}

}

You're probably thinking, "how does that help?" It doesn't seem to make it any easier to test, and it's just made our code more complicated. Well, it's time for our next step. What I'm really interested in testing is that the *ConsolePrinter*'s *alert* method has been called at the appropriate time. I don't really care what the *alert* method is doing. It could be printing to the console, or having a sword fight with another method. Whatever!

So here's the first part of the trick. We're going to add an attribute to the *ConsolePrinter* that will tell us whether the alert method has been called or not.

class ConsolePrinter {

var alertHasBeenCalled = false

def alert() = {

println("Error, can't pour that much milk")

alertHasBeenCalled = true

}

}

You might see where this is going. If you're feeling up to it, stop reading now and have a go at updating the test.

Go on, have a go.

No peeking!

Ok? Well this is how I would do it.

"Trying to pour more milk than is available" should "result in an alert being triggered" in {

val carton = new MilkCarton(10)

// Optionally make sure that the alert hasn't been called to start with

carton.alerter.alertHasBeenCalled shouldBe false

carton.pourMilk(20)

carton.alerter.alertHasBeenCalled shouldBe true

}

Great! Got a passing test? That must be it then!

Well, not quite. We've still got some work to do. It's generally considered bad practice to add extraneous code to your classes just to be able to test them. We want lean and focused classes. So, using this *alertHasBeenCalled* variable seems to be a good idea, but we don't want it in our *ConsolePrinter*. Here's the next part of the trick. We're going to create a **testing stub** to use instead of the *ConsolePrinter*, just for use in our tests. It could look like this:

class ConsolePrinterStub {

var alertHasBeenCalled = false

def alert() = alertHasBeenCalled = true

}

and our *ConsolePrinter* will go back to

class ConsolePrinter {

def alert() = println("Error, can't pour that much milk")

}

We don't care about testing the method implementation, so the stub ignores the *println*, and just sets the *alertHasBeenCalled* variable to *true*. Now the final part of the trick is to use the stub in the test instead of the actual *ConsolePrinter*.

It probably doesn't leap out at you how to do this. I mean, we're creating an instance of the *ConsolePrinter* directly inside our *MilkCarton* class:

class MilkCarton(var amountOfMilk: Int) {

val alerter = new ConsolePrinter

// Omitted code

}

How can we get the *alerter* to be a *ConsolePrinterStub* instead? In fact, we've already learnt the tools we need to do this. We're going to use a combination of **trait** and **constructor parameters** to decide whether we want a *ConsolePrinter* or a *ConsolePrinterStub* at the point we create the *MilkCarton* object.

First let's create a trait representing objects that have an *alert* method.

trait Alerter {

def alert()

}

Now make both our actual printer and the stub extend this trait:

class ConsolePrinter extends Alerter {

// Omitted code

}

class ConsolePrinterStub extends Alerter {

// Omitted code

}

Add a constructor parameter of type *Alerter* to the *MilkCarton* class:

class MilkCarton(var amountOfMilk: Int, alerter: Alerter) {

def pourMilk(x: Int) = {

if (x > amountOfMilk) alerter.alert()

else amountOfMilk -= x

}

}

In our *Program* object, pass a *ConsolePrinter* into the *MilkCarton* when it's newed up:

object Program extends App {

val carton = new MilkCarton(amountOfMilk = 10, alerter = new ConsolePrinter)

carton.pourMilk(20) // Should print out "Error, can't pour that much milk

}

And pass the stub into the *MilkCarton* in the test:

"Trying to pour more milk than is available" should "result in an alert being triggered" in {

// We need to have a variable to keep reference to the stub, so we can refer to it later

val alerter = new ConsolePrinterStub

val carton = new MilkCarton(amountOfMilk = 10, alerter)

// Optionally make sure that the alert hasn't been called to start with

alerter.alertHasBeenCalled shouldBe false

carton.pourMilk(20)

alerter.alertHasBeenCalled shouldBe true

}

Hooray, everything's wired up. Take your time to look through that again, and make sure you understand how it works. This pattern of passing in, or injecting, dependencies into classes is called **dependency injection**, and is a really useful way to make sure that your code is well structured, well tested, and easy to change.

In terms of being easy to change, let's say I want to use a different dependency to do the alerting, perhaps to send an email instead of printing to the console. As long as that class has an *alert* method and extends the *Alerter* trait, all we have to do is pass in a new instance of this class to the *MilkCarton* instead, and we don't have to change any of the code in our class.

class EmailSender extends Alerter {

def alert() = // Some code to send an email

}

object Program extends App {

val carton = new MilkCarton(amountOfMilk = 10, alerter = new EmailSender)

carton.pourMilk(20) // Should send an email

}

Nice!

A common real world use case for dependency injection is using a dependency to save data. When you start writing an application, you want to get something working as quickly as possible to get early feedback on it. So you might decide it's quickest just to save things to memory. But then you start getting real users, and just saving things to memory isn't good enough, as everything will be lost when the application is restarted. So you decide to switch out your dependency that saves to memory with a dependency that saves to the file system. Then your application starts getting really successful, and you want to do analytics on the data that's been saved. Storing your data in files isn't ideal for doing analytics, so you swap out your file system saver dependency with a database saver dependency. And all this time you don't have to change the core code of your application. You're just injecting different dependencies.

Mocking dependencies

We've seen how to substitute our dependencies with stubs for testing purposes. I said earlier that I'd show you two ways of testing the alerting functionality in the *MilkCarton* class. The alternative to creating stubs is to use a **mocking framework**. This is quite similar, in that it relies on using dependency injection, but you don't have to create special versions of your dependencies just for testing. We're going to rework the *MilkCarton* example to illustrate using mocks.

We need to install the Mockito framework in our project in order to be able to use mocks. Add the following line at the bottom of your `build.sbt` file:

libraryDependencies += "org.mockito" % "mockito-core" % "2.15.0"

IntelliJ should now download Mockito for you. If it doesn't seem to recognise Mockito as you follow along with the example code, you might not have turned on auto imports. If so, go to the **terminal** tab at the bottom of IntelliJ and run `*sbt update*`.

We're going to go back to the *MilkCartonSpec* file and add another test. We need to make Mockito available within our file, which means importing it into the file. Add the imports at the top of the file, like so:

import org.scalatest.\_

import org.mockito.Mockito.\_

import org.scalatest.mockito.MockitoSugar

We're also going to have our *MilkCartonSpec* class extend another trait, which will allow us to use the Mockito syntax in our tests:

class MilkCartonSpec extends FlatSpec with Matchers with MockitoSugar {

}

So for reference, our test file should now look like this:

import org.scalatest.\_

import org.mockito.Mockito.\_

import org.scalatest.mockito.MockitoSugar

class MilkCartonSpec extends FlatSpec with Matchers with MockitoSugar {

"Trying to pour more milk than is available" should "result in an alert being triggered" in {

// We need to have a variable to keep reference to the stub, so we can refer to it later

val alerter = new ConsolePrinterStub

val carton = new MilkCarton(amountOfMilk = 10, alerter)

// Optionally make sure that the alert hasn't been called to start with

alerter.alertHasBeenCalled shouldBe false

carton.pourMilk(20)

alerter.alertHasBeenCalled shouldBe true

}

}

Add another test below. It's going to be testing the same thing as the existing test, but two tests can't share the same name within a class, so we'll have to call it something different. I'm going to call mine,

"Using a mock" should "work" in {

// To do

}

Don't worry too much about remembering the syntax for mocking. This is just to show you the principals behind it. A quick Internet search for "Scalatest mocks" will point you to lots more information if you want to dig deeper. The syntax for creating a mock is the word **mock** followed by the name of the class or trait you want to mock inside square brackets. In our example, we want to create a mock of the *Alerter* trait, so the syntax is *mock[Alerter]*.

We need to assign the mock to a value, so that we can refer to it later. Let's start by adding the mock to our test:

"Using a mock" should "work" in {

val mockAlerter = mock[Alerter]

// To do

}

By the way, don't give your *val* the name “mock”, as it won't work, and it won't be obvious why it isn't working. An easy way to consistently name your mocks is to prepend the word “mock” to the name of the thing you're mocking.

Now we've got an object that is a mock of the *Alerter*. We can use this just like any object that extends *Alerter*. For instance, we can pass it into methods or constructors that expect an *Alerter* parameter, and we can call any of the methods defined in the *Alerter* trait on it. Let's try calling *alert()* on our mock:

"Using a mock" should "work" in {

val mockAlerter = mock[Alerter]

mockAlerter.alert()

// To do

}

You might think this would cause a problem, as there's no method body defined for our *alert* method. But the mocking framework is kind of automatically creating a stub for you. You can imagine that behind the scenes the mock is holding some state about whether the method has been called or not, and when you call the method it updates the state. The way to verify whether a method has been called on a mock is to pass the mock into Mockito's *verify* method, then call the method you want to verify. It's easier to demonstrate this than to describe it:

"Using a mock" should "work" in {

val mockAlerter = mock[Alerter]

mockAlerter.alert()

verify(mockAlerter).alert()

}

So what this test is doing is creating a mock of the *Alerter*, calling the *alert* method on the mock, then verifying that the *alert* method was called. If you swap the last two lines the test should fail, because you'd be trying to verify that the method has been called before it was actually called.

We're almost there. Let's just add in our *MilkCarton* to the test:

"Using a mock" should "work" in {

val mockAlerter = mock[Alerter]

val carton = new MilkCarton(amountOfMilk = 10, alerter = mockAlerter)

carton.pourMilk(20)

verify(mockAlerter).alert()

}

Mockito has lots of cool features to help you simulate your dependencies and test how they are being used. When your dependency has got complex behaviour you might have to end up writing an even more complicated stub to test it, whereas mocking can be a lot more flexible. A couple of features you might find interesting are counting the number of times a method has been called on a mock,

"Using a mock" should "work" in {

val mockAlerter = mock[Alerter]

val carton = new MilkCarton(amountOfMilk = 10, alerter = mockAlerter)

verify(mockAlerter, times(0)).alert // Make sure alert hasn't been called yet

carton.pourMilk(20)

verify(mockAlerter, times(1)).alert() // Alert should have been called once

carton.pourMilk(20)

verify(mockAlerter, times(2)).alert() // Alert should have been called twice

}

and being able to specify the behaviour you expect from your mock. As a completely contrived example, let's say we want to test a method that takes a random number generator dependency. If the dependency generates a number below 10, then our method returns the random number times 2. If it generates a number over 10, then it returns the number divided by 2. So our random number generator trait would look like this:

trait RandomNumberGenerator {

def getNumber(): Int

}

And our method will look like this:

object Contrived {

def doSomeFunkyMaths(rng: RandomNumberGenerator) = {

val randomNumber = rng.getNumber()

if (randomNumber < 10) randomNumber \* 2

else randomNumber / 2

}

}

Now I want to write the following tests:

import org.scalatest.\_

import org.mockito.Mockito.\_

import org.scalatest.mockito.MockitoSugar

class ContrivedSpec extends FlatSpec with Matchers with MockitoSugar {

"DoSomeFunkyMaths" should "return double a random number that is less than 10" in {

// To do

}

"DoSomeFunkyMaths" should "return half a random number that is more than 10" in {

// To do

}

}

We can't use a real random number generator, as we have no way of finding the value it is giving to the method, so we can't test whether our method is doubling it or halving it correctly. We need to test with a dependency where we can control the number that is being generated. Mockito gives us a nice way of doing this, by specifying what behaviour we want a method to have.

"DoSomeFunkyMaths" should "return double a random number that is less than 10" in {

val mockRNG = mock[RandomNumberGenerator]

// Specify that the mock RNG should return 2 when the getNumber method is called

when(mockRNG.getNumber).thenReturn(2)

Contrived.doSomeFunkyMaths(mockRNG) shouldBe 4

}

"DoSomeFunkyMaths" should "return half a random number that is more than 10" in {

val mockRNG = mock[RandomNumberGenerator]

// Specify that the mock RNG should return 40 when the getNumber method is called

when(mockRNG.getNumber).thenReturn(40)

Contrived.doSomeFunkyMaths(mockRNG) shouldBe 20

}

Summary

Well done! Those were a tough couple of chapters. I know some of these concepts can be a bit mind bending, but if you can basically understand the idea of what's going on here, all it needs is practice for it to become second nature. Testing with stubs and mocks could fill up its own book, so we've really only scratched the surface here, but you've now got a good grounding in the fundamentals. Many people who write code for a living don't know these techniques for writing clear and well tested code, so if you're following along give yourself a big pat on the back!

Lists and Arrays

In this chapter we're going to have our first look at a *data structure*. Data structures allow us to... well... structure related bits of data in useful ways. There are loads of common data structures, including: arrays; singly-linked lists; doubly-linked lists; sets; maps; queues; stacks; heaps; graphs etc etc. They structure collections of data in different ways, for different use cases.

For instance, a **queue** is a *first-in-first-out* data structure. You can think of it like a pipe into which you can push a series of objects. Then you can get objects back out of the pipe in the order in which you put them in. Some of these examples I'm going to write in pseudocode, which just means it looks like code, but won't actually run. It's used in books like this to illustrate concepts without getting bogged down in the details. In pseudocode, a queue would look something like this:

queue.push("a")

queue.push("b")

queue.push("c")

// queue now contains ("a", "b", "c")

val first = queue.getElement()

// first = "a", queue now contains ("b", "c")

val second = queue.getElement()

// second = "b", queue now contains ("c")

val third = queue.getElement()

// third = "c", queue is now empty

Whereas a **stack** is a *last-in-first-out* data structure. You can think of it like a stack you might make out of books. You can put new books on the top of the stack, and when you take a book from the stack you have to take it from the top. As a comparison with the queue, it would look like this:

stack.push("a")

stack.push("b")

stack.push("c")

// stack now contains ("a", "b", "c")

val first = stack.getElement()

// first = "c", stack now contains ("a", "b")

val second = stack.getElement()

// second = "b", stack now contains ("a")

val third = stack.getElement()

// third = "a", stack is now empty

To really get a good feel for data structures, it's useful to have an understanding of how programs actually store and access data. I'm going to use an analogy, in which we're going to pretend that our program is running inside a telephone exchange, and there are a hundred houses representing our computer memory. The houses each have a telephone number, number 1 being the first house, and number 100 being the last. Our program can ask the exchange to call up any house and ask it to either store some data or tell the program what data is currently stored in it.

*Disclaimer: computers don't contain tiny houses and telephone exchanges. This is deliberately simplified!*

What data can these houses store? They could be simple bits of data like strings and integers, or they could be more complex objects like milk cartons and monsters.

We can now see what's actually happening when we assign data to a variable. Let's step through what happens in this assignment:

val x = "Hello World!"

The program asks the telephone exchange for a house that's not currently being used. The exchange has a little book where it keeps track of which houses have data, and which don't. Let's say House 5 is currently empty. So the exchange tells the program, "you can use House 5".

"Great!", says the program. "Please could you phone up House 5 and tell it to store the string 'Hello World'".

"No problemo!", says the exchange. And so house 5 ends up storing our string.

The program has its own little book, where it keeps track of which houses are storing the values for each of its variables. So it updates its book to record that the value for variable `x` is stored in house 5.

Now the program comes across the following line of code:

println(x)

"Hmm", it says, scratching its chin. "Now where did I put the value for variable `x`?" It looks it up in the book, and sees that it's in House 5. So it asks the exchange to get the value from House 5.

"Yo!", says the exchange to House 5. "What data you got?"

"I'm holding onto the string 'Hello World', like you told me to", says House 5

"Coolio, I'll just pass that along to the program", says the exchange.

"Thanks guys," says the program. Now I can print out "Hello World!"

And there you go. That's just how computer memory works!

Just to prove that I'm not making this up, let's write a little program. We'll need a class. It doesn't really matter what it is. You can use a `MilkCarton`, a `Monster` or write a new one. It doesn't need to have any attributes or methods. We're just going to use it to create some object instances of it. Assuming you've got something like:

class Monster {

}

create the main part of your program like so:

object Program extends App {

println(new Monster)

println(new Monster)

}

Run the program, and you should see something like the following in the terminal, although the random letters and numbers at the end will be different for you:

Monster@7dc7cbad

Monster@d2cc05a

Looks a bit strange doesn't it? Those weird letters and numbers are actually the phone numbers of the houses where the objects are stored. Or more accurately, they are the memory addresses of the memory registers where the objects are stored. Because we've created two different `Monster` objects, they have to be stored in two different locations in memory.

If you're interested, those addresses are printed in *hexadecimal*. How computers work in binary, and different binary notations, are really very interesting, but not core to this book. There is also plenty of great material for free online about this subject, so if you're so inclined I'd encourage you to do some reading about binary and hexadecimal at some point.

We've seen that the *println* method definition in the *Predef* object is:

def println(x: Any): Unit

So you can pass anything into it to be printed to the terminal. But what does it mean to print out a *Monster* object, or a *MilkCarton*? The writers of the *println* method decided that the most sensible thing was to have a default implementation that just prints out the type of the object, followed by an *@,* then the object's memory address.

This isn't really very useful, and not what you'd normally want, so they've provide a way to override this default. When *println* receives an object, it first looks to see whether that object has a method called *toString* defined on it. The method has to return a string, and that is what will be printed out. If the method isn't present, then the default is used. So we could get our *Monster* to print out something more useful like so:

class Monster {

override def toString() = "I'm a Monster!"

}

Run the program again, and you'll get the following output:

I'm a Monster!

I'm a Monster!

And to make it even more useful, let's change it so we can distinguish between the two objects:

class Monster(name: String) {

override def toString() = s"I'm a Monster called $name!"

}

Change your program to pass in different names to your *Monster*s in the constructor parameters, and run the program again.

That little exercise was an illustration of memory addresses, but now let's get back to data structures. We now know enough to be able to talk about two of the most basic, but also most commonly used data structures: arrays and lists.

Arrays

An array is a sequence of memory addresses that are next to each other. Using our houses analogy, if we wanted an array that can hold 5 objects of pieces of data we'd ask the exchange to find a block of 5 empty houses with sequential numbers. It might find that houses 11, 12, 13, 14 and 15 are empty, so it sets them aside for our array, and it returns the number of the first house in the array: 11.

Now the program just needs to store a single variable referencing the start of the array, rather than having to have a variable for every element in the array. You can see how this would be useful when you start having hundreds or thousands of elements in the array. It can then very quickly access any element of the array by asking the exchange to do some simple maths.

Say it wants the first element, it will ask the exchange for the number of the first element (11). If it wants the second element, it will ask the exchange to add 1 to the number of the first element (11 + 1 = 12). If it wants the fifth element, it will ask the exchange to add 4 to the number of the first element (11 + 4 = 15).

The number it asks to add on to the number of the first element is called the **index** of the array. Arrays are zero-indexed data structures, because to get the first element of the array you ask to add 0 to the number of the first element.

Let's do an example, modelling a shopping list with an array. The syntax for creating a new array is *new Array* followed by the data type that the array will contain within square brackets, followed by the size of the array within parentheses. I know I want to buy 5 items, so I'm going to have an array with 5 elements, each holding a string describing the items:

val shoppingList = new Array[String](5)

Now I want to add a string to each element of the array. We reference individual elements in the array by specifying the index of the element in parentheses after the name of the array. So we can assign a string to the first element in the array (i.e. at index 0) like this:

shoppingList(0) = "Apples"

And we can retrieve the value of the element like this:

val x = shoppingList(0)

println(x) // Prints "Apples"

// Or without having to use an extra variable to hold the data

println(shoppingList(0)) // Prints "Apples"

Let's write a little program to fill out all five elements of our shopping list, then print out the middle element:

object Program extends App {

val shoppingList = new Array[String](5)

shoppingList(0) = "Apples"

shoppingList(1) = "Toothpaste"

shoppingList(2) = "Potatoes"

shoppingList(3) = "Newspaper"

shoppingList(4) = "Bread"

println(shoppingList(2)) // Prints "Potatoes"

}

A useful feature of arrays is that it's easy to change the data in specific elements. Let's say we don't want potatoes after all, but want to swap them for Easter eggs. We can just assign a new string to the third element of the array:

object Program extends App {

// You can use this syntax for creating an Array and populating its elements at the same time

val shoppingList = Array[String]("Apples", "Toothpaste", "Potatoes", "Newspaper", "Bread")

println(shoppingList(2)) // Prints "Potatoes"

shoppingList(2) = "Easter eggs"

println(shoppingList(2)) // Prints "Easter eggs"

}

Adding, retrieving and changing data in arrays is really quick. Because the elements are all in a continuous block of memory, given the address of the first element in the array and the index of the element it's really simple for the program to add them together to find out the address of the element, and then can access that element directly. Our example of setting the third element to "Easter eggs" would involve the following steps:

> 1. Get address of the first element in the shoppingList array (say address 25)

> 2. Add index of 2 to this address to get the address of the third element (27)

> 3. Set data at memory address 27 to "Easter eggs"

That's only 3 steps. But how good would an array be for adding new elements somewhere in the middle. So instead of swapping potatoes for Easter eggs, I want to add Easter eggs. And I'm going to say that the array is in priority order, so if I don't have time to buy all the items I know that the most important ones are at the beginning of the list. And I *really* love Easter eggs, so much so that they come under apples in importance to me.

In order to add Easter eggs between apples and toothpaste, the program first has to make sure that there is enough space in the array to take another element. If our array is already full, the computer will need to find another block of contiguous memory addresses that has space for six elements, and copy across the existing five elements first. It then needs to 'make space' for the new element at index 1 by shifting down all the following elements one at a time:

> 1. Get value from index 4 (Bread)

> 2. Put Bread in index 5

> 3. Get value from index 3 (Newspaper)

> 4. Put Newspaper in index 4

> 5. Get value from index 2 (Potatoes)

> 6. Put Potatoes in index 3

> 7. Get value from index 1 (Toothpaste)

> 8. Put toothpaste in index 2

> 9. Finally got space in index 1, so put Easter eggs in index 1

And it's even worse than it looks. For each of these 9 steps, it has to do the 3 steps described above to actually get or set the data in each element. So that's 27 steps to add an extra element at index 1. Phew!

Now I'm going to describe another data structure that also holds sequences of data, and lets you reference individual elements by index, but which would only take 3 steps to add an element at index 1.

Lists

On the surface a list seems very similar to an array, and let's you do many of the same things with sequences of data. But the implementation of a list is very different, which gives it different performance characteristics to an array. Whereas an array needs to find a contiguous block of memory addresses to hold its elements, a list can store them all over the place.

Using our houses analogy, the first element of the list could be in House 4, the second in House 10, the third in House 2 etc. So if the list elements are spread out all over the place, how does the program know the addresses for the elements? In a similar way to an array, the program just needs to keep track of the address of the first element in the list. That's because each element in the list contains not only the data we want it to hold, but also the memory address of the next element.

In pseudocode, we could write out the implementation of a list using two classes, like this:

class ListElement {

val data = ???

val addressOfNextElement = ???

}

class List {

val addressOfFirstElement = ???

}

Now if we want to get access to the third element in the list, the computer can take the following steps:

> 1. Get the address of the first element from the List

> 2. Get the ListElement at that address (1st element)

> 3. Get the address of the next element from the ListElement

> 4. Get the ListElement at the next address (2nd element)

> 5. Get the address of the next element from the ListElement

> 6. Get the ListElement at the next address (3rd element)

> 7. Do whatever we want to do with the data in the 3rd ListElement

Whoa, that's a lot more steps to access an element than with an array. That's right, an array can consistently access any element in it in three steps, whereas the number of steps increases with increasing element index for a list. This is an example of a trade off, where some data structures can have worse performance than others for certain things, but then be better than them for other things. Let's show you the benefit lists have over arrays when inserting new elements.

Say we've got our shopping list of five items in a list rather than an array, and we want to add Easter eggs between apples and toothpaste. The steps would be like this:

> 1. Get the address of the first element from the List

> 2. Get the apples ListElement at that address

> 3. Create a new ListElement, with data = "Easter eggs", and set its addressOfNextElement to the address of the toothpaste ListElement (the addressOfNextElement from the apples ListElement we got in step 2). Apples and toothpaste will now both be pointing to Easter Eggs as their next element

> 4. Save the Easter eggs ListElement to memory, and take a note of its address

> 5. Set the apples ListElement to point to the Easter eggs ListElement (now apples will be pointing to Easter eggs, which will be pointing to toothpaste)

That was 5 steps for a list, versus 27 for an array. There's a lot more subtlety in the differences between arrays and lists, so deciding which one to choose for a particular use case can be a bit of an art. However, at this stage in your programming journey it's probably enough just to have an understanding that differences exist. Unless you're writing programs to crunch through massive amounts of data, you're unlikely to notice any difference in the performance of your programs.

Let's write our shopping list using a List instead:

object Program extends App {

val shoppingList = List[String]("Apples", "Toothpaste", "Potatoes", "Newspaper", "Bread")

println(shoppingList(3)) // Prints Newspaper

}

The syntax is pretty similar to Arrays, but note that you don't use the *new* keyword, and instead of passing in an integer to specify the size of the Array, we pass in all the elements of the List. To retrieve an element at a specific index, the syntax is the same as with an Array. There's a very important difference to note though: Arrays are mutable and Lists are immutable. That might sound a bit funny, as we used the word *val* to declare each of our shopping lists:

val shoppingListArray = new Array[String](5)

val shoppingListList = List[String](...)

Doesn't that mean they're both immutable? In actual fact what we're saying is that the *variables* *shoppingListArray* and *shoppingListList* are immutable. We're not saying anything about the mutability of the actual Array or List themselves. Let me demonstrate with a quick example:

val immutable = new List[Integer](1, 2)

immutable = new List[Integer](3, 4) // Won't work. We can't reassign something to a var

var mutable = new List[Integer](1, 2)

mutable = new List[Integer](3, 4) // This is fine. mutable is now a List containing (3, 4)

Ok, that showed how the *val* and *var* keywords applied to the mutability of the variables. Now let's see the difference in mutability between Arrays and Lists:

val array = new Array[Integer](2) \\ Using a val, so variable array is immutable

// Elements in the array are mutable. We're able to change them, like this:

array(0) = 1

array(0) = 2

// But we can't assign a new Array to the array variable

array = new[Array](3) // Won't work

var list = List[Int](1, 2) // Using a var, so variable list is mutable

// The list itself is immutable, so we can't change it

list(0) = 3 // Won't work

// But we can assign a new List to the list variable

list = List[Int](3, 4)

If you think about it, it makes sense. An Array is just a sequence of memory address. Once the Array has been created, those addresses don't change. So the addresses are immutable. But the program can access and change the data held in those memory addresses, so the data is mutable.

A List is more than a sequence of addresses. It is modelled as Scala classes (like our *List* and *ListElement*). The designers of Scala made the decision that the attributes of these classes should be *val*s, so once a *ListElement* is created we can't change its data or the address of the next element. This may seem restrictive, but it's actually a good choice for the language.

There are many benefits to keeping things immutable, and purely functional programs don't have mutability at all. However, different languages work differently, and if you pick up another language like Java you'll find that its Lists are mutable.

This makes for a bit of a conundrum. I said that one of the benefits of Lists was being able to insert new elements within them. And they won't be that useful if you can't potatoes to Easter eggs. But how can we do this if you're not allowed to alter the List?

The answer is to take your List and use it to make a new one. There are some really simple ways to do this, which we'll come to soon, but I'm actually going to show you a *hard* way first. No, it's not because I'm sadistic! It's because you'll learn about a really important and fundamental way of dealing with immutable data structures.

Recursion

The problem we're trying to solve is one of *iteration*, also called *loops*, or just *doing something repetitively*. Let's start with a really simple example. Say we want to print out "Hello World!" twice. We'd probably just write:

object Program extends App {

println("Hello World!")

println("Hello World!")

}

Seems ok. But what if we wanted to print it out a hundred times? Or a thousand? We definitely don't want to copy and paste that a thousand times! We're going to use a recursive method to get this working in less than 10 lines of code. The first thing to realise is that you can call methods from within other methods.

object Program extends App {

def printName(name: String) = printHello(name) // Calling printHello method from within printName

def printHello(x: String) = println(s"Hello $x")

printName("Ian")

}

The trick with a recursive method is that it calls *itself*. And because there's a call to itself within its body, every time it calls itself it calls itself again. And again and again in a loop. The other thing most recursive methods have is a way of knowing when to stop, otherwise they'll just keep looping forever. I know it can be a bit difficult to get your head round this, and see if you can make sense of it before reading the description below.

object Program extends App {

// You need to specify the return type for recursive methods.

// Here the method doesn't return anything, so we specify a return type of Unit

def loop(counter: Int): Unit = {

if (counter > 0) {

println("Hello World!")

loop(counter - 1)

}

}

loop(1000)

}

I find it easiest to understand methods like this by walking through the steps it takes in my head. To start with, we've got a method called *loop* that takes an integer as a method parameter. We've called the parameter *counter*, but it doesn't matter what it's called. The last line of the program is where we start everything off by actually calling our method. I don't want to step through 1000 loops in my head, so lets try and work out what happens if we call *loop(2)* instead.

> 1. The parameter *counter* is assigned the value 2

> 2. The if statement checks whether *counter* is greater than 0. 2 is greater than 0, so the body of the if statement is executed

> 3. We print out "Hello World!"

> 4. The method calls itself, passing in a value of *counter* minus 1. 2 minus 1 is 1, so it's calling *loop(1)*

> 5. The *loop* method is called passing in a value of 1, which is assigned to the *counter* parameter

> 6. 1 is greater than 0, so the body of the if statement executes

> 7. We print out "Hello World!"

> 8. The method calls itself passing in a value of 0

> 9. The method is called with a value of 0, which is assigned to the *counter* parameter

> 10. 0 is not greater than 0, so the body of the if statement is not executed

> 11. There's no more code in the method body, so the method finishes

Hopefully that makes sense. If not, go back and walk it through in your mind a few times. The two crucial things to remember about recursive methods are:

> 1. They call themselves

> 2. They have some condition that tells them whether to call themselves again or to stop

If you've got that, great! It's definitely not a beginner's concept, so well done!

We're going to use recursion to do some stuff with Lists now. They are great for doing something with the first element of the List, then calling themselves to do the same thing with the second element, then calling themselves for the third element, etc. And we can tell them to stop when they get to the end of the List. The first things we're going to do is to take a List of integers, and use recursion to add up all the elements of the List.

Don't feel that you need to remember all the syntax for working for data structures. If you forget whether it's *new List()*, *List()[String]* or whatever, it's very easy to look it up. It's more important to understand the concepts.

We're going to use some of the attributes built into Lists to help us. The first element of the List is represented by the **head** attribute, and the rest of the List is represented by the **tail** attribute. The following example will demonstrate this:

object Program extends App {

val wholeList = List[Int](1, 2, 3)

val listHead = wholeList.head

val listTail = wholeList.tail

println(wholeList) // Prints List(1, 2, 3)

println(listHead) // Prints 1

println(listTail) // Prints List(2, 3)

}

And we can use the *Boolean* attributes *isEmpty* and *nonEmpty* to check whether the List is empty or not, like so:

val emptyList = List[Int]()

println(emptyList.isEmpty) // Prints true

println(emptyList.nonEmpty) // Prints false

val listWithElements = List[Int](1)

println(listWithElements.isEmpty) // Prints false

println(listWithElements.nonEmpty) // Prints true

So now we've got all the tools we need to add up the elements of a List, using recursion to iterate through the elements, *head* and *tail* to deconstruct the List, and *isEmpty* or *nonEmpty* to create the condition in which the recursion ends.

We haven't written a test in this chapter yet, so let's take a test-driven approach to this. I'm going to create an object called *ListOps* in which we can put methods to operate on Lists, so go ahead and create a *ListOps* object and a *ListOpsSpec* test class. Our first test will look like this:

class ListOpsSpec extends FlatSpec with Matchers {

"sumList" should "return the sum of the elements in a List" in {

val list = List[Int](1, 2, 3)

ListOps.sumList(list) shouldBe 6

}

}

object ListOps {

def sumList(list: List[Int], total: Int = 0): Int = {

if (list.nonEmpty) {

val firstElement = list.head

val restOfList = list.tail

val newTotal = total + firstElement

sumList(tail, newTotal)

}

else {

total

}

}

}

Try stepping through this code in your head to understand what's going on. There's nothing new here that you haven't covered already. When you're ready we'll go through it together.

> 1. We're calling *sumList* with a single parameter - a List containing (1, 2, 3). This gets assigned to the *list* parameter

> 2. Because we're not passing in a second parameter, the *total* parameter takes the default value of 0

> 3. We check that the List is not empty. It isn't, so the body of the if statement gets executed

> 4. We deconstruct the List into two variables. *firstElement* is assigned the value 1, which is the head of the list. *restOfList* is assigned the tail of the List, which is a new List containing the elements (2, 3)

> 5. We add the first element to our running total, so *newTotal* is assigned the value 0 + 1

> 6. We call *sumList*, passing in the tail of the List and the *newTotal*. It's the equivalent of calling *sumList(List[Int](2, 3), 1)*

> 7. The List containing (2, 3) is assigned to the *list* parameter, and the value 1 is assigned to the *total* parameter

> 8. *list* isn't empty, so we go into the body of the if statement

> 9. We deconstruct the *list*. *firstElement* is assigned the value 2, *restOfList* is assigned a new List containing a single element (3), and *newTotal* is assigned 1 + 2 = 3

> 10. We call *sumList* again

> 11. *list* is assigned the List containing a single element (3), and *total* is assigned the value 3

> 12. *list* isn't empty, so *firstElement* is assigned the value 3, *restOfList* is assigned a new List with no elements, *newTotal* is assigned 3 + 3 = 6, and we call the method again

> 13. *list* is now empty, so *list.nonEmpty* is false. We don't execute the body of the if statement, but go to the else branch instead

> 14. The else branch just returns the value of the *total* parameter, which is 6

There's quite a lot going on there, but if you go through it line by line hopefully it makes some sense. I wrote it in quite a verbose way to make it easier to understand, but if you like you can do it in fewer lines of code by getting rid of the variable assignments within the if statement body:

def sumList(list: List[Int], total: Int = 0): Int = {

if (list.nonEmpty) sum(list.tail, total + list.head)

else total

}

Use whichever you prefer, but as you get used to it you'll probably end up using the more compact version.

See if you can write another recursive method to multiple the elements of a List. In other words, write a recursive method to make this test pass:

"multiplyList" should "return the multiple of the elements of a List" in {

val list = List[Int](1, 2, 3, 4)

ListOps.multiplyList(list) shouldBe 24

}

If you run into problems with it not giving you the right answer, you might find it helpful to print out the values of variables as it iterates by using *println*. For instance, putting a call to *println(total)* in the method will print out the running total each time the method is called, and will help you to check if it's behaving as you expect.

If you need a hint, don't assign a default value of 0 to the *total* parameter. Anything multiplied by 0 is 0.

Let's finish the chapter by implementing the same thing for Arrays. Unlike Lists, Arrays don't have *head* and *tail* attributes. We can only access the elements of Arrays by referring to them by index. So we're going to have to take a different approach. Start by creating an *ArrayOps* object and an *ArrayOpsSpec* test class. The tests will be pretty much the same as the tests we wrote for the *ListOps*, but let’s start with multiplication this time:

class ArrayOpsSpec extends FlatSpec with Matchers {

"multiplyArray" should "return the multiple of the elements of an Array" in {

val array = Array[Int](1, 2, 3, 4) // Using syntax to populate an Array on creation

ArrayOps.multiplyArray(array) shouldBe 24

}

}

You need to know one final thing before you can solve this. Arrays have an attribute called *length* which tells you how many elements they have.

val x = Array[Int](1, 2, 3, 4)

println(x.length) // Prints 4

// Arrays are zero indexed - the first element has index 0.

// So the last element has index of length - 1.

println(x(0)) // Prints 1

println(x(x.length - 1)) // Prints 4

// If you try to access an index that is outside the Array, you'll get an error

println(x(-1)) // Index too low, won't work

println(x(x.length)) // Index too high, won't work

If you feel up to it, try writing the implementation before reading on.

Remember, you can't deconstruct the Array into a head and a tail, so you'll need another way to iteratively access the next element of the Array each time you call the recursive method.

You've seen how we can pass in a running total as a method parameter. How about passing in a parameter representing the index of the element we want to access as well?

My solution looks like this:

object ArrayOps {

def multiplyArray(array: Array[Int], total: Int = 1, index: Int = 0): Int = {

if (index < array.length) multiplyArray(array, total + array(index), index + 1)

else total

}

}

Finish up by writing a test and implementation for *ArrayOps.sumArray*, then we're done. Awesome job!

Functions

So far we've been defining behaviour as methods on objects, using the *def* keyword. They may or may not take method parameters, and they may or may not return things. And all we can do with them is call them. Here are a few examples of what we've learnt about methods:

class SomeMethods {

// Return type is Unit, meaning it doesn't return anything to the caller

def sayHello(): Unit = println("Hello")

// Takes one parameter of type Int, and returns an Int to the caller

def timesTwo(x: Int): Int = x \* 2

// Takes two parameters of type Int, and returns an Int to the caller

def add(x: Int, y: Int): Int = x + y

// Calling a method on another object

// Note the syntax [object][period][method]

def pourMilkFrom(carton: MilkCarton, amount: Int): Unit = carton.pourMilk(amount)

// Calling another method within the same class

// Note you don't specify the class name or put a period, just call the 'add' method directly

def addThenTimesTwo(x: Int, y: Int): Int = {

val added = add(x, y)

timesTwo(added)

}

}

That's all well and lovely, but it would be really useful to be able to treat methods like variables. This would allow us, for instance, to pass methods into other methods as parameters, or have methods return other methods. I'm going to show you some really cool examples of this later in the chapter, but as a simple and slightly pointless example let's say that we want to have a *calculate* method that will take a number and another method describing the calculation as parameters. What I'm trying to do is something like this:

object WontWork {

def timesTwo(x: Int): Int = x \* 2

def square(x: Int): Int = x \* x

def calculate(number: Int, calculation: ???): Int = calculation(number)

}

class WontWorkSpec extends FlatSpec with Matchers {

"Calculate" should "run the calculation method passed into it" in {

WontWork.calculate(3, WontWork.timesTwo) shouldBe 6

WontWork.calculate(3, WontWork.square) shouldBe 9

}

}

Hopefully it's fairly clear from the test case what we're trying to do. If we pass a number and the *timesTwo* method into the *calculate* method, we want to assign the *timesTwo* method to the *calculation* method parameter. In the body of the *calculate* method we want to get the value of the *calculation* parameter, which is the *timesTwo* method, and then call that method passing in the number.

We can pass in the *square* method instead, and it will square the number. In fact, we can see that the *calculation* parameter is a method that itself takes a single parameter (an *Int*). And we can also see that the *calculation* method has to return an *Int*, as this is what the *calculate* method returns. So we're not restricted to passing in *timesTwo* and *square* methods. We should be able to pass in *any* method that takes an *Int* and returns an *Int*.

You may need to reread that and let it sink in. If you're still not quite sure about it, don't worry. It should make more sense as we work through the chapter.

We have a bit of a problem with our code. The name of the object may have given you a hint that it won’t work! We need to specify the types of method parameters (e.g. *number: Int*). So what's the type of the *calculation* parameter? It's actually a **function**.

Hmm, that's nice, but what's a function? Well, functions are virtually identical to methods, except you can treat them like variables, pass them into methods (or other functions), and have methods (or other functions) return them. The syntax is a little bit different to a method. I'll rewrite our *timesTwo* method as a function, then go through what's going on:

val timesTwo: Int => Int = x => x \* 2

Whoa, that looks complicated! Stay calm, and we'll break it down into easy chunks. Firstly you can see that this is a *val* rather than a *def*, and we know the syntax for *val*s:

val x: Int = 8

// [val] [name] [:] [type] [=] [value]

It's exactly the same for the function. So we have the word *val*, the name *timesTwo*, a `*:`*, the type *Int => Int*, an *`=`,* and the value *x => x \* 2*. The only really new thing here is the `*=>*` sign. In my mind, when I see `=>` I think, "goes to". So the type is a function of `Int => Int`, or "Int goes to Int". In other words, it's a function that takes an *Int* parameter, and returns an *Int*. This is called the **type signature** of the function.

The value is what gets executed when the function is called. In this example it's `*x => x \* 2*`, or "x goes to x \* 2". If you squint, it looks like a method, where the *x* on the left of the `=>` is the method parameter, and everything on the right of the `=>` is the method body.

Now we've got a function, it's very easy to call it. It looks just like calling a method:

object WillWork {

val timesTwo: Int => Int = x => x \* 2

val square: Int => Int = x => x \* x

val result1 = timesTwo(3) // result1 is 6

val result2 = square(3) // result2 is 9

}

Cool. Now we're got our functions that we want to pass into our *calculate* method we can fill in the type of the *calculation* parameter. The type is a function of `*Int => Int*`, and we write it like this:

def calculate(number: Int, calculation: Int => Int): Int = calculation(number)

So the whole thing now looks like this:

object WillWork {

val timesTwo: Int => Int = x => x \* 2

val square: Int => Int = x => x \* x

def calculate(number: Int, calculation: Int => Int): Int = calculation(number)

}

class WillWorkSpec extends FlatSpec with Matchers {

"Calculate" should "run the calculation method passed into it" in {

WillWork.calculate(3, WillWork.timesTwo) shouldBe 6

WillWork.calculate(3, WillWork.square) shouldBe 9

}

}

Here's another little trick. Look at how we're calling the *calculate* method in the test case. We're passing in the two parameters in different ways. The first parameter is just the value (*3*), whereas the second parameter is the name of variable that the function is assigned to (*WillWork.timeTwo*). It would be annoying if we couldn't just pass values directly into methods, as it would mean we would have to create variables for every parameter, like this:

"Calculate" should "run the calculation method passed into it" in {

val number = 3

WillWork.calculate(number, WillWork.timesTwo) shouldBe 6

}

We can do the same thing with functions, passing the value of the function directly into the method. Remember, the value of the function is the part on the right side of the `=`. Let's add some more method calls to the test:

class WillWorkSpec extends FlatSpec with Matchers {

"Calculate" should "run the calculation method passed into it" in {

WillWork.calculate(3, WillWork.timesTwo) shouldBe 6

WillWork.calculate(3, x => x \* 2) shouldBe 6 // passing in the value of the timesTwo function

WillWork.calculate(3, x => x \* x) shouldBe 9

// The function parameter doesn't have to be called x

WillWork.calculate(3, input => input + 5) shouldBe 8

// This won't work though

WillWork.calculate(3, input => "Hello World") shouldBe "Hello World"

}

}

Why doesn't the final statement work? Well we've defined the *WillWork.calculate* method to take a function which takes an *Int* and returns an *Int*, but we're calling the method with a function that takes an *Int* and returns a *String*.

Have a go at updating the *WillWork* object to make this test pass. Remember, you can *overload* methods with the same name, so long as they have different parameter types. And a function of "Int goes to Int" is a different type to a function of "Int goes to String".

This is my solution:

object WillWork {

// Don't need the twoTimes and square variables anymore

def calculate(number: Int, calculation: Int => Int): Int = calculation(number)

def calculate(number: Int, calculation: Int => String): String = calculation(number)

}

The functions we've seen so far have type signatures that take a single parameter, and return a single type. But there are other combinations too. Here are some examples of different type signatures:

// If your function doesn't take exactly 1 parameter, you have to wrap the parameters in parentheses.

// Here we're wrapping 0 parameters in parentheses.

val gimmeFive: () => Int = () => 2 + 3

// Here our function takes 2 parameters, so we've wrapped them in parenthesis, separated with a comma.

val add: (Int, Int) => Int = (x, y) => x + y

// This function takes no parameters, and returns nothing (Unit) to the caller.

val nada: () => Unit = () => println("Hello there!")

// This one takes a single String parameter, but returns nothing to the caller.

val printMe: String => Unit = me => println(me)

// Just to show you that you can use types other than String and Int. Here we return a Boolean.

val lengthOfStringEqualsNumber: (String, Int) => Boolean = (str, number) => str.length == number

// How about a Monster creator?

val monster: (String, Int) => Monster = (name, amount) => new Monster(name = name, health = amount)

// Or something a bit more complex?

val fight: (Player, Player) => Player = (player1, player2) => {

// Use curly braces if the function body takes more than one line

player1.attack(player2)

player2.attack(player1)

if (player1.health > player2.health) player1

else player2

}

How about you show me something useful?

I hope that was somewhat interesting, but I can understand if you're finding it hard to see the point of all this. Sure, as you get more experienced you'll start writing your own methods and functions that take or return functions. These are called *higher order functions* by the way. But let's start with some useful higher order functions that other people have written. In fact there are a tonne of them on classes that are built into Scala. I'm going to show you a higher order function called **map** that's defined on a class we already know about: the *List*.

Say we have a List of integers, and we'd like to do some kind of transformation on it to get a new List, where all the elements have been doubled. Here's a test case describing what we want:

"We" should "be able to double the elements in a list" in {

val originalList = List[Int](1, 2, 3)

val doubledList = ??? // Some code to create a new list with the elements doubled

doubledList shouldBe List[Int](2, 4, 6)

}

As I mentioned, the *List* class has a function called *map* which takes a single parameter: a function of type `A => B`, where `A` is the type of the elements in the List (in this case *Int*), and `B` is the type of the elements we want in our new List (also *Int* in this example). It goes through the List, applying this function to each element and creating a new List from the results.

We've already seen a function that takes an *Int*, returns an *Int*, and has the effect of doubling the input: our *timesTwo* function. Let's try passing that into the *map* function and see what happens:

"We" should "be able to double the elements in a list" in {

val timesTwo: Int => Int = x => x \* 2

val originalList = List[Int](1, 2, 3)

val doubledList = originalList.map(timesTwo)

doubledList shouldBe List[Int](2, 4, 6)

}

And there we go! Using a function to easily transform every element in a list. As you know, we can also pass a function value directly into the *map* function. Here's another test with everything condensed.

"We" should "be able to square the elements in a list" in {

List[Int](1, 2, 3).map(x => x \* x) shouldBe List[Int](1, 4, 9)

}

The type of the elements in the transformed list don't have to be the same as the type in the original list:

"We" should "be able to transform a list to have elements of a different type" in {

List[String]("Hello", "dear", "reader").map(x => x.length) shouldBe List[Int](5, 4, 6)

}

And *map* isn't the only higher order function defined for Lists. Oh no, there are plenty more. And they're also found on other data structures as well. Say we've got a List of strings, and we only want those over a certain length? There's a higher order function for that! It's called *filter*, and it takes a function with a type signature of `A => Boolean`, where `A` is the type of the elements in the List. It applies the function to each element in the List, and if the result of the function is *true* then the element gets added to the new List. Otherwise it's denied entry.

"We" should "be able to filter words longer than a certain value" in {

val input = List[String]("Hello", "dear", "reader")

input.filter(x => x.length > 4) shouldBe List[String]("Hello", "reader")

}

We're only scratching the surface here. If you want a whole load more information on Lists and the ways you can use them, a quick Google search for "Scala Lists" will get you started down the rabbit hole.

A great feature of these methods are that they don't alter the existing List, but return a new List. This means that we can call another method on the returned List in order to transform the original List in several ways. Let's say we have a List of words, and we'd like to know the total number of letters in the words that start with a letter that comes after "m" in the alphabet. We can start by filtering the List into a new List that only contains the words starting with a letter after "m".

"We" should "be able to find out the number of letters in words starting after m" in {

val originalList = List[String]("the", "cow", "jumped", "over", "the", "moon")

val filteredList = originalList.filter(word => word > "m")

filteredList shouldBe List[String]("the", "over", "the", "moon")

}

Then we can map the words to their lengths:

"We" should "be able to find out the number of letters in words starting after m" in {

val originalList = List[String]("the", "cow", "jumped", "over", "the", "moon")

val filteredList = originalList.filter(word => word > "m")

val mappedList = filteredList.map(word => word.length)

mappedList shouldBe List[Int](3, 4, 3, 4)

}

And to finish we just need to sum up the values in the *mappedList*. Remember our *ListOps* class from the last chapter? We had a handy method that does just that.

"We" should "be able to find out the number of letters in words starting after m" in {

val originalList = List[String]("the", "cow", "jumped", "over", "the", "moon")

val filteredList = originalList.filter(word => word > "m")

val mappedList = filteredList.map(word => word.length)

val totalWords = ListOps.sumList(mappedList)

totalWords shouldBe 14

}

We could have written one big function that takes in the List of words, iterates over the elements looking for words starting after "m", getting the size of the word, and adding it to counter of some sort. But that would be a very specific piece of code that is only good for this one use case. Using smaller functions and composing them together allows them to be used in many different cases.

You can think of the big function as a cast iron model of a car, and the smaller functions as Lego bricks that you could use to create a car, but could also create lots of other things as well. This concept is so important to the art of coding that it has its own name - the **Single Responsibility Principle**. There are nuances to this, but basically you can think of it as keeping your classes, objects, methods and functions small, and focussed on doing just one thing well.

Our test code works, but I feel it's a bit messy. I'd like to **refactor** the test. Refactoring just means changing (hopefully improving) the code, without affecting what the code actually does. In this instance, there are two things we can change to make the code more readable. Firstly, there's actually a method defined on the List class that will sum up the elements in a List, so we can use that instead of our own *sumList* method. And secondly, we don't need all these intermediate *val*s to assign the stages of the transformation to. We can just chain the methods together, like so:

"We" should "be able to find out the number of letters in words starting after m" in {

val originalList = List[String]("the", "cow", "jumped", "over", "the", "moon")

originalList.filter(word => word > "m").map(word => word.length).sum shouldBe 14

}

If it's hard to read all on one line, you can split the method calls onto separate lines, like this:

"We" should "be able to find out the number of letters in words starting after m" in {

val originalList = List[String]("the", "cow", "jumped", "over", "the", "moon")

originalList.filter(word => word > "m")

.map(word => word.length)

.sum shouldBe 14

}

That looks much nicer to me.

I just have to say one thing about the *sum* method. If you're eagle eyed, you may have noticed that it doesn't have any parentheses after it. Scala has a rule that if a method doesn't take any parameters and it returns a value, then you shouldn't use empty parentheses when you define or call the method. This is because a method that doesn't take any parameters and returns a value looks just like a variable to the caller of the method, so the caller shouldn't have to distinguish between whether they are calling a method or just referencing the value of a variable. This is what I mean:

object Demo {

val x = 10

def y() = 15

def z = 20

}

To use these, you would write:

Demo.x // 10

Demo.y() // 15

Demo.z // 20

From the user's point of view, they're all doing the same thing: giving us an integer. The user doesn't need to know that *y* and *z* are methods, so writing *z* without parentheses lets the user treat it the same as a variable.

Ok, we've seen some of the higher order methods operating on Lists of strings and integers. Well, they're not restricted to working on these built in types. We can just as easily use them to transform Lists of types that we've created ourselves. Maybe we're writing a game, and have a List to keep track of all the monsters. At some point in the game we want to get rid of all the monsters with low health. Here's a monster:

class Monster(name: String, health: Int)

And an object to handle our monsters:

object MonsterHandler {

def cullMonsters(monsters: List[Monster]): List[Monster] = ???

}

Let's write a test for our *cullMonsters* method:

class MonsterHandlerSpec extends FlatSpec with Matchers {

"cullMonsters" should "get rid of all monsters with health below 10" in {

val originalMonsters = List[Monster](

new Monster("Barry", 20),

new Monster("Helen", 5),

new Monster("Jimmy", 15))

// Let's check that we just have Barry and Jimmy after culling

MonsterHandler.cullMonsters(originalMonsters)

.map(monster => monster.name) shouldBe List[String]("Barry", "Jimmy")

}

}

Have a go at filling in the implementation for the *cullMonsters* method. Here's my solution:

object MonsterHandler {

def cullMonsters(monsters: List[Monster]): List[Monster] = {

monsters.filter(monster => monster.health > 10)

}

}

Maps and case classes

Now we've seen that Lists have useful methods built into them that allow us to transform the data in them, it's time to look at a slightly more complex data structure: the **Map**. Don't get confused between the *data structure* Map, with a big 'M', and the *method* map with a small 'm'. Although they sound the same, they're not related. In fact, you can call the map method on Map data structures, in the same way that you can call the map method on List data structures. It will all become clear shortly!

A Map is a data structure that stores a mapping between one value and another value. An example of a Map would be a phone book, where people are mapped to phone numbers. If you want to find someone's number, you look up their name in the book, and next to their name is their number.

It's perfectly possible to create a phone book using a good old List. In fact, that's what we're going to do first, before I tell you why Maps are better! We're going to model each pair of values (name and number) using an **Abstract Data Type**. This is just a class with data, but no methods, and helps us to reason about our program more easily by having things grouped and named nicely.

In Scala we use **case classes** to represent Abstract Data Types (ADTs). They're extremely similar to normal classes, except Scala adds some useful functionality to them that makes them good for using as ADTs. We'll see some of this functionality, such as pattern matching, later.

Create a new class in the *src/main/scala* directory called *PhoneBookEntry*, then replace the generated contents of the file with the following:

case class PhoneBookEntry(name: String, number: Int)

There's our ADT. It should look very familiar to you. It’s declared exactly the same way you'd declare a class, except it has the word *case* at the beginning, and there is no class body. Here's how you create an instance of your case class:

object Program extends App {

val x = PhoneBookEntry("Ian", 13452) // Note, don't use the 'new' keyword

val y = PhoneBookEntry(name = "Anna", number = 64584) // You can use named parameters as well

}

And we can access the values of our instances just like we do with regular classes:

object Program extends App {

val x = PhoneBookEntry("Ian", 13452)

val friendsName = x.name

val friendsNumber = x.number

}

So how would we go about using this to create our phone book? We know we're going to use a List, but the users of our phone book shouldn't have to care about how we've implemented it, so we're going to wrap the List inside another class. In that way, users should be able to call methods on our class, and if we decide to change the List to something else (say a Map) then the users won't have to change their code to make it work.

It's very common, and good practice, to hide the details of how your classes work and strictly control what the users of your class can see. I think it makes sense to call our class *PhoneBook*, and for this example it will already be populated with names and numbers, and users will only be able to look up people's numbers from it. Here's the functionality described in a test:

class PhoneBookSpec extends FlatSpec with Matchers {

"Querying the phone book with a name" should "return that person's number" in {

val phoneBook = new PhoneBook

phoneBook.lookup("Ian") shouldBe 13452

}

}

Note that we're modelling the numbers as *Int*s, so you can't start them with a zero.

If you're feeling brave, stop reading here and have a go at implementing the class yourself. Just make up a few names and numbers, but make sure to include the name/number pair needed to make the test pass.

You know everything you need to do it.

Go on...

OK, great! Hopefully you have something like this:

class PhoneBook {

val entries: List[PhoneBookEntry] = List(

PhoneBookEntry("Barry", 4637),

PhoneBookEntry("Jenny", 43256),

PhoneBookEntry("Rover", 986),

PhoneBookEntry("Ian", 13452),

PhoneBookEntry("Spock", 76438)

)

def lookup(query: String): Int = {

// There are many ways of doing this. Yours might be different to mine.

val listWithCorrectEntry = entries.filter(entry => entry.name == query)

val correctEntry = listWithCorrectEntry.head

correctEntry.number

}

}

However you implemented this, the very nature of a List means that we have to visit each element in order to find out whether it's the one we're looking for. The way I've done it, using the *filter* method, means that even if the entry we're looking for is the first one in the List the program will check every single element. There are ways of writing this so that it stops as soon as it finds a matching element, but still it's not a great solution. What if we've got a million entries in our book, and the person we're looking for happens to be at the end? That's a lot of wasted processing time. There's a better way of doing this, and I know that you've guessed it. Maps!

Remember that data is stored in memory, and if the program knows the memory address of the data it can access it directly. Maps work by knowing the memory addresses of all the elements in them, so they can get any element in one go. They use a trick called **hashing** to do this.

In this context, hashing is a clever way of taking some value and converting it into a memory address. Hashing the same value will always produce the same address. And hashing different values will usually produce different addresses.

We know that Maps map one thing to another (e.g. names to phone numbers). The first thing is called the **key**, and the thing the key maps to is called the **value**. So in our example, we're going to have a Map of names (keys) to phone numbers (values). When a key and value are added to a Map, the computer hashes the key to produce the memory address that the key and value should be stored at. And when you ask the Map for the value of a key, it again hashes the key to get the memory address so it can access the value directly. Neat!

The syntax for creating a Map is similar to that for a List, but one obvious difference is that a Map contains two types, rather than just one in a List. You write the types in square brackets, separated by a comma. The type of the key goes first, followed by the type of the value. So in our phone book the keys are *String*s and the values are *Int*s:

val entries: Map[String, Int] = ???

There are a couple of ways of specifying the elements of a Map. I like using the arrow syntax, as it makes it clear that a key is being mapped to a value. Recreating our phone book entries as a Map rather than a List, it looks like:

val entries: Map[String, Int] = Map(

"Barry" -> 4637,

"Jenny" -> 43256,

"Rover" -> 986,

"Ian" -> 13452,

"Spock" -> 76438

)

And the simplest way to query a Map is to pass the key to the Map in the same way that you'd pass the index to a List:

val friendsNumber = entries("Ian")

Now we can refactor our *PhoneBook* class to use a Map instead of a List, and note how we don't have to change the test as it wasn't aware of the implementation details of the class:

class PhoneBook {

val entries: Map[String, Int] = Map(

"Barry" -> 4637,

"Jenny" -> 43256,

"Rover" -> 986,

"Ian" -> 13452,

"Spock" -> 76438

)

def lookup(query: String): Int = entries(query)

}

Not only does that look nicer, but we get constant performance no matter how many entries there are in the phone book. Hooray!

Options

What should happen if we query our phone book for someone who's not in it? There are several ways we could handle this. It could return an error message of some sort. Or maybe it would just give a default number. Let's try it out and see what happens. Add a test for this scenario:

"Querying the phone book for someone who isn't in it " should "do something" in {

val phoneBook = new PhoneBook

phoneBook.lookup("Unknown person")

}

Run the test, and see what happens. You'll get some scary-looking error messages in the terminal, like this:

Exception in thread "main" java.util.NoSuchElementException: key not found: "Unknown person"

at scala.collection.immutable.Map$Map3.apply(Map.scala:170)

at ....

....

What's happened here is that the program has tried to look up a key in the Map and can't find it, so has no idea what to do. Because there's no obvious default way of dealing with this situation, it throws its hands up in the air and the program crashes. This is a common pattern for when you ask for values that can't be found, such as when you try to access the 10th element of a List that only has 9 elements. And it's extremely rare that having the program crash is actually the behaviour you'd want.

Scala has a nice data structure called an **Option**, that models values that may or may not be there. Using Options forces you to consider what should happen if a value isn't present.

You can think of an Option as a box, that either contains a value, or doesn't. We have to specify the type of value that the Option can contain, in a similar way to how we specify the type of values that a List or Map contain. For instance, an Option that could potentially contain an *Int* would be *Option[Int]*. Both the Map and the List data structures have a method called *get*, that will return an Option instead of a value directly. Unless you're absolutely sure that the key is present in your Map, or the index is available in your List, you should use this method to make sure your program doesn't crash.

Let's go ahead and change the *lookup* method in our *PhoneBook* class to use the *get* method:

def lookup(query: String): Int = entries.get(query)

That's not going to work yet. We've declared that the *lookup* method returns an *Int*, but *entries.get(query)* returns an *Option[Int]*. So we need to do something to convert the *Option[Int]* into an *Int*. This is where we're forced to decide what to do if the Option has no value. It's easy if it contains an *Int* - we just return the *Int*. If it doesn't contain a value I'm going to return a default value of 0. I'll show you three ways of doing that.

Firstly, we can use the *isDefined* method on the Option, which returns *true* if the Option contains a value, and *false* if it doesn't. Then once we know that the Option contains a value, we can use the *get* method on it to get the value out of it:

def lookup(query: String): Int = {

val optionalResult: Option[Int] = entries.get(query)

if (optionalResult.isDefined) optionalResult.get

else 0

}

Another way of doing this is to use the *getOrElse* method on the Option. You pass a default value into the method. The method will return the value in the Option if it contains one, otherwise it will return your default.

def lookup(query: String): Int = {

val optionalResult: Option[Int] = entries.get(query)

optionalResult.getOrElse(0)

}

The third way is to use pattern matching on the Option.

Pattern matching

Pattern matching is like a more powerful way of using *if* and *if else* statements. Here are two simple code snippets, each doing the same thing:

val x = 3

val ifResult = {

if (x == 1) "a"

else if (x == 2) "b"

else if (x == 3) "c"

}

// ifResult is "c"

val matchResult = x match {

case 1 => "a"

case 2 => "b"

case 3 => "c"

}

// matchResult = "c"

To create a pattern match, put the word *match* after the variable you want to compare things to, then within curly braces is the body of the match expression. The body consists of a series of *case* *statements*, which start with the word *case* followed by a value to compare with the match variable. If the match variable and the case value are the same, then the expression on the right hand side of the `=>` is evaluated.

In this example the expressions on the right of the `=>` signs are just strings, so when the value of *x*, which is 3, matches the case statement *case 3*, the whole match expression returns the string "c", which gets assigned to the *matchResult* variable.

The right hand sides of the case statements can be more complicated than just returning a value, and if they go over several lines then wrap them in curly braces, like so:

// You can match directly on a value, rather than assigning it to a variable first

val result = "Hello" match {

case "Hello" => {

println("Matched Hello")

1 + 2

}

case "Goodbye" => 4

}

// result gets assigned the value of 3

The pattern matching syntax can look a bit neater than loads of *if else* clauses, but the real power comes when you combine it with case classes. I said earlier that Scala adds some cool functionality to case classes, and this includes being able to use them in pattern matching.

I'll write an example using Monsters, Wizards and Players, before showing you how this relates to pattern matching Options. In our game, Wizards should be able to use their magic powers to scry on other characters. If they scry on a Monster, it should print out the Monster's name and health. The Player is magically shielded though, so if the Wizard scrys on it then it goes unnoticed.

I'm going to create a *Scryable* trait that the *Player* and *Monster*s can extend, and the *Wizard* will have a *scry* method that takes a single method parameter with a type of *Scryable*. Remember how we had a class for *Monster*s, as we wanted to be able to create many *Monster*s for the game, but an object for the *Player*, as there would only ever be a single *Player*? Well we're going to use a case class for our *Monster*s now, and a case object for our *Player*.

trait Scryable

case class Monster(name: String, health: Int) extends Scryable

case object Player extends Scryable

You can put all of that in the same file if you like, or else create separate files for the trait, *Monster* and *Player*. I'm just going to stick everything in the same file for this little example. Now let's create our *Wizard*.

trait Scryable

case class Monster(name: String, health: Int) extends Scryable

case object Player extends Scryable

class Wizard {

def scry(enemy: Scryable) = {

// Now we have to work out if the enemy is a Monster or the Player.

// If it's a Monster, print out the name and health.

// If it's the Player, print out "Nothing to see here..."

}

}

This is how we're going to call the method:

object Program extends App {

val wiz = new Wizard()

val monster = Monster("Barry", 20)

wiz.scry(monster) // Should print "Argh, you found Barry! I've got 20 health"

wiz.scry(Player) // Should print "Nothing to see here..."

}

Now all we need to do is fill in the *scry* method. You know we're going to use pattern matching. You'll also see a nice feature of pattern matching with case classes, whereby we can deconstruct the case class attributes into variables in one go. Let's see it in action:

class Wizard {

def scry(enemy: Scryable) = enemy match {

case Monster(n, h) => println(s"Argh, you found $n! I've got $h health")

case Player => println("Nothing to see here...")

}

}

We're matching against the actual underlying type of the *Scryable* that's being passed into the method. If it's the *Monster*, we assign the *Monster*'s attributes to the variables *n* and *h*. They get assigned in the same order that they were written when we defined the case class (*case class Monster(name: String, health: Int)*), so the value of the *name* attribute gets assigned to the *n* variable, and the value of the *health* attribute gets assigned to the *h* variable. We then use those variables with string interpolation on the right hand side of the case statement.

Have a go at running the program now, and the correct lines should be printed out.

Not only can we deconstruct case classes into their component attributes using pattern matching, but we can also match against specific attributes. Let's say there's a *Monster* called Jimmy, who the *Wizard* is friends with. If the *Wizard* scrys on the *Monster* with the name Jimmy, it should print out a friendly message instead. Here's how we could make that happen:

class Wizard {

def scry(enemy: Scryable) = enemy match {

case Monster("Jimmy", \_) => println("Alright matey, how's it going?")

case Monster(n, h) => println(s"Argh, you found $n! I've got $h health")

case Player => println("Nothing to see here...")

}

}

object Program extends App {

val wiz = new Wizard()

val monster = Monster("Barry", 20)

val friend = Monster("Jimmy", 20)

wiz.scry(monster) // Should print "Argh, you found Barry! I've got 20 health"

wiz.scry(Player) // Should print "Nothing to see here..."

wiz.scry(friend) // Should print "Alright matey, how's it going?"

}

I used an underscore in *case Monster("Jimmy", \_)*. This just means, "don't bother assigning this attribute to any variable. I'm not going to use it".

If you wanted to use Jimmy's health you could just name the variable *case Monster("Jimmy", jimmysHealth)*. Also, it's important to put the Jimmy case before the general *Monster* case. The computer goes through each case statement in order, and stops at the first one that matches.

All this is extremely similar to the way we can pattern match on Options. *Option* is like a trait. It has two concrete implementations that extend the trait. An Option that contains a value is represented by a case class called *Some*, which has a single attribute holding the value. An Option that doesn't contain a value is represented by the case object *None*.

There are some more advanced features you need to know to be able to understand the real implementation of Options, so the following is pseudocode. If you're interested in seeing the actual implementation, just create an *Option* in a program in IntelliJ and **Ctrl + Click** on it to go to the source file.

trait Option

case class Some(value: Int) extends Option

case object None extends Option

Here's an example of using pattern matching on the Option we get from our Map:

class PhoneBook {

val entries: Map[String, Int] = Map(

"Barry" -> 4637,

"Jenny" -> 43256,

"Rover" -> 986,

"Ian" -> 13452,

"Spock" -> 76438

)

def lookup(query: String): Int = entries.get(query) match {

case Some(986) => {

// For some reason we want to do something weird with this entry

println(s"You searched for $query. Adding 2 to the number")

986 + 2

}

case Some(number) => number // Just return the number

case None => 0 // The key wasn't found, so return a default value of 0

}

}

Congratulations! You've learnt a tonne of things in this chapter. We're going to build on everything we've learnt in the next chapter by creating a Top Trumps style card game - plus we'll get to actually take some user input into our programs!

Trumps!

Now we’re going to get on to some fun! We're finally going to build a program that actually does something interesting. We'll be using a lot of what we've covered before, as well as learning how to take input into a program, both from files and from users. So let's get on, and make a game!

The game

We're going to make a Top Trumps style card game. Unfortunately fancy graphics are beyond the scope of this book (although you'll be all set to learn how to use proper game engines after finishing it), so this will be a command line program that will be text based.

It will read in a file containing the details of all the cards, and create a representation of the deck of cards in memory. It will shuffle the cards and deal half to the player and half to the computer. Then it will draw the top card from the player's hand and print out the details of the card on screen.

The player will then be able to choose an attribute from the card to compare with the computer's top card. Whoever loses the round has their card removed, then the hands are shuffled and the next round is played. This goes on until one player loses all their cards, at which point the game will print out whether the player won or lost.

Start by creating a new project. You can call it whatever you like. I'm going to call mine *TrumpsGame*.

The model

We're going to use case classes to model the Abstract Data Types in our game. The most obvious thing we need to model for our card game is a **Card**. Our cards will have a character name, and values for the character's strength, intelligence and courage. An example might be:

> Name: Captain Kirk

> Strength: 8

> Intelligence: 7

> Courage: 9

This is really easy to model as a case class. Go ahead and create a new Scala Class in the *src/main/scala* directory called *Card*.

case class Card(name: String, strength: Int, intelligence: Int, courage: Int)

We'll create some more case classes later, but it's not super clear what we'll need yet, so that's enough for now.

Reading files

So far all of our programs have done exactly the same thing when we run them. There's been no way to change what they do by passing in external input. Well, all that's about to change. We're going to create a *comma-separated-values* file (CSV) that will contain the details of all the cards for the game. This means that we can easily change the cards by changing a file, rather than having to change code.

CSV files contain text fields with a specific structure. Each line in the file has a specific number of values, separated by commas. You can think of them as a spreadsheet written out as a text file, and in fact spreadsheet programs let you export spreadsheets as CSV files. So once you've learnt how to do this, you'll be able to read spreadsheets into your programs by converting them to CSVs and following this technique.

A common place to put resource files for programs is in a *resources* directory in your project. Create a directory called *resources* at the same level as the *src* folder in your project (*TrumpsGame/resources*). Do this by right clicking on the project name directory in the project view, and selecting **New → Directory**. Create a text file in your new directory, and call it *data.csv*, by right clicking on the *resources* directory and selecting **New → File**. Copy the following into your *data.csv* file:

Name, Strength, Intelligence, Courage

Kirk, 8, 6, 9

Picard, 6, 9, 9

Spock, 9, 9, 7

Data, 10, 10, 8

The first line is the header of the file, describing what each of the columns are. Column one is *Name*, column two is *Strength*, etc. Then each line after that describes one of the cards. To get the cards ready to play with we need to do several things:

1. Read the file into the program

2. Ignore the first line of the file

3. Go through each line of the file, creating a *Card* object from each one

This will give us a List of *Card*s. Then we need to:

4. Shuffle the cards

5. Deal the cards into two hands, one for the player and one for the computer

I'm going to start by writing a test. All of this logic is related, so we're going to put it in a single class called *CardLoader*. Create this class, and also a *CardLoaderSpec* test class.

I'm not going to start coding my program to load the file, as I know that we can convert a file into a List of strings, where each string is a line in the file, so I want to start by testing that we can convert a List of file lines into a List of *Card*s:

class CardLoaderSpec extends FlatSpec with Matchers {

val cardLoader = new CardLoader

"createCardsFromLines" should "ignore the header line, and create cards from the rest of the lines" in {

val lines = List(

"Name, Strength, Intelligence, Courage",

"One, 1, 2, 3",

"Two, 4, 5, 6")

cardLoader.createCardsFromLines(lines) shouldBe List(Card("One", 1, 2, 3), Card("Two", 4, 5, 6))

}

}

Hopefully that makes sense. I've made the assumption that we're going to be able to get a *List[String]* from the file, and so I'm testing that we can pass that List into a *createCardsFromLines* method and get back a *List[Card]*. Next thing to do is to write the *createCardsFromLines* method:

class CardLoader {

def createCardsFromLines(lines: List[String]): List[Card] = {

val linesWithoutHeader = lines.drop(1)

linesWithoutHeader.map { line =>

val values = line.split(",")

val trimmedValues = values.map(value => value.trim)

Card(trimmedValues(0), trimmedValues(1).toInt, trimmedValues(2).toInt, trimmedValues(3).toInt)

}

}

}

The *drop* method is part of the List class. It takes an integer and returns a new List with the specified number of lines removed from the start. Here we're dropping 1 line, so *linesWithoutHeader* is a List without the header line.

Next we're mapping over the List of strings. Each element in the List (i.e. a string) is assigned to the *line* variable. The *split* method is part of the String class, and lets us split up a string into a List of strings by cutting the string wherever it finds a delimiter character. In this instance we're cutting the strings every time we find a comma. Using this on the string "Kirk, 1, 2, 3" returns a *List[String]("Kirk", " 1", " 2", " 3")*.

Note that there's a space before each of the numbers. That's because there's a space after each of the commas in the original string. So we then map over each of these Lists and call the *trim* method on the strings. This just removes any spaces from the beginnings and ends of the strings.

Finally our *map* converts the original line into a *Card* by instantiating a *Card* case class with the values from the List of trimmed strings. Because we're only working with strings at the moment, we need to tell the program that some of those strings are actually integers, by calling the *toInt* method on them.

We should have a passing test. Make sure you understand what's going on here before you move on.

What do we do with a List of Cards? We need to shuffle them and deal out the hands. Each of the hands can just be a *List[Card]*, but I'd like to be able to pass the player cards and the computer cards around together, so I'm going to model a *Deck* class. Again, this is an ADT, so we'll use a case class like this:

case class Deck(playerCards: List[Card], computerCards: List[Card])

We need to be able to split a List of *Card*s into a *Deck*. There's a bit of logic worth testing here, to make sure that the player and the computer get the same number of cards, so we're going to create a test for a new method that will take a List of *Cards* and return a *Deck*. Add this to the *CardLoaderSpec* class:

"splitCards" should "evenly split an even number of cards" in {

val cards = List(Card("One", 1, 1, 1), Card("Two", 1, 1, 1), Card("Three", 1, 1, 1), Card("Four", 1, 1, 1))

val deck = cardLoader.splitCards(cards)

deck.playerCards shouldBe List(Card("One", 1, 1, 1), Card("Two", 1, 1, 1))

deck.computerCards shouldBe List(Card("Three", 1, 1, 1), Card("Four", 1, 1, 1))

}

You should be able to work out what's going on in this test. Here's the implementation of the method, in the *CardLoader* class:

def splitCards(cards: List[Card]): Deck = {

val start = 0

val mid = cards.length / 2

val end = cards.length

val playerCards = cards.slice(start, mid)

val computerCards = cards.slice(mid, end)

Deck(playerCards, computerCards)

}

Look! We're using another method built into the List class, *slice*, to create the player's hand from the first half of the List, and the computer's hand from the other half.

Notice that we're only testing that we can split an even number of cards into two evenly sized hands. We're not sure exactly what would happen if we start with an odd number of cards. If you like you can decide what you'd like to happen if we get an odd number of cards, write a test for it, then update the implementation to make the test pass. For instance, you could say that either the player or the computer gets the extra card. Or maybe one of the cards isn't used for the game. If you don't want to do this, then just be careful to make sure that your data file contains an even number of cards.

Those are all the tests we're going to write for this class. We also need to shuffle the cards, but it's hard to write a test for this that will always pass. We could test that the shuffled cards are in a different order to the original cards, but shuffling is a random process, and as such it's possible that sometimes the cards will be shuffled and the order will not change, which would mean our test would fail. So the final part of our *CardLoader* class will be a method that the *Game* will call to load the cards, shuffle them, and return a *Deck*:

import scala.util.Random

class CardLoader {

def loadCards(): Deck = {

val lines: List[String] = ??? // Read the data file into a List of strings

val cards = createCardsFromLines(lines)

val shuffledCards = Random.shuffle(cards)

splitCards(shuffledCards)

}

// Other methods omitted

}

We're using a handy class from the *scala.util* package called *Random*. It has a method called *shuffle* which takes a List and returns a new List with the elements shuffled. Nice!

Follow the code through and check that you understand it. We haven't implemented the part that reads the file yet, but assuming that is going to work, we are then converting our List of file lines to a List of *Card*s, shuffling those cards, then splitting them into a *Deck*.

Actually reading a file and converting the lines of the file into a List of strings is actually very easy. We need to import another class from the scala library, called *Source*. This has a method called *fromFile* which takes a string with the location and name of the file to read. Here's how we use it:

import scala.io.Source

val lines: List[String] = Source.fromFile("resources/data.csv").getLines.toList

There are some intermediate steps you don't really need to know about, but if you're interested, the *fromFile* method returns an object of type *BufferedSource*. The *BufferedSource* type has a method called *getLines* which returns an object of type *Iterator[String*]. And the *Iterator[String]* type has a method called *toList*, which returns the *List[String]*.

Rather than hard coding the file name in this class, I'd like to make the class as generic as possible, so we're going to let the caller of the method pass in the name of the file as a method parameter. I've changed the name of the method from *loadCards()* to *loadCardsFrom(file: String)* to make this clearer. The complete class looks like this:

import scala.io.Source

import scala.util.Random

class CardLoader {

def loadCardsFrom(file: String): Deck = {

val lines: List[String] = Source.fromFile(file).getLines.toList

val cards = createCardsFromLines(lines)

val shuffledCards = Random.shuffle(cards)

splitCards(shuffledCards)

}

def createCardsFromLines(lines: List[String]): List[Card] = {

val linesWithoutHeader = lines.drop(1)

linesWithoutHeader.map { line =>

val values = line.split(",")

val trimmedValues = values.map(value => value.trim)

Card(trimmedValues(0), trimmedValues(1).toInt, trimmedValues(2).toInt, trimmedValues(3).toInt)

}

}

def splitCards(cards: List[Card]): Deck = {

val start = 0

val mid = cards.length / 2

val end = cards.length

val playerCards = cards.slice(start, mid)

val computerCards = cards.slice(mid, end)

Deck(playerCards, computerCards)

}

}

Getting things going

Ok, we've modelled our cards and deck, and have a nice tested class that lets us load cards from a file. I think we're ready to write the entry point to our program and see something working. To start with, we'll just use the *CardLoader* to get us a *Deck*, print out a message welcoming the player to the game, and then describing the player's first card to them. We need a new file called *Game.scala* in the *src/main/scala* directory, in which we'll have a *Game* object extending the *App* trait:

object Game extends App {

}

As you know, this is where the program will start running from. We need to add the functionality described above:

object Game extends App {

val cardLoader = new CardLoader

println("Let's play!")

val deck = cardLoader.loadCardsFrom("resources/data.csv")

val card = deck.playerCards.head

println(s"You've drawn ${card.name}")

}

Run the program a few times. You should get different cards, because the deck is being shuffled.

It's completely logical

Now we're going to get into the really interesting part of the program. Implementing the game logic. This is where we need to keep playing rounds until one player runs out of cards. We'll need the ability for the player to select an attribute to play, then the ability to compare the value of the player's attribute with the same attribute on the computer's card. The logic should be able to tell whether the player or the computer won the round, and discard the losing card from the relevant hand. It will then need to see whether the game is over, and if it is then display either a winning or a losing message. If the game isn't over then it will need to shuffle both players' cards and play another round.

This sounds like it could be pretty complex, so I'm going to neatly wrap up this functionality in its own class. We'll call this the *GameLogic* class. I want a recursive method that the *Game* object can call to start the first round. It will then keep on calling itself each round until the game is over.

Once the game is over, it will return some kind of status to the *Game* object, so that it can decide how to notify the player whether they won or lost. A nice way to model statuses like this is to use case objects. They don't need to be case classes, as there will just be a single instance to represent each status. And we can use them in a nice way for pattern matching. We'll have a *Status* trait which the case objects *Win* and *Lose* will extend. Put this in a new file called *Status.scala*:

sealed trait Status

case object Win extends Status

case object Lose extends Status

Saying this is a *sealed* trait means that anything extending the trait has to be in the same file. Normally you can extend traits in different files to where the trait is written (just like our *Game* object isn't in the same file as the *App* trait). Sealed traits are useful for when you are going to use them in pattern matching, as the computer then knows all the possible cases that could be matched and will warn you if you forget to cover a case.

We now need a *GameLogic* class:

class GameLogic {

}

and a test file for it:

import org.scalatest.\_

class GameLogicSpec extends FlatSpec with Matchers {

}

We'll start by writing two straightforward tests for our *play* method. The method will take the player's cards and the computer's cards as parameters. We only want the method to return a value once the game is over, at which point we want to know the *Status* as well as how many cards the winner has left, so we can print out a message like, "Hooray! You won with 2 cards left!"

Let's create another case class to represent these two things so we can pass them around together:

case class GameResult(status: Status, cardsLeft: Int)

So the definition of our *play* method should look like this:

def play(playerCards: List[Card], computerCards: List[Card]): GameResult = ???

The tests for this method are that it returns the right *GameResult* depending on whether the player or the computer has won. What does it look like when one of the players has won? It's when the other player has no cards left. So we know that if we call the method and one of the parameters is an empty list, then the game is over. Here are the tests:

import org.scalatest.\_

class GameLogicSpec extends FlatSpec with Matchers {

val gameLogic = new GameLogic

"If the computer runs out of cards it" should "return Win and the number of cards the player has left" in {

gameLogic.play(List(Card("Player", 1, 1, 1)), List[Card]()) shouldBe GameResult(Win, 1)

}

"If the player runs out of cards it" should "return Lose and the number of cards the computer has left" in {

gameLogic.play(List[Card](), List(Card("Player", 1, 1, 1))) shouldBe GameResult(Lose, 1)

}

}

Try writing the implementation of the method to make these tests pass.

Here's my solution:

class GameLogic(playerInput: PlayerInput) {

def play(playerCards: List[Card], computerCards: List[Card]): GameResult = {

if (computerCards.isEmpty) GameResult(Win, playerCards.length)

else if (playerCards.isEmpty) GameResult(Lose, computerCards.length)

else GameResult(Win, 2) // Remove this line in a minute

}

}

We need that last *else* just to make the program compile. We've declared that the method will return a *GameResult*, so if we didn't have that final statement and both the player and the computer had cards it wouldn't return anything. We'll remove it in a minute.

We can actually add the call to the *play* method from our *Game* object now, then deal with the end game conditions by pattern matching on the *GameResult*:

object Game extends App {

val cardLoader = new CardLoader

val gameLogic = new GameLogic

println("Let's play!")

val deck = cardLoader.loadCardsFrom("resources/data.csv")

gameLogic.play(deck.playerCards, deck.computerCards) match {

case GameResult(Win, cardsLeft) => println(s"You won, with $cardsLeft cards left!")

case GameResult(Lose, cardsLeft) => println(s"Sorry, you lost. The computer has $cardsLeft cards left")

case \_ => println("Whoops, that was unexpected!")

}

}

The `*case \_*` is the default matcher that will get executed if the other cases don't match. In this instance we know that one of the other two cases will match (the player will either win or lose), so we add this just to keep the compiler happy.

If you run the program now, the player should win with 2 cards left. Can you see why?

Now is the time to replace that *else GameResult(Win, 2)* statement with what we actually want to happen if both players have cards. This is what should actually happen:

> Print out the details of the player's top card

> Ask the player to select an attribute to play

> Compare the value of that attribute with the equivalent attribute on the computer's top card

> Decide who won the round

> If the player won, then discard the computer's top card

> If the computer won, then discard the player's top card

> If it's a draw, then keep all the cards

> Shuffle both player's cards, and call the play method again with them

Ah, there's a condition in which the players can draw! This is easy to handle. Let's just add a new case object that extends *Status*:

sealed trait Status

case object Win extends Status

case object Lose extends Status

case object Draw extends Status

We're going to have some logic for comparing the player and computer attributes to work out who won the round, so I think we'll have a method for this. Let's call it *compareAttributes*. It can go in the *GameLogic* class, and here are the tests for it in the *GameLogicSpec* class:

"If the player attribute is greater than the computer attribute it" should "return Win" in {

gameLogic.compareAttributes(2, 1) shouldBe Win

}

"If the player attribute is less than the computer attribute it" should "return Lose" in {

gameLogic.compareAttributes(1, 2) shouldBe Lose

}

"If the player attribute is the same as the computer attribute it" should "return Draw" in {

gameLogic.compareAttributes(2, 2) shouldBe Draw

}

Again, have a go at implementing this method to make the test pass before reading on.

This is my approach:

class GameLogic {

def compareAttributes(playerAttribute: Int, computerAttribute: Int): Status = {

if (playerAttribute > computerAttribute) Win

else if (playerAttribute < computerAttribute) Lose

else Draw

}

// Other methods omitted

}

I'm going to replace the dodgy *else* statement with a method called *getPlayerChoiceAndCompareCards*. It will take the player's cards and the computer's cards as parameters, will deal with displaying the card details to the player, getting the player input, and comparing the card attributes. Then it will return the *Status* of the round. In fact, knowing what the inputs to and output from the method will be, we can finish writing our *play* method:

def play(playerCards: List[Card], computerCards: List[Card]): GameResult = {

println("-------------") // A simple visual way to separate the rounds

if (computerCards.isEmpty) GameResult(Win, playerCards.length)

else if (playerCards.isEmpty) GameResult(Lose, computerCards.length)

else {

getPlayerChoiceAndCompareCards(playerCards, computerCards) match {

case Win => play(Random.shuffle(playerCards), Random.shuffle(computerCards.tail))

case Lose => play(Random.shuffle(playerCards.tail), Random.shuffle(computerCards))

case Draw => play(Random.shuffle(playerCards), Random.shuffle(computerCards))

}

}

}

So now we can see how the recursive loop of the game works. Given both players have cards, we call the *getPlayerChoiceAndCompareCards* method. This will let us know who won the round. It will then start the next round by shuffling the cards, with the losing card removed if it wasn't a draw, and calling the *play* method again.

This will go on, round and round, until one of the end conditions is true (one of the players has no cards left), at which point a *GameResult* is returned to the *Game* object. The *GameResult* is used in a pattern match in the *Game* object, and the winning or losing message is printed out.

All we need to do now is fill in the code for the *getPlayerChoiceAndCompareCards* method. We're going to import the import the *scala.io.StdIn* class so we can use the *readInt* method. This waits for the user to enter a number in the terminal and press **Enter**, then reads that number into the program as an integer. Be careful though - if it can't turn the input into an integer (such as the string "Hello") then the program will crash. I'm also going to create another case class to hold the values of the attributes we want to compare, just for convenience sake. It looks like this:

case class AttributeValues(player: Int, computer: Int)

Here's the whole *GameLogic* class with the *getPlayerChoiceAndCompareCards* method implemented:

import scala.util.Random

import scala.io.StdIn.\_

class GameLogic(playerInput: PlayerInput) {

def play(playerCards: List[Card], computerCards: List[Card]): GameResult = {

println("-------------")

if (computerCards.isEmpty) GameResult(Win, playerCards.length)

else if (playerCards.isEmpty) GameResult(Lose, computerCards.length)

else {

getPlayerChoiceAndCompareCards(playerCards, computerCards) match {

case Win => play(Random.shuffle(playerCards), Random.shuffle(computerCards.tail))

case Lose => play(Random.shuffle(playerCards.tail), Random.shuffle(computerCards))

case Draw => play(Random.shuffle(playerCards), Random.shuffle(computerCards))

}

}

}

def getPlayerChoiceAndCompareCards(playerCards: List[Card], computerCards: List[Card]): Status = {

val playerCard = playerCards.head

val computerCard = computerCards.head

println(s"You've drawn ${card.name}")

println(s"Enter 1 to play Strength ${card.strength}")

println(s"Enter 2 to play Intelligence ${card.intelligence}")

println(s"Enter 3 to play Courage ${card.courage}")

print("> ")

val choice = readInt()

val attributeValues = choice match {

case 1 => AttributeValues(playerCard.strength, computerCard.strength)

case 2 => AttributeValues(playerCard.intelligence, computerCard.intelligence)

case 3 => AttributeValues(playerCard.courage, computerCard.courage)

}

val attribute = choice match {

case 1 => "Strength"

case 2 => "Intelligence"

case 3 => "Courage"

}

val status = compareAttributes(attributeValues.player, attributeValues.computer)

status match {

case Win => println(s"You beat ${computerCard.name}, who had $attribute of ${attributeValues.computer}")

case Lose => println(s"${computerCard.name} beat you with $attribute of ${attributeValues.computer}")

case Draw => println(s"It's a draw. ${computerCard.name} had the same $attribute as you.")

}

status

}

def compareAttributes(playerAttribute: Int, computerAttribute: Int): Status = {

if (playerAttribute > computerAttribute) Win

else if (playerAttribute < computerAttribute) Lose

else Draw

}

}

And there we go. You should have a fully functional game! Give it spin.

That *getPlayerChoiceAndCompareCards* method is a bit long for my liking. I'll leave it as an exercise for you to have a go at refactoring it into smaller methods, with each focussed on doing a single thing.

After that have a play around with the program. Try making some changes, such as adding more attributes to the cards.

Also see if you can find any problems with the game and fix them. For instance, what happens if a player enters a choice that's not 1, 2 or 3? You might have to make some decisions about what happens in unexpected cases. That's part of the fun of programming!